Practical End-to-End Verification of Cyber-Physical Systems

Rose Bohrer

CMU-CS-21-115

May 2021

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
André Platzer, Chair
Stefan Mitsch
Frank Pfenning
Bradley Schmerl
Tobias Nipkow (TU Munich)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by NDSEG, the National Science Foundation under grant numbers CNS-1054246 and CNS-1739629, the Department of Transportation under grant number DTRT12GUTC11, the Department of Defense under grant number DTRT57-15-D-30011, the Air Force under grant numbers FA87501220291 and FA95501610288, the Alexander von Humboldt Foundation, and a Siebel Scholarship.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: formal methods, cyber-physical systems, theorem proving, hybrid systems, end-to-end verification, constructive logic, game logic, hybrid games

For my family

Abstract

Cyber-physical systems (CPSs) combining discrete control and continuous physical dynamics are pervasive in modern society: examples include driver assistance in cars, industrial robotics, airborne collision avoidance systems, and the electrical grid. Many of these systems are safety-critical because they operate in close proximity to humans. Formal safety verification of these systems is important because it is a key tool for attaining the strongest possible safety guarantees.

Hybrid systems models, in particular, are a successful formalism for CPS. Hybrid systems theorem-proving in differential dynamic logic (dL) and its generalization differential *game* logic (dGL) are notable for strong logical foundations and successful application in case studies using the theorem provers KeYmaera and KeYmaera X. However, safety verification of models does not imply safety of *implementations*, which might not be faithful to the model. Moreover, a machine-checked proof is only as trustworthy as the software which checks it, thus correctness of proof-checkers is crucial.

This thesis addresses implementation and soundness gaps by using constructive logic and programming languages as the foundation of an end-to-end verification toolchain. That is: *Constructive Differential Game Logic (CdGL) enables practical, end-to-end verification of cyber-physical systems.* CdGL enables synthesis of implementations with bulletproof theoretical foundations. Logic is the keystone of the end-to-end connection from high-level proofs and foundations to implementations. Our pursuit of practical proving includes innovations in the proof language itself.

CdGL proofs, in contrast to dGL, are suitable for synthesizing *controllers* which determine safe actions for a CPS and *monitors* which check the compliance of the external environment with model assumptions. The synthesized code is automatically proven correct down to machine-code level. The foundations are also strengthened by our formalization of classical dL's soundness in Isabelle/HOL, allowing hybrid systems proofs in dL to be exported and rechecked. We evaluate the toolchain on a 2D robot which follows arcs. The model and implementation cross-validate each other: monitors catch incorrect code and assumptions, while testing with monitors enabled allows us to assess the realism of the model.

Acknowledgments

Thanks to the committee for reading everything you read. To Dan Licata and Karl Crary: without you I would have never even started this PhD. Special thanks to all my family, therapists, doctors, labmates, and clubmates that have supported me; you are too many to list.

Contents

1 Introduction **1**
- 1.1 Outline and Contributions . . . 8
- 1.2 Related Work . . . 9
  - 1.2.1 Verification of CPS . . . 9
  - 1.2.2 Synthesis . . . 15

I End-to-End Verification of Classical Hybrid Systems **23**

2 Formalization of Classical dL **25**
- 2.1 Isabelle/HOL Primer . . . 28
- 2.2 Syntax . . . 31
  - 2.2.1 Term Syntax . . . 32
  - 2.2.2 Differential Program Syntax . . . 35
  - 2.2.3 Hybrid Program Syntax . . . 36
  - 2.2.4 Formula Syntax . . . 37
  - 2.2.5 Example Model . . . 38
- 2.3 Semantics . . . 39
  - 2.3.1 Term Semantics . . . 40
  - 2.3.2 Formula Semantics . . . 43
  - 2.3.3 Hybrid Program Semantics . . . 45
- 2.4 Static Semantics . . . 48
- 2.5 Axioms . . . 51
- 2.6 Rules . . . 54
  - 2.6.1 Renaming Rules . . . 56
  - 2.6.2 Substitution Rule . . . 56
  - 2.6.3 Isabelle/HOL Formalization . . . 60
- 2.7 Soundness Proof . . . 67
  - 2.7.1 Static Semantics Proofs . . . 68
  - 2.7.2 Adjoints and Substitution . . . 68
- 2.8 Proof Checker . . . 69
- 2.9 Code Generation . . . 73
- 2.10 Discussion . . . 76
  - 2.10.1 KeYmaera X Soundness Bug . . . 77

2.10.2 Implications for Developing and Formalizing Provers 79
2.11 Related Work 81

3 Monitor Synthesis for Classical Hybrid Systems 83
3.1 Related Work 89
3.2 1D Robot Example 91
3.3 ModelPlex Sandbox Synthesis 93
3.3.1 Controller Monitor Formula 93
3.3.2 Plant Monitor Formula 96
3.3.3 Fallback Control 97
3.3.4 Provably Safe Sandboxing 97
3.4 Interval Word Arithmetic Translation 98
3.5 Sandbox Implementation in CakeML 105
3.5.1 CakeML Sandbox 105
3.5.2 CakeML FFIs 106
3.5.3 Verifying the CakeML Sandbox 109
3.6 Hardware Evaluation 110
3.7 2D Robot Case Study 115
3.7.1 2D Robot Model 115
3.7.2 Proofs 120
3.7.3 Simulations 121
3.8 Discussion 125
3.8.1 Bellerophon Language Primer 125
3.8.2 Bellerophon Proof Script Examples 126
3.8.3 Limitations of Classical VeriPhy 130

II Constructive Game Logics 137

4 Constructive Discrete Game Logic 139
4.1 Introduction 139
4.2 Player Terminology and Player Constructivity 143
4.3 Related Work 145
4.4 Syntax 148
4.5 Example Games 151
4.6 Semantics 154
4.6.1 Realizers 155
4.6.2 Formula and Game Semantics 160
4.7 Proof Calculus 170
4.8 Theory: Soundness 179
4.9 Operational Semantics 182
4.10 Theory: Constructivity 188
4.11 Summary 192

5 Constructive Differential Game Logic **193**
5.1 Introduction . . . 193
5.2 Related Work . . . 195
5.2.1 Syntax of CdGL . . . 196
5.2.2 Example Game . . . 199
5.3 Type-theoretic Semantics . . . 201
5.3.1 Type Theory Assumptions . . . 202
5.3.2 Semantics of CdGL . . . 203
5.3.3 Connecting CdGL to dGL . . . 208
5.4 Proof Calculus . . . 210
5.4.1 First-order Arithmetic Proofs . . . 211
5.4.2 ODE Proofs . . . 213
5.5 Theory: Soundness . . . 215
5.6 Theory: Extraction and Execution . . . 218
5.7 Summary and Discussion . . . 221

6 Refining Constructive Hybrid Games **223**
6.1 Related Work . . . 225
6.2 Constructive Differential Game Logic . . . 227
6.2.1 Syntax . . . 227
6.2.2 Example Game . . . 227
6.3 Type-Theoretic Semantics . . . 228
6.4 Refinement Proof Calculus . . . 230
6.5 Theory . . . 234
6.5.1 Soundness . . . 234
6.5.2 Reification . . . 235
6.6 Discussion . . . 243

III Proof-Structured Synthesis **245**

7 Structured Proofs for Dynamic Logics **247**
7.1 Related Work . . . 249
7.2 Relationship of Kaisar to CdGL . . . 254
7.2.1 Kaisar Strategy Language . . . 255
7.2.2 Connecting Kaisar, Games, and Systems with Refinement . . . 257
7.3 Kaisar by Example: Proving Hybrid Games . . . 259
7.3.1 Core Propositional Connectives . . . 260
7.3.2 Unstructured Proof Steps . . . 263
7.3.3 Verifying Discrete Programs . . . 264
7.3.4 Verifying Hybrid Games . . . 265
7.3.5 Uniform Ghost Reasoning . . . 269
7.3.6 Static Single Assignment . . . 272
7.3.7 Time-Traveling Proofs with Labeled Reasoning . . . 277

7.3.8 Proof Patterns for CPS . . . 282
7.4 Implementation . . . 290
7.4.1 Constructive Arithmetic . . . 290
7.4.2 Contexts . . . 291
7.5 Results and Evaluation . . . 292
7.6 Summary . . . 297

8 Proof-Directed Game Synthesis **299**
8.1 Introduction . . . 299
8.1.1 Limitations of Classical VeriPhy . . . 299
8.1.2 Justifications for Limitations of Classical VeriPhy . . . 301
8.1.3 Goals of Constructive VeriPhy . . . 302
8.1.4 Limitations of Constructive VeriPhy . . . 302
8.2 Design . . . 303
8.2.1 Workflow and Pipeline Outline . . . 304
8.2.2 Correctness Argument . . . 305
8.3 Implementation . . . 310
8.3.1 Sandbox Generation . . . 310
8.3.2 The ProofPlex Algorithm . . . 312
8.3.3 Data Structures . . . 313
8.3.4 Execution . . . 320
8.4 Synthesis Considerations for Modeling . . . 322
8.5 Results Comparison . . . 324
8.5.1 GoPiGo Results . . . 324
8.5.2 AirSim Results . . . 330
8.5.3 Lessons Learned . . . 333

9 Conclusion **337**

A Appendices to Chapter 4 **343**
A.1 Proof Calculus . . . 343
A.2 Full Operational Semantics . . . 346
A.3 Example Proofs . . . 352
A.4 Theory Proofs . . . 356
A.4.1 Preliminaries . . . 356
A.4.2 Static Semantics . . . 357
A.4.3 Realizers . . . 357
A.4.4 Repetition . . . 357
A.4.5 Repetition Realizers and Monotonicity . . . 367
A.4.6 Soundness . . . 374
A.4.7 Proof Theory . . . 405

B Appendices to Chapter 5 **423**
B.1 Example Proofs . . . 423
B.1.1 Proof Overview . . . 423
B.1.2 Algebraic Derivations . . . 425
B.1.3 Natural Deduction Proof . . . 430
B.2 Theory Proofs . . . 436
B.2.1 Preliminaries and Assumptions . . . 436
B.2.2 Notations and Proof Style . . . 438
B.2.3 Static Semantics. . . . 439
B.2.4 Proofs of Stated Results . . . 441

C Appendix to Chapter 6 **477**
C.1 Theory Proofs . . . 477
C.1.1 Notations and Preliminaries . . . 477
C.1.2 Properties of Refinement . . . 478
C.1.3 Substitution . . . 480
C.1.4 Soundness . . . 482
C.1.5 Reification . . . 489

D Appendices to Chapter 7 **501**
D.1 Kaisar and Bellerophon Case Studies . . . 501
D.1.1 PLDI . . . 501
D.1.2 IJRR: Theorem 1 . . . 510
D.1.3 RA-L: Theorem 1 . . . 516

References **539**

List of Figures

1.1 Goals of the thesis. 7

2.1 Axioms of dL. 53
2.2 Axiomatic rules. 55
2.3 Selected classical sequent calculus rules. 55
2.4 Renaming rules. 56
2.5 Uniform substitution algorithm. 58

3.1 High assurance artifacts and steps in the VeriPhy verification pipeline. . . 85
3.2 End-to-end proof chain for end-to-end result. 88
3.3 Sandbox controller overview. 94
3.4 ModelPlex controller monitor synthesis. 95
3.5 Sandbox of a velocity-controlled ground robot. 98
3.6 Interval arithmetic for executable dL, helper functions. 100
3.7 Interval arithmetic for executable dL, terms. 101
3.8 Interval arithmetic for executable dL, formulas. 102
3.9 Interval arithmetic for executable dL, programs. 102
3.10 Controller sandbox, simulated plant. 114
3.11 Controller sandbox, real robot. 115
3.12 2D driving model: system variables. 116
3.13 Trajectories of dynamics for different choices of k. 116
3.14 Trajectories of plant for choices of $\mathsf{k} > 0$ when $\varepsilon = 1$. 117
3.15 Controller model for 2D circular driving. 118
3.16 Annular section through the (blue) waypoint $(2.5, 2.5)$. 118
3.17 Relative model specifies non-unique motion. 119
3.18 Implementation and environments built in AirSim. 123
3.19 Outline: VeriPhy sandbox safety tactic. 131

4.1 CGL proof calculus: propositional rules. 172
4.2 CGL proof calculus: some non-propositional rules. 173
4.3 CGL proof calculus: first-order games. 174
4.4 CGL proof calculus: loops. 175
4.5 Operational semantics: β-rules. 183
4.6 Operational semantics: monotonicity rules. 185
4.7 Operational semantics: commuting conversion rules. 186

4.8 Operational semantics: commuting conversion rules (contd.) 187
4.9 Operational semantics: structural rules. 188

5.1 Safe driving envelope. 200
5.2 CdGL proof calculus: ODEs. 212

6.1 Refinement of discrete connectives. 231
6.2 Algebraic rules. 233
6.3 Differential equation refinements. 233

7.1 Demonic strategy connectives. 255
7.2 Angelic strategy connectives. 256
7.3 Kaisar forward-chaining natural deduction proof rules. 262
7.4 SSA algorithm for strategies. 275
7.5 Two rules for Angelic loops. 286

8.1 Scala datatype for ProofPlex strategy IR. 315
8.2 Scala trait for Demonic strategies. 317
8.3 Execution algorithm for strategies. 321
8.4 Robot experiments with Demonic model. 326
8.5 Robot experiments with Angelic Sandbox model. 327
8.6 Robot experiments with Timed Angelic Control model. 328
8.7 Robot experiments with Reach-Avoid model. 329

A.1 CGL proof calculus: propositional rules. 343
A.2 CGL proof calculus: first-order games, first-order arithmetic. 344
A.3 CGL proof calculus: loops. 345
A.4 Operational semantics: β-rules. 347
A.5 Operational semantics: monotonicity rules. 348
A.6 Operational semantics: commuting conversion rules. 349
A.7 Operational semantics: commuting conversion rules (contd.) 350
A.8 Operational semantics: structural rules. 351
A.9 Proof terms for Nim example. 354
A.10 Proof terms for cake-cutting example. 355
A.11 CGL static semantics. 358

B.1 Safe driving envelope. 424

C.1 CdGL proof proof-terms: ODEs. 479

List of Tables

1.1 Comparison of verification approaches. Cells are colored based on how well each approach meets the need of each character, with green being best and orange being worst. 6

3.1 External functions and their intended meaning. 107
3.2 Average speed, monitor failure rates, plant violation rates, for AirSim and human driver in Rectangle, Turns, and Clover for Patrol missions. 124

4.1 Terminologies for players. 144

7.1 Proof metrics for Bellerophon proofs and Kaisar ports. 293

8.1 Average speed, monitor failure rates, plant violation rates, for AirSim and human driver in Rectangle, Turns, and Clover for patrol missions. 330

Chapter 1

Introduction

Cyber-physical systems (CPSs) combining discrete control and continuous physical dynamics are pervasive in modern society: examples include driver assistance in cars, industrial robots, airborne collision avoidance systems, the electrical grid, and medical devices. Many of these systems are safety-critical or even life-critical because they operate in close proximity to humans and in some cases perform life-sustaining functions. Formal verification of these systems is a key tool for attaining the strongest possible guarantees that they meet their safety and correctness objectives. Therefore, as the importance of CPSs in society grows, so does the importance of their formal verification. *Hybrid systems* models combining discrete transitions with continuous differential equations (ODEs), in particular, have succeeded in providing a common formalism for the discrete and continuous aspects of a CPS. Hybrid systems theorem-proving in differential dynamic logic (dL) (Platzer, 2018a, 2008a, 2017a, 2011, 2010b, 2012c) is a notable approach for verification of cyber-physical systems, in particular its strong logical foundations and successful application in a number of case studies (Jeannin et al., 2017; Loos, Platzer, & Nistor, 2011; Mitsch, Ghorbal, Vogelbacher, & Platzer, 2017; Platzer & Quesel, 2008b) using the theorem prover KeYmaera (Platzer & Quesel, 2008a) and its successor, the KeYmaera X (Fulton, Mitsch, Quesel, Völp, & Platzer, 2015) theorem prover. Proofs in dL are compared to other verification approaches in Section 1.2.

While dL has many successful applications to date, there are always new verification challenges. As CPSs have become pervasive in society, their implementation complexity has grown greatly, as shown by publicly available size estimates for well-known CPSs. NASA's Curiosity mission required roughly 3 million lines of code (Holzmann, 2013), a number dwarfed by aircraft, and automotives: well over 6.5 million lines for the Boeing 787 (Wagner & Norris, 2009), and at least tens of millions of lines in modern automotives (Greengard, 2015). Many modern CPS controllers even include machine-learned components whose complexity is measured not in lines of code but in the size of data tables that drive the controller: the ACAS X airborne collision avoidance system is driven by tables containing millions of entries that lead to trillions of combinations (Jeannin et al., 2015).

This thesis builds on the dL tradition and its ModelPlex monitoring approach (Mitsch & Platzer, 2016b) to develop an *end-to-end* verification approach which can cope with the complexity of modern CPS implementations. End-to-end verification is best under-

stood as a grand challenge problem, meaning that its scope is expansive and that diverse technical approaches can contribute to the same broad goal. All end-to-end verification approaches recognize that today's formal models leave out important aspects of reality, and that bridging the gaps between model and reality is crucial to improving the correctness of real systems. Because end-to-end verification is a grand challenge problem, no single thesis can address every relevant research front. This thesis addresses end-to-end verification on three fronts: design and implementing the VeriPhy approach for end-to-end verification through synthesis of sandbox controllers[1], developing logical foundations which support general-case synthesis with rigorous correctness guarantees, and developing a proof language Kaisar which assists with the complex models are proofs needed by VeriPhy. Other research fronts are beyond our scope but are also important. For example, the verification of sensing and actuation is a crucial and philosophically challenging problem. Verification of fundamental system software such as compilers and operating systems also plays a key role in constructing complete, correct systems. To arrive at a precise thesis statement, we first summarize the specific contributions of this thesis to end-to-end verification.

The first major applied contribution of the thesis is VeriPhy: an approach and synthesis tool for end-to-end CPS verification based on dL. Two implementations of VeriPhy will be given: a *classical* implementation of VeriPhy based on dL is given first. The limitations of classical VeriPhy will inspire an investigation of constructive foundations that culminate in a *constructive* implementation of VeriPhy whose strengths and weaknesses complement the classical version.

(Classical) VeriPhy takes a proven-correct dL model as its input and produces a sandbox controller which monitors an untrusted controller for compliance with the model, replacing any potentially-safe decisions with proven-safe fallback decisions. The monitoring formulas are generated by invoking KeYmaera X's correct-by-construction ModelPlex (Mitsch & Platzer, 2016b) monitor synthesis tool. ModelPlex provides a correctness proof for the monitoring formulas, which VeriPhy extends to a proof of correct sandbox control. Fallback controllers generally ensure safety by sacrificing secondary liveness objectives: for example, a car may engage its emergency brake to remain safe, causing it to not reach its destination. VeriPhy's strengths are a high degree of automation, a rigorous argument that the final system implementation is safe, and support for hybrid systems models with non-trivial (e.g., non-linear) differential ODEs. VeriPhy achieves these goals by adopting a structure similar to a verified compiler: synthesis is divided into simple passes, each of which is proven correct with a refinement-like or simulation-like argument.

Classical VeriPhy provides a particularly thorough proof argument with formal proof artifacts throughout each transformation pass. In practice, its limitations include strict fixed modeling and proof formats for controller models, limited precision for arithmetic computations, and dependence on automated proofs which are not guaranteed to succeed. Constructive VeriPhy is motivated by the desired to overcome these limitations while also adding greater functionality. Specifically, classical VeriPhy can only synthesize and guaran-

[1]The VeriPhy monitoring approach exploits the existing ModelPlex (Mitsch & Platzer, 2016b) method for correct-by-construction monitoring of system compliance with dL models, but goes further by extending safe monitoring to safe sandbox control.

tee safety of sandbox controllers. While sandbox control is a crucial paradigm for managing controller implementation complexity, classical VeriPhy has no hope of proving controller *liveness* because sandbox controllers intentionally sacrifice liveness when the alternative is sacrificing safety. The only hope for proving liveness is to allow (optional) support for *whitebox* controllers, meaning controller models with *explicit* control calculations that are amenable to both safety and liveness proof. Because a verified whitebox controller gives an explicit control calculation which can be proved safe, no fallback controller is needed, allowing the system to make progress in every case so long as the verified controller is also proved live. We consider liveness guarantees an important part of the broader end-to-end verification philosophy because the end-to-end philosophy emphasizes comprehensive guarantees and liveness is a crucial element of functional correctness. Moreover, we will find that the same work which enables support for whitebox liveness guarantees will help address classical VeriPhy's limitations which arose even when proving safety guarantees, such as restrictions on proof formats and reliance on incomplete automated proof methods.

Logical foundations are the second major focus of the thesis: new logical foundations are the key enabling factor which will support constructive VeriPhy in addressing the limitations of classical VeriPhy. Because the VeriPhy synthesis algorithm is driven by hybrid system models and their proofs, it is only natural that advances in modeling and proof foundations can enable advances in synthesis. We develop *Constructive Differential Game Logic* (CdGL), the extension of dL to constructive hybrid games. Because there exists a logic dGL (Platzer, 2015a) which extends dL to classical hybrid games, CdGL is also the constructive variant of dGL. In developing CdGL, we show that reasoning principles from dGL extend smoothly to an entirely new constructive semantics once the nuances of constructive arithmetic have been confronted. Our semantic developments for CdGL provide a roadmap for synthesis: an explicit proof term language and its operational semantics provide a clear input language for tools which process proofs, while an operational semantics for strategies of games describes how controllers and monitors should actually be executed. In short, we show that games help support safe, live whitebox controllers while constructivity ensures that *all* CdGL proofs can be translated to code. By supporting games, VeriPhy also inherits existing benefits of hybrid games, including easier modeling of adversarial systems and easier high-level modeling of controllers. CdGL features a refinement calculus, which supports the correctness argument for constructive VeriPhy.

The third focus of the thesis is proof language design and implementation. We introduce Kaisar, a language of structured proofs for CdGL from which constructive VeriPhy can synthesize code. We demonstrate how Kaisar's structured paradigm alleviates usability, maintainability, and scalability challenges present in current-generation proof languages. Lastly, constructive VeriPhy is implemented with Kaisar as its inputs and a low-level executable intermediate language as its output, as of this writing. We evaluate constructive VeriPhy against its classical counterpart, showing that the constructive implementation's strengths include input language flexibility and support for liveness guarantees, but its larger trusted code base and mathematically complex foundations[2] mean its correctness argument is an informal one as of this writing, as opposed to the rigorous proof for classical

[2]Formalization challenges presented by the foundations are discussed in Chapter 5 and Chapter 6.

VeriPhy with extensive formal proof artifacts. Classical VeriPhy also provides a lower-lever target language for synthesis: machine code.

The three research thrusts of the thesis are not isolated from each other: the classical implementation of end-to-end verification and its example applications respectively inspire our new constructive foundations and new proof language design principles. Those principles are put to the test when Kaisar and constructive VeriPhy are implemented and evaluated. More precisely, our experience with classical VeriPhy inspires the remaining research thrusts of the thesis by inspiring us to meet three competing sets of needs. For the narrow purposes of a thesis statement,

- *Verification* means that we establish a high degree of confidence in correctness, backed by formal proof.
- *End-to-end* means that the correctness properties which were verified of the model (the input end) are extended to hold of implementation-level code (the output end).
- *Practical* means that the effort required for modeling and proof, including maintenance effort, is modest compared to the literature, particularly when the approach is scaled to models and implementations that are of applied interest. Regarding practicality of models and their high-level proofs, our focuses include maintainability and readability of model and proof artifacts. Regarding practicality of implementation-level guarantees, our focus is on automating the correctness proofs of implementation-level code given proofs of high-level models.

Given these definitions, we can give a concise thesis statement:

> Constructive Differential Game Logic (CdGL) enables practical, end-to-end verification of cyber-physical systems.

Our definitions of "verification," "end-to-end," and "practical" are not the only possible definitions, as "end-to-end" and "practical" are notoriously vague terms. Our definitions are not arbitrary either, as they reflect the competing needs of different people who participate in development of verified software. To emphasize the competing goals of end-to-end verification, we introduce a cast of characters and show how their needs can be met simultaneously, to a far greater extent than prior work. In showing how the competing needs can be met, we show how the thesis meets the needs of practical, end-to-end verification which are listed above.

The Cast. The contributions of this thesis are united by their common goal of resolving three competing needs: rigorous formal foundations, easy-to-use implementation code, and easy-to-use verification technology. We introduce three characters which represent the respective needs: the Logician, the Engineer, and the Logic-User. Though constructed by the author, the characters are a stylized way to emphasize and resolve real differences of perspective and priority on the spectrum between theorists and practitioners.

The Logician follows in the formalist tradition of David Hilbert: to know something is to have a proof of it. The Logician holds mathematics to the highest standard possible,

and they do so with good reason. CPSs are often life-critical, and if all the Logician's paranoia can prevent defects in CPSs, it has been worthwhile. Through the history of CPS, a variety of safety incidents (MacKenzie, 1994) have occurred, showing that safety is a practical and not merely theoretical concern. Because safety is crucial and proofs by humans are not immune from errors, today's Logician seeks the most rigorous, formal proofs possible. The gold standard is a machine-checkable proof in a formal proof calculus along with strong evidence that the proof calculus and proofchecker are sound. In some cases, it is even feasible to prove one proofchecker sound using another, a process which sometimes resolves bugs in proofchecking tools (Chapter 2). The Logician's commitment to formal proof yields highly trustworthy results, but the human effort required for those results is significant and can increase rapidly with system complexity.

The Engineer places their focus elsewhere, recognizing the potentially high cost of formal verification. The Engineer is the one tasked with designing, building, and delivering a production CPS under time and budget constraints. The Engineer will gladly use formal methods, but only if they can show concrete safety benefits on realistic systems in a short timeframe and with limited specialist training. Techniques that appeal to the Logician might appall the Engineer because they are time-consuming or only guarantee safety of an ideal model. The Engineer would rather fix one bug in the implementation than ten bugs in an ideal model, since fixing a bug in the model is not guaranteed to improve the quality of shipped code.

The Logic-User is oft-forgotten, and sits between the Logician and Engineer on the spectrum from theory to practice. The Logic-User (called the Proof Engineer (Ringer, Palmskog, Sergey, Gligoric, & Tatlock, 2019) or Verification Engineer (Mitsch, Passmore, & Platzer, 2014) by other authors), is the person tasked with employing verification tools *at scale*. Unlike the Logician, the Logic-User does not obsess with the soundness of proof rules, because they trust that the Logician has implemented the verification tool correctly. The Logic-User believes in the value of a verified model, but sympathizes with the Engineer's plea that only a nuanced model could hope to capture the difficulties faced in practice. Verification takes time, but the Logic-User needs to spend verification time wisely: time should not be wasted on verification tasks that could easily be automated away, and no unnecessary barriers to learning the tool should be erected.

As other authors have noted (Ringer et al., 2019), the productivity of the Logic-User takes on growing significance today as the scale of verification efforts increases. In hybrid systems specifically, we additionally note that scalability challenges can set in at surprisingly small scales, with "large" case studies taking a hundreds of lines of proof (Mitsch et al., 2017) and the largest known proofs being on the order of a thousand lines (Jeannin et al., 2017). Thus, while we will speak of scalability as something the Logic-User cares about, the scale of modern hybrid systems proofs is modest when compared to the largest extant formal proofs, let alone largest known codebases. It is an explicit non-goal of this thesis to present models and proofs of larger scale than prior work. We instead seek to diagnose factors which have limited proof scale to date, then propose, implement, and evaluate technologies which can alleviate those underlying factors.

As shown in Table 1.1, no prior approach satisfies all characters to the same extent as our new approach based on CdGL and VeriPhy:

Approach	Logician	Engineer	Logic-User
GPITP	formal	manual effort	labor-intensive
Automata	paper	monitors, controls	error-prone
dL before	paper	sound monitor formula	less error-prone, less labor-intensive
dL after	formal	sound monitor program	less error-prone, less labor-intensive
CdGL	formal	monitors, controls	least error-prone, less labor-intensive

Table 1.1: Comparison of verification approaches. Cells are colored based on how well each approach meets the need of each character, with green being best and orange being worst.

- General-purpose interactive theorem provers (GPITP) such as the HOL family and Coq satisfy the Logician's desire for solid logical foundations because their foundations have been studied extensively, with various degrees of formalization (Barras, 2010; Kumar, Arthan, Myreen, & Owens, 2016). Even so, the Logician may have some skepticism of the code extractors provided in GPITPs, because verification efforts for code extractors (Mullen, Pernsteiner, Wilcox, Tatlock, & Grossman, 2018; Anand et al., 2017; Ioannidis, Kaashoek, & Zeldovich, 2019; Hupel & Nipkow, 2018) typically do not cover all (Hupel, 2019a) features needed in practice. The Logic-User's concerns are more extensive: in a general-purpose logic, it may be more difficult to write correct models, and correctness proofs certainly require more effort when there is no specialized support for CPS. The Engineer typically does not interact directly with a theorem prover, but would still be dissatisfied if the generated code cannot be easily integrated with the codebase of an existing implementation of the system.
- In hybrid automata-based approaches, safety checking and code synthesis (Toom, Naks, Pantel, Gandriau, & Wati, 2008; Henzinger, Horowitz, & Majumdar, 1999; Kloetzer & Belta, 2008; Nilsson et al., 2016; Taly & Tiwari, 2010; Tomlin, Lygeros, & Sastry, 2000; Althoff & Dolan, 2014) typically do not have formal soundness proofs. The lack of formal proofs is arguably more significant for automata-based tools than theorem provers for two reasons. First, the automata-based verification algorithm may be harder to implement soundly because it must bridge a wide conceptual gap between hybrid system semantics and low-level operations on the data structures that represent system state. Second, code generation for automata-based tools may be harder to implement soundly because the code generated by theorem provers follows its formal model more closely by comparison: the formal model is already a programmatic model in logics such as dL. If a particular Logic-User is more familiar with program verification, there may also be a learning curve for automata-based approaches. The converse holds for a Logic-User with a background in automata-based tools.
- Of the available choices, dL assists the Logic-User with a program-like syntax that helps avoid modeling mistakes and with domain-specific proof rules that are less laborious than proofs embedded in a GPITP. However, the Engineer would find that dL's state-of-the-art synthesis tools for dL (Mitsch & Platzer, 2016b) only synthesize monitor formulas as opposed to executable controllers, which also frustrates the Logician

because a comprehensive correctness argument requires a concrete controller. Support for non-linear differential equations in those tools is also experimental. Before this thesis, the Logician might also complain that dL's foundations were developed and proved only on paper and not with a machine-checked proof. As a contribution of the thesis, dL's soundness theorem is now machine-checkable (Chapter 2).

- VeriPhy synthesizes executable controllers as desired by the Engineer, while maintaining strong formal guarantees as desired by the Logician. Upon introducing VeriPhy in Chapter 3, we will see that the choice of foundations and proof language play a key role in whether VeriPhy can scale to the complex models needed by the Engineer without compromising the proof simplicity desired by the Logic-User. We develop CdGL and show that by adopting it as the basis of a second VeriPhy implementation (Chapter 8) with a new proof language (Chapter 7), we provide the flexibility that the Engineer needs for practical models while supporting high-level proofs for the Logic-User and supporting extraction of *whitebox* controllers from proven-live hybrid games for the first time ever, as desired by the Engineer.

In every approach, model validation is also an area of concern: the Logician knows that a real system is only safe if the model accurately describes reality. Model validation is also important to the Logic-User and Engineer because the Logic-User does not want to spend time proving an inaccurate model and the Engineer wants the system implementation to benefit from the formal efforts. The VeriPhy approach serves to both verify and validate models, as violations of physical modeling assumptions are detected at runtime.

The goal of the thesis is to meet the needs of the Logician, Engineer, and Logic-User at once. This is our contribution to "Practical End-to-End Verification of CPSs".

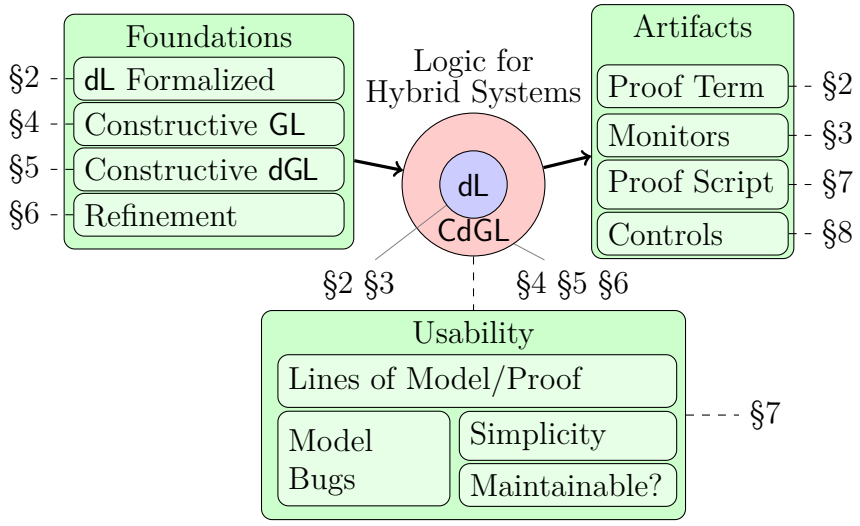


Figure 1.1: Goals of the thesis.

1.1 Outline and Contributions

The relationships between the chapters of the thesis are laid out in Fig. 1.1. Each chapter develops new formal foundations for the Logician, produces software artifacts for the Engineer, and/or addresses usability needs of the Logic-User. Note that the three parts of the thesis do not correspond to our three characters but to a three-act structure. Part I develops the VeriPhy end-to-end approach for classical hybrid systems using existing foundations and proof languages. Our experience with VeriPhy reveals the limitations of synthesis built on a classical dL foundation. Part II proposes a new foundational solution: constructive hybrid games with refinement (CdGL). Part III shows that the new foundations resolve the limitations of classical VeriPhy: we build a new proof language Kaisar and new VeriPhy implementation for CdGL, whose architecture is crucially built on insights about game refinement and whose informal correctness crucially appeals to formal theorems about refinement. Throughout the thesis, the work is evaluated with a series of driving (or wheeled robotics) case studies. The case studies cover safety, liveness, and reach-avoid correctness for straight and curved driving in 1 and 2 dimensions. The code synthesized from the driving models is evaluated on a Raspberry Pi-based robot and in the simulator AirSim.

Part I begins by formalizing dL in Isabelle/HOL (Chapter 2), which satisfies the Logician by increasing our trust in the dL foundations. When possible, the Isabelle/HOL formalization follows the KeYmaera X proof calculus for dL, increasing confidence in KeYmaera X as well. Chapter 3 introduces the design and classical implementation of the monitor synthesis tool VeriPhy (Bohrer, Tan, Mitsch, Myreen, & Platzer, 2018) which connects a dL model to implementation code while transferring formal safety guarantees to the real world. Part I concludes by discussing the limitations of classical VeriPhy for dL, including a brittle implementation, inability to synthesize whitebox controllers from liveness proofs, and model complexity which increases quickly with control scheme complexity.

The constructive game logic of Part II provides a foundational solution to the limitations of classical VeriPhy. Game proofs combine control and monitor reasoning in a common artifact. Constructivity gives a Curry-Howard isomorphism for constructive games which demonstrates that proofs correspond to monitoring and control code in *every* case. Proofs about our player are interpreted as executable strategies which specify the moves (or control decisions) taken by our player, while proofs about our opponent are interpreted as executable strategies which make no moves but check (monitor) whether the opponent's moves follow the rules of the game. Thus, game logic and constructivity combine to provide a basis for monitor and control synthesis which admits a robust, general-case implementation. Game models also stay simple even as control grows complex, which reduces the risk of safety-critical modeling mistakes by the Logic-User. Chapter 6 develops a refinement logic for constructive hybrid games. Refinement is used to show a reduction from proven hybrid games to hybrid systems, which in theory reduces end-to-end verification of games to end-to-end verification of systems.

Part III implements proof and synthesis tools for constructive games, thus reaping the practical benefits of our new foundations. Along the way, we see that refinement is a convenient organizing principle for correctness proofs about game verification and synthesis. As the most applied part of the thesis, Part III has the strongest focus on the Logic-User.

Chapter 7 develops Kaisar, a language of structured proofs for CdGL, where proofs are game models annotated with constructive first-order reasoning. The Kaisar design is informed by limitations encountered while proving case studies for use with classical VeriPhy and the solutions to those limitations are informed by the author's work on hybrid dL (Bohrer & Platzer, 2018). Kaisar is integrated with a new implementation of VeriPhy to provide a truly "end-to-end" approach. Chapter 8 provides the completely new implementation of *constructive* VeriPhy, which synthesizes controllers and monitors from Kaisar proofs. Kaisar is evaluated by rewriting the VeriPhy case studies in Kaisar and comparing metrics including proof script length and number of lines changed when adding new model features, the latter being a proxy for maintainability. Constructive VeriPhy is evaluated by synthesizing code from the Kaisar proofs, which is tested in an interpreter, both on hardware and in simulation. These evaluations demonstrate how constructive VeriPhy supports the needs of Engineer and Logic-User. While Constructive VeriPhy emphasizes the Logic-User and Engineer over the Logician, the deep relationship of constructive VeriPhy with game logic and game refinement is discussed to suggest how the Logician's high standards of rigor from classical VeriPhy might be achieved for constructive VeriPhy as well.

The following works completed during the PhD are not discussed at length in this thesis, in order to keep the focus on end-to-end verification. The author's works on definite description in dL (Bohrer, Fernández, & Platzer, 2019) and hybrid logic (Bohrer & Platzer, 2018) respectively serve to provide a unifying framework for the several term languages of the thesis and to provide an understanding of labeled reasoning in Kaisar (Chapter 7).

1.2 Related Work

This section discusses related works that are broadly related to the thesis as a whole: works on verification of cyber-physical systems, synthesis, and end-to-end correctness. Works which are relevant primarily to one chapter are discussed in the corresponding chapter. Verification and synthesis are both broad topics, but we show that our approach has unique strengths among end-to-end CPS verification approaches.

1.2.1 Verification of CPS

Verification of CPSs is a broad and actively studied field. While CPS verification and hybrid systems verification are not synonymous, hybrid systems are the heart of CPS verification because they can represent the interacting discrete and continuous dynamics that distinguish CPSs from other computing systems. Discrete models can be applied to CPSs, but discrete formalisms oversimplify crucial continuous dynamics and common discrete formalisms such as state machines (Rabin & Scott, 1959), process calculi (Hoare, 1978), and dynamic logic programs (Harel, Tiuryn, & Kozen, 2000) are strict fragments of more expressive hybrid formalisms. Within the field of hybrid systems, verification approaches vary in several ways: formalisms range from automata to programming languages, verification techniques range from model checking to theorem proving, and the theorem provers range from general-purpose to special-purpose. We survey the range of hybrid systems ver-

ification technologies and discuss why this thesis uses theorem-proving in special-purpose dL-style logics with programmatic models of hybrid systems. Verification and synthesis are often studied separately, but our approach fundamentally exploits proofs to drive synthesis. We now discuss the above classes of works in greater detail.

1.2.1.1 Formalisms for Hybrid Systems

There are several prevalent representations for hybrid systems which, even when equivalent, lend themselves to different modeling and verification approaches. Because hybrid systems reachability is only decidable for narrow classes such as initialized rectangular automata (Henzinger, Kopke, Puri, & Varaiya, 1998), all major hybrid systems verification approaches must confront the limits of decidability, e.g., by adopting approximations or by using human insight to drive verification.

This thesis models hybrid systems as *hybrid programs* (Platzer, 2018a, 2008a, 2017a, 2012b), the modeling language of dL, where hybrid systems are written in a programming language that includes differential equation evolutions as program statements. Other authors have used closely-related programmatic models to investigate hybrid systems information flow (Prabhakar & Köpf, 2013) and to analyze invariants of differential equations by reducing differential equations to loops with infinitesimal timesteps (Suenaga & Hasuo, 2011). Programmatic models are well-suited for syntactic, deductive proof, and have the advantage that their use can be taught by analogy to other programming languages. Programs naturally express a broad class of hybrid systems including nonlinear differential equations and nonlinear discrete assignments and guards.

Hybrid automata (Henzinger, 1996) are a widely-used representations for hybrid systems. Hybrid automata are well-suited for model checking and, as with programs, can be used to express non-linear differential equations, assignments, and guards. Model checkers typically target a specific fragment of hybrid automata or specific fragment of safety properties, which we list when we discuss each model checker. Automata are well-known in theoretical computer science, but their use may pose a learning curve for lay practitioners.

Process calculi have been developed for hybrid systems and excel at modeling concurrent or communicating CPSs with events. They include Hybrid CSP (Zhou, Wang, & Ravn, 1995; Liu et al., 2010), HyPA (Cuijpers & Reniers, 2005), and Hybrid χ (Schiffelers, van Beek, Man, Reniers, & Rooda, 2003), which have been used for both model-checking and theorem-proving. As with automata, process calculi may require a learning curve for practitioners, but their syntax is more programmatic and even shares some operators with hybrid programs.

A notable practical approach for event-based models is Event-B (Abrial, 2010), which makes crucial use of refinement to reduce verification of more complex event-based models to simpler event-based models. Compared to our definition of refinement (Chapter 6), theirs includes a more aggressive treatment of ghost variables: relational arguments between ghost and non-ghost state allow refinement properties to hold between systems with differing sets of state variables. Our definition of refinement fixes the dimension of the state, which leads to a simpler definition but means that extra dummy assignments are sometimes necessary when comparing programs whose state spaces differ. Extensions of Event-B have been

proposed for hybrid systems (Banach, Butler, Qin, Verma, & Zhu, 2015; Su, Abrial, & Zhu, 2014; Banach, 2013; Dupont, Ameur, Pantel, & Singh, 2018) but not hybrid games. Among proposed hybrid systems extensions, those which have been implemented (Dupont et al., 2018) require non-trivial setup effort by the user, to the best of our knowledge. While it does not have hybrid games today, Event-B remains a viable platform for development of new refinement reasoning technology for CPS in the long term.

Regardless, the logical and semantic foundations of game refinement remain of independent interest, as do implementation approaches other than Event-B. In this thesis, we first study foundations that should be of interest regardless of tool choice (Chapter 6), then design and implement the Kaisar language (Chapter 7) which can verify hybrid games by refining them to systems which perform the game's winning strategy.

Programmatic models (Harel et al., 2000), automata (Rabin & Scott, 1959), and process calculi (Hoare, 1978) are all commonly used for discrete programs in addition to hybrid systems. Verification of CPS can be pursued using discrete models (Lecomte, Déharbe, Prun, & Mottin, 2020; Akella, Tang, & McMillin, 2010; Cousot et al., 2005), but the drawback is that discrete models are less faithful to a real CPS and thus verification results do not translate as easily to real systems: a theorem is only as good as the model. One approach to bridge the modeling gap between discrete models and hybrid systems models is to generate a conservative discrete approximation of a hybrid system (Alur, Henzinger, Lafferriere, & Pappas, 2000) where every safety theorem of the discrete system is also a safety property of the hybrid system that it approximates. However, discrete approximations of hybrid systems are often far too conservative in practice, i.e., there are many safe systems whose discrete approximations cannot be shown to be safe. Additionally, the size of the discrete approximation grows quickly with the complexity of the hybrid system, presenting scalability challenges for nontrivial hybrid system models.

Interoperation between different modeling languages and between different automata-based tools are areas of active research. The modeling language IPL (Ruchkin, Sunshine, Iraci, Schmerl, & Garlan, 2018) was developed to provide interoperation between different modeling languages which address different levels of system abstraction or different aspects of system correctness. Works which address interoperation between automata-based tools with differing modeling notations and semantics include HyST (Bak, Bogomolov, & Johnson, 2015) and HSIF (Pinto, Carloni, Passerone, & Sangiovanni-Vincentelli, 2006).

1.2.1.2 Model-Checking by Reachability Analysis

Model-checking for hybrid systems by reachability analysis is thoroughly studied. Well-known model-checkers include SpaceEx (Frehse et al., 2011), Flow* (X. Chen, Ábrahám, & Sankaranarayanan, 2013), and CORA (Althoff, 2015), as well as C2E2 (Duggirala, Mitra, Viswanathan, & Potok, 2015).

These tools adopt different underlying data structures and algorithms which result in support for different classes of hybrid systems, different verification properties, and different levels of scalability.

SpaceEx supports piecewise affine automata (Guernic, 2009), a broader class than

the earlier PHAVer[3] (Frehse, 2005). SpaceEx emphasizes scalability for this relatively simple class of hybrid systems; it achieves scalability to ≈100 continuous variables by combining polyhedra and support functions in its representation of (sets of) system states. SpaceEx pursues unbounded-time safety guarantees using invariant arguments. Because SpaceEx conservatively approximates the reachable states of a system, it may reject a set which is invariant under the true dynamics of a hybrid system but not invariant under the approximate semantics. Stable systems are a potential application domain for approximate invariant analyses: if all system states converge toward an equilibrium point over time, it is possible for an invariant check to succeed even under a conservative semantics.

C2E2, Flow*, and CORA all support nonlinear differential equations. C2E2 performs bounded analysis of nonlinear systems by simulating them repeatedly with progressively tighter error bounds until a property is proved or falsified, a process which is not guaranteed to terminate for safety properties which hold with a low degree of robustness, e.g., for exact safety bounds as opposed to conservative safety bounds. Flow* performs bounded-time safety analysis using Taylor models which conservatively approximate solutions of ODEs with polynomial upper and lower bounds. CORA is a MATLAB toolbox for both set-based analysis and point-based simulation with an emphasis on flexibility regarding choice of set representation and reachability analysis algorithm. CORA supports nonlinear ODEs by conservatively abstracting them to polynomial difference inclusions (Althoff, 2013). In each of C2E2, Flow*, and CORA, the emphasis on bounded-time guarantees helps offset the mathematical complexity of non-linear ODEs.

Compared to theorem-provers, the implementation of reachability checkers typically entails the design and implementation of complex data structures and algorithms for the representation of sets of system states and system evolution over time. Those data structures and algorithms are an important contributor to the trusted code base of reachability checkers, a contributor which is not present in the trusted computing base of a theorem-prover. For example, the trusted core of SpaceEx has been estimated[4] (Fulton et al., 2015) at 100 K lines of code.

Compared to theorem provers, the existing model checkers consider narrower classes of systems or narrower classes of system properties. For example, SpaceEx only considers piecewise affine systems and the only safety regions it supports are convex sets that can be represented with polyhedra or support functions. C2E2 and Flow* only consider bounded-time safety, with C2E2 only supporting polyhedral sets of initial system states.

As with theorem-proving, hybrid systems model-checking is canonically used today for offline verification of safety properties. Model-checking has been applied to many application domains, including ground robotics (X. Chen et al., 2015), which is also the area of focus for the case studies in this thesis. While offline safety verification is a canonical use of model-checkers, other uses have also received attention:

- Runtime reachability analysis has been performed in real time on cars (Althoff &

[3]PHAVer supported *linear hybrid automata* (Henzinger, 1996), where differential inequalities have piecewise-*constant* bounds. Linear hybrid automata should not be confused with automata whose differential equations are linear, a class which is far more expressive than linear hybrid automata and which constitutes a significant subset of affine automata.

[4]The estimate is in the conference presentation slides, not the paper.

Dolan, 2014; Lin, Chen, Khurana, & Dolan, 2020), which enables the use of fallback controllers for safety (Pereira & Althoff, 2015) (see discussion in Section 1.2.2.2).

- Liveness analysis (Cimatti, Griggio, Mover, & Tonetta, 2014) is available in some solvers (Cimatti, Griggio, Mover, & Tonetta, 2015), albeit fewer than safety.

To better support the use of reachability analysis algorithms in trusted contexts, a set-based reachability checker for ODEs has been verified (Immler, 2015) using Isabelle/HOL's differential equations library (Immler & Traut, 2016). A verified integrator for ODEs based on computable reals has also been developed in Coq (Makarov & Spitters, 2013). While ODE reachability is a fundamental building block for hybrid systems reachability, the verified checker does not support hybrid systems.

1.2.1.3 Theorem-Provers

Interactive theorem-proving is one of the major approaches to hybrid systems verification, and is characterized by high expressiveness, which can come at the cost of increased verification effort. Hybrid systems theorem proving has been pursued both in existing general-purpose interactive theorem provers (GPITPs) (Rouhling, 2018; S. Foster, 2019; Huerta y Munive & Struth, 2019) and in domain-specific provers (Platzer & Quesel, 2008a; Fulton et al., 2015; Liu et al., 2010) for domain-specific logics (Platzer, 2008a). GPITPs excel at flexibility because they provide access to large libraries of general-purpose mathematics and are capable of modeling diverse system dynamics beyond hybrid systems. Because GPITP modeling is flexible, a given CPS is sometimes modeled directly rather than using a specific representation of hybrid systems (Rizaldi, Immler, Schürmann, & Althoff, 2018; Rizaldi et al., 2017; Chan, Ricketts, Lerner, & Malecha, 2016). Mathematical libraries have also been built to address background topics relevant to CPS, including geometry (Affeldt & Cohen, 2017) and dynamical systems (Cohen & Rouhling, 2017). The downside of using a GPITP is that specialized support for hybrid systems is not provided out-of-the-box, which may result in less effective use of automation or a steeper learning curve. Efforts have been made to provide hybrid system proofs as a library within GPITPs (S. Foster, 2019; Huerta y Munive & Struth, 2019), but automation for differential equations is highly specialized, which can cause automation in libraries to lag behind specialized tools. For example, there is a notable line of work (S. Foster, 2019; Huerta y Munive & Struth, 2019; Hickman, Laursen, & Foster, 2021) which provides a hybrid systems library in Isabelle/HOL based on Unifying Theories of Programming (UTP) (Hoare & He, 1998) and refinement calculi. As this line of work improves its automated support for integration of ODE solvers (Hickman et al., 2021) and its support for crucial invariant-based reasoning methods (Huerta y Munive & Struth, 2019), important gaps in automation relative to domain-specific approaches still remain, such as the lack of access to real arithmetic decision procedures (e.g. (G. E. Collins & Hong, 1991)) and lack of advanced ODE invariant search procedures (e.g. (Sogokon, Mitsch, Tan, Cordwell, & Platzer, 2019)). On the other hand, a notable strength of this approach is its easy access to the wide variety of mathematics that has been formalized in Isabelle/HOL. Even the avoidance of real arithmetic decision procedures has its advantages, because it means that the decision procedures need not be added to the trusted computing base of the verification effort, with the tradeoff

being that absence of such procedures significantly inhibits scalability of verification due to the extensive effort required for interactive verification of arithmetic properties.

Domain-specific hybrid systems provers include KeYmaera (Platzer & Quesel, 2008a) and its successor KeYmaera X (Fulton et al., 2015) as well as the Hybrid Hoare Prover (Liu et al., 2010). KeYmaera X supports dL and its extension dGL (Platzer, 2015a), while Hybrid Hoare Prover supports a Hybrid Hoare Logic for Hybrid CSP. Specialized provers typically provide superior automation and a relatively gentle learning curve, but cannot model systems outside their chosen domain at all.

Not to be confused with dGL, the original KeYmaera supported its earlier relative called dDG$\mathcal{L}$ (Quesel, 2013, Ch. 4). A key technical limitation of dDG$\mathcal{L}$ compared to GL, dGL, and CdGL is that it employs an *advance-notice* semantics for loops, meaning that the player who controls a loop must decide the number of loop repetitions before the loop starts, a restriction which is significant because it precludes the commonly-desired (Chapter 5) feature of allowing loop repetition logic to *react* to the moves the opponent has made throughout earlier loop repetitions. The standard loop semantics used by GL, dGL, and CdGL can be used to implement advance-notice loops but not vice-versa.

The logic dDG$\mathcal{L}$ is also notable because its original motivations fit within the philosophy of end-to-end verification: dDG$\mathcal{L}$ was used to state and prove similarity properties between hybrid systems (Quesel, 2013, Ch. 5) under the motivation that a high-level and low-level hybrid system model of the same CPS might have similar but not identical behavior. If a high-level model is proved safe and a similarity proof allows its safety guarantee to transfer to a lower-level model that is easier to implement, then progress has been made toward end-to-end verification.

While we and dDG$\mathcal{L}$ share high-level motivations, dDG$\mathcal{L}$ only shows similarity properties between hybrid systems and does not address lower levels of abstraction such as compiled code that uses approximate arithmetic, thus prior work on dDG$\mathcal{L}$ does not present guarantees for executable code as we do. Their notion of similarity allows scaling or translating time (Quesel, 2013, Ch. 5) because low-level and high-level models may differ in their treatment of time. Our refinement calculus (Chapter 6) addresses a more general refinement property where one system refines another if all safety properties of the former are safety properties of the latter[5]. However, their work would provide a useful roadmap if we were to try to prove time translation and scaling properties in our system.

Differential Dynamic Logics This thesis continues a long line of work using differential dynamic logic (dL) (Platzer, 2008a, 2012c, 2017a), its relatives, and its implementations in the KeYmaera (Platzer & Quesel, 2008a) and KeYmaera X (Fulton et al., 2015) theorem provers. We now discuss that work in greater detail. Verification in dL is symbolic and expressive, excelling at general-case correctness theorems without reliance on conservative numeric analysis. Assignments, guards, and differential equations can all be non-linear. Hybrid systems can have symbolic parameters: for example, an abstract driving model can be proved safe independent of any particular car's acceleration and braking rates. Safety

[5]Our system can prove refinements for both games and systems, but the description given here is phrased for systems because the case for systems requires less explanation.

guarantees are typically unbounded-time. The range of hybrid systems supported by dL and KeYmaera X is well outside decidable fragments (Henzinger et al., 1998). Rather than adopt conservative approximations, dL overcomes undecidability through manual proof interactions from the proof author, which are often necessary in nontrivial case studies across many application domains (Platzer, 2016). While user interaction is typically required, KeYmaera X does provide significant proof automation, allowing the user to focus their efforts on identifying core insights such as safety invariants. For example, invariants of differential equations can often be checked automatically using an algorithm which is in theory[6] able to decide semianalytic invariants of semianalytic differential equations (Platzer & Tan, 2020). The Bellerophon (Fulton, Mitsch, Bohrer, & Platzer, 2017) proof script language provides a combinator-style notation for expressing dL proofs in KeYmaera X as combinations of (often high-level, automated) proof steps. A benefit of deductive proof relative to approximation methods is its higher degree of completeness: there are fewer true properties that have no proof. Specifically, there are relative completeness results for dL (Platzer, 2008a, 2012b, 2017a) which reduce completeness of dL to completeness of fully discrete or fully continuous logics. The high-level takeaway of the relative completeness theorems is that dL is "as complete as possible".

The examples and case studies presented in this thesis focus on driving. Ours (Section 3.7.3) are not the first dL driving case studies: prior work (Mitsch, Ghorbal, & Platzer, 2013; Mitsch et al., 2017) has studied both 2D safety in the presence of sensor uncertainty and actuator disturbance and 1D liveness. This thesis includes a case study which we use to assess VeriPhy's ability to verify realistic systems. Our case study generalizes dL proofs of 1D straight-line motion (with direct velocity control) (Bohrer et al., 2018), of 2D obstacle avoidance, and 1D liveness (Mitsch et al., 2013, 2017). A separate effort proved liveness with dL rules and assumed perfect sensing and constant speed, but did not result in a machine-checkable proof (Martin, Ghorbal, Goubault, & Putot, 2017). Both their controllers and ours are closely related to the classic Dynamic-Window (Fox, Burgard, & Thrun, 1997) control algorithm. Our driving case study improves upon prior efforts by providing 2D liveness guarantees and an explicit model of waypoint-following. Moreover, we have used our driving model to evaluate our end-to-end verification tool VeriPhy and the evaluation process informs the model design: we know with confidence that our model is flexible enough to capture realistic system behaviors, because the model has actually been validated against a realistic simulation at runtime.

1.2.2 Synthesis

Our VeriPhy approach for end-to-end verification relies fundamentally on synthesizing correct controllers and monitors from a hybrid system model that has been proved correct in dL. We discuss the broader context of synthesis in CPS: synthesis can be applied to high-level planning and control tasks, synthesis can be used for online verification via monitors, and synthesized code can contribute to end-to-end verification approaches, including

[6]The implementation does not allow arbitrary semianalytic invariants because the only terms it allows are those of KeYmaera X.

ours. However, the approaches vary greatly in which artifacts are synthesized and what correctness properties they guarantee. In particular, (classical) VeriPhy is distinguished by its formal implementation-level guarantees about physical safety of the CPS at runtime.

On the flip side, synthesis approaches vary widely in their input material: it is more difficult to synthesize code when only provided a model rather than provided both a model and proof. VeriPhy assumes availability of both a model and proof, which makes the synthesis task easier. The synthesis problem is also called *code extraction* in the case that a proof is provided.

1.2.2.1 Offline Synthesis for Planning and Control

The design and implementation of CPSs is typically divided into several levels of abstraction: high-level plans are developed first, then controllers ensure that the plans are followed. Synthesis can be applied at both the planning and control layers and the correctness of each layer is important. Equally important is the observation that hybrid system models support a natural decoupling between planning and control (Nerode & Kohn, 1993): our case study (Section 3.7.3) will assume a planner has been given and will prove control correctness with respect to a planner. Our work could then be combined with any approach that verifies correct planning.

We discuss works which synthesize plans or controllers, but do not pursue end-to-end verification in our sense. Specifically: *i)* they focus on correctness of simple concrete control laws rather than an architecture which can ensure correctness in the presence of complex untrusted codebases, *ii)* they synthesize plans or controllers which are safe under the assumption that real physics perfectly match a model, rather than enforcing compliance at runtime, and *iii)* they do not extend their guarantees to low-level code and its interactions with the physical world. While the following works are not end-to-end, they could potentially serve as components of an end-to-end verified system: a concrete controller synthesized with the following approaches could be used as an untrusted controller within an end-to-end approach based on sandbox control. Synthesis for planning and control have been pursued in the following ways:

- Hybrid systems models have been used in the synthesis of high-level robot motion plans (Bhatia, Kavraki, & Vardi, 2010; Fainekos, Girard, Kress-Gazit, & Pappas, 2009) to ensure that the synthesized plan is physically feasible according to the model. The cited works synthesize plans to satisfy a correctness specification given in temporal logic.
- Controllers have been synthesized: *i)* for rectangular hybrid games (Henzinger et al., 1999), *ii)* from temporal logic specifications for linear systems (Kloetzer & Belta, 2008), *iii)* for adaptive cruise control (Nilsson et al., 2016), tested in simulation and on hardware *iv)* from safety proofs (Taly & Tiwari, 2010) for switched systems using templates, and *v)* by generating hybrid game specifications (Tomlin et al., 2000).
- The tools LTLMoP (Finucane, Jing, & Kress-Gazit, 2010) and TuLiP (Filippidis, Dathathri, Livingston, Ozay, & Murray, 2016) can synthesize robot controls that satisfy a high-level temporal logic specification. Their strength is that they provide

easy-to-learn user interfaces for modeling and verification. Their weaknesses are that their models discretize space and time, and there is not a chain of formal artifacts for each step of synthesis. Thus, even though their contributions to a broader agenda of end-to-end verification are noteworthy, they do not provide end-to-end *guarantees* in the narrower sense which we seek.

Of the above techniques, only two involve hybrid games. In the former (Henzinger et al., 1999), the highly restrictive class of rectangular hybrid games is considered. The latter (Tomlin et al., 2000) uses games as an *intermediate* language, not its *source*. Many automatic techniques are restricted to simplistic ODEs for good reason: only simple classes (Shakernia, Pappas, & Sastry, 2001; Shakernia, Sastry, & Pappas, 2000) have decidability guarantees, so synthesizers for more complex classes are usually incomplete.

Our end-to-end approach (Chapter 3) varies in several key ways. We do not assume that physical reality perfectly matches a model because it never does. Instead, we monitor the real world's compliance with the model so that fallback controllers can be used as a best-effort to restore compliance and so that we can detect model violations. Detecting model violations is an important ingredient in end-to-end verification because it improves model realism in the long term: assuming an iterative development process, models can often be revised to eliminate violations which are present in an initial version. Lastly, we provide end-to-end formal proof artifacts which not only show safe execution of low-level code but show that the physical system under its control satisfies physical safety properties, e.g., collision avoidance.

Our approach ensures safety by sacrificing liveness when untrusted code proposes unsafe actions or when the physical world does not meet assumptions. Safety takes priority over liveness because controller liveness violations can rarely cause harm to humans, but liveness is still a fundamental aspect of functional correctness. In Chapter 8, we address synthesis of controllers from liveness proofs. Liveness guarantees ensure that system objectives are actually achieved, such as a car reaching its destination.

1.2.2.2 Online Verification

The related works discussed in Section 1.2.2.1 are offline: i.e., they are used before-the-fact to ascertain correctness. In contrast, *online verification* (or runtime verification) uses runtime monitor checks to enforce correctness. Online verification excels both at providing safety for control systems which are too complex to model and prove in their entirety and at reacting to a physical world that is only known in full at runtime, where sensor data are available. The value of runtime validation cannot be overstated because system modeling would otherwise present a Gordian knot: without real-world data, it is impossible to truly know whether a proposed model matches reality. By simply checking observed data against a model, runtime verification can cut that Gordian knot.

While the advantages of online verification are substantial, so are the limitations and subtleties. Even online verification approaches can only ensure safety if they incorporate predictions about future physical behavior: detecting that our car has crashed will not make us safe, rather we must check whether braking is necessary now to avoid future crashes. When online approaches incorporate untrusted controllers, they must respond in

a provably safe way when untrusted controllers fail to propose safe control actions, which often violates secondary liveness objectives. More broadly, liveness can never be guaranteed by observing runtime behavior, because liveness is a prediction about future behaviors that have not yet been observed.

Our work builds on a robust tradition of online verification, which we survey below. The key ingredients of our end-to-end safety guarantees are correct construction of monitoring conditions, formal correctness proofs for controllers which employ monitors, and transfer of correctness guarantees from high-level to low-level controller implementations. None of the prior works listed below feature all of these necessary ingredients which enable a formal safety theorem for low-level code and its interaction with a physical system.

- The basis of online verification is the SIMPLEX (Seto, Krogh, Sha, & Chutinan, 1998) method, which uses a trusted monitor to decide between an untrusted controller and trusted fallback. SIMPLEX is most helpful when untrusted controllers are complex, but monitors and fallback controllers are simple. The key insight of SIMPLEX is that the controller need not be trusted because safety depends only on the control decision which it outputs, and the output is only executed if the monitor approves it, else the trusted fallback controller is executed. The SIMPLEX method explains at a high level why controllers based on monitoring are correct, but does not come with a formal proof.
- ModelPlex (Mitsch & Platzer, 2016b, 2014) is a feature of the KeYmaera X prover which synthesizes dL monitor formulas from proven-safe dL models of controllers and physics (plant models). ModelPlex proves that the formula it produces is a correct monitor in the sense that states which satisfy the formula correspond to state transitions permitted by models of respective control or plant models. However, ModelPlex only returns a formula, not a controller, and does not consider how the formula should be executed; VeriPhy picks up where ModelPlex leaves off in order to build verified controllers from correct monitors. In short, ModelPlex provides the knife with which VeriPhy cuts the Gordian knot.

  Our experience with ModelPlex has highlighted the importance of exploiting proof content such as invariants during synthesis. While ModelPlex's support for invariants (Mitsch & Platzer, 2018) is not a contribution of this thesis, our logic CdGL provides a rigorous foundation for computations over proofs, ensuring robust handling of proof invariants in constructive VeriPhy (Chapter 8).
- Our VeriPhy synthesis tool (Chapter 3, Chapter 8) builds on SIMPLEX and ModelPlex. Unlike SIMPLEX, ModelPlex gives correct-by-construction monitor conditions, but ModelPlex only provides a formula over real numbers which provably captures the transitions of a program and does not show how the monitor should be incorporated into a controller or executed. Classical VeriPhy uses ModelPlex monitor conditions in its correct controllers and proves that the controllers stay correct as they are compiled and executed, while constructive VeriPhy generates its own monitors. Classical VeriPhy uses the verified compiler CakeML (Kumar, Myreen, Norrish, & Owens, 2014) for correctness of the final compilation step. As of this writing, constructive VeriPhy uses an unverified interpreter for the final step.

- Both ModelPlex and VeriPhy fall under the paradigm of *predictive runtime verification*: they enforce safety invariants which are strong enough to make safe control possible indefinitely so long as model compliance continues. A number of other predictive runtime verification approaches have been developed for discrete software (Ehlers & Finkbeiner, 2011; Meredith & Rosu, 2010) and CPSs (Bartocci et al., 2012; Pinisetty et al., 2017; K. Yu, Chen, & Dong, 2014; Babaee, Gurfinkel, & Fischmeister, 2018) as well, but they do not come with foundational end-to-end system safety proofs as classical VeriPhy does.
- High-Assurance SPIRAL (HA-SPIRAL) (Franchetti et al., 2017) is a pragmatic compilation toolchain for ModelPlex-synthesized monitors for dL a là SPIRAL (Püschel et al., 2005), but struggles to provide formal end-to-end guarantees. It is discussed at greater length in Section 1.2.2.3.
- Runtime reachability analysis has been used for car control (Althoff & Dolan, 2014; Lin et al., 2020), but relies on correctness both of the plant model and of the analysis implementation. These assumptions present a gap which stands in the way of an end-to-end guarantee because models of driving and implementations of reachability analysis tools are difficult to get right.
- Separately, the study of *conformance* investigates when models at different levels of abstraction are faithful to one another, so that guarantees at one abstraction level transfer to the other. A survey of the literature is available (Roehm, Oehlerking, Woehrle, & Althoff, 2019).
- MOP (F. Chen & Rosu, 2007) is a runtime verification framework which allows correctness contracts to be annotated on (object-oriented) programs and then automatically generates runtime monitors for those specifications using aspect-oriented techniques. Its implementation in ROS is called ROSRV (J. Huang et al., 2014). Because ROS emphasizes message-passing communication in robotic systems, ROSRV emphasizes monitoring of correctness contracts for system events, including monitoring of secure access control policies. ROSRV's evaluation (J. Huang et al., 2014) demonstrated its usefulness on a commercial wheeled robotics platform. While the correctness contracts provided by MOP can help eliminate implementation bugs, the contracts are an arbitrary specification provided by the developer and have no particular relationship to any dynamical model nor any particular relationship to a given safety guarantee for system control. Thus, MOP contracts do not directly amount to enforcement of physical safety.

In conclusion, while classical VeriPhy's correctness argument is crucially structured around a sandbox controller which is built using a runtime verification approach, existing runtime verification techniques would not on their own prove physical safety of the sandbox controller. Rather, past work in runtime verification provides the fundamental building blocks which allow VeriPhy to *i)* provide guarantees for controller *programs ii)* guarantee safety when real physics matches modeling assumptions and alert us when it does not, and *iii)* preserve those guarantees throughout compilation.

Runtime verification remains a topic of active research and ongoing interest. Of the

many active lines of runtime verification research, we remark that VeriPhy would stand to verify from future research that seeks to ensure hybrid systems monitors are as permissive as possible without compromising safety.

1.2.2.3 End-to-End Approaches

The grand challenge problem of end-to-end verification encompasses any effort which extends formal verification from high-level models to concrete implementations. Among end-to-end efforts, this thesis emphasizes the importance of formal guarantees linking the model to real-world safety at runtime. Runtime verification (Section 1.2.2.2) is essential to providing those guarantees because it cuts the Gordian knot of modeling: a safety argument does not need to *assume* that a physical model is correct if it can instead *observe* that a physical model is complied with at runtime. We furthermore emphasize that the formal link between model and implementation should be provided for a broad class of models with as much automation as possible, and that new proof technologies serve an important enabling role for the development of systems that are correct end-to-end.

We discuss several works which pursue an end-to-end philosophy but do not provide end-to-end guarantees in our sense. We first discuss HA-SPIRAL separately because it requires the most detailed comparison, then discuss several approaches which cannot bridge the gap between physics and models of physics because they do not employ runtime monitoring.

High-Assurance SPIRAL (HA-SPIRAL) (Franchetti et al., 2017) may appear similar to VeriPhy at first: it takes a verified dL model as its input and generates code using a ModelPlex monitor. However, not only does HA-SPIRAL lack our end-to-end guarantees, but its approach makes such guarantees impossible. HA-SPIRAL never reasons semantically about hybrid systems, choosing to argue compilation correctness by syntactic transformation alone. Any end-to-end correctness guarantee for CPS must be semantic because it must transfer guarantees from the high-level semantics of hybrid systems to the low-level semantics of implementation code. An end-to-end approach must also crucially prove that its controller has a safe impact on the physical behavior of the system. Because HA-SPIRAL forgets about hybrid systems as soon as ModelPlex hands it a monitor *formula*, its architecture fundamentally precludes showing that its control *program* has a safe effect on the physical dynamics of the CPS. Lastly, the correctness proofs in HA-SPIRAL stop short of verifying machine code (Zaliva & Franchetti, 2018).

HA-SPIRAL is not without its strengths. It has been applied to geo-fencing and dynamic-window control (Low & Franchetti, 2017), both of which are important applications. One of its most notable strengths is that it places a strong emphasis on specialized compiler optimizations inherited from SPIRAL such as those which optimize vector operations to use special-purpose vector instructions. That reliance on SPIRAL's optimizations could however make provable compilation difficult in the long term: verification of compiler optimizations is known (e.g. in CakeML (Tan et al., 2016)) to require significant effort, and SPIRAL features many optimizations which do not come with formal correctness proofs or formal semantics.

The following works are notable because they start with formal safety proofs about CPSs and end with executable code, but they are not end-to-end because they do not en-

force compliance with physical models at runtime and do not have foundational arguments linking models of code to executable extracted code. In the first work, the proved property does not imply safe avoidance of collisions; in the second, implementation code was written entirely by hand.

- ROSCoq (Anand & Knepper, 2015) is a framework based on the Logic of Events for reasoning about distributed CPSs in Coq with constructive reals and generating verified controllers. Their use of constructive reals is a predecessor to the use of constructive reals in Chapter 5, though VeriPhy does not use computable reals for execution because of their limited performance. The ROSCoq robot model includes ODEs which model physical motion, but the invariant rules provided in their framework are relatively low-level properties of constructive real analysis and constructive ODEs, resulting in significant manual proof. Though the manual proof effort is nontrivial, their case study shows how constructive proofs can be performed in Coq for both discrete control and continuous physical dynamics. They synthesize controllers using Coq's built-in code extraction feature, which does not have an exhaustive formal proof of code extraction correctness.

  ROSCoq does confront the fact that real-world system behavior can diverge from a model, but takes a different approach compared to VeriPhy. Instead of monitoring whether system behavior complies with a model, ROSCoq proves bounds on the magnitude of position error as a function of all error sources in the system. One strength of their approach is that it accounts for error while keeping simple exact ODE dynamics, but the downside is that they do not explicitly prove canonical safety properties such as collision avoidance because, for example, a collision could actually occur when system error is large.

- VeriDrone (Ricketts, Malecha, Alvarez, Gowda, & Lerner, 2015) is a framework for verifying hybrid systems in Coq that relies on a discrete-time temporal logic called RTLA, inspired by TLA (Lamport, 1992). Their framework provides an invariant rule for ODEs, but does not include rules comparable to the other dL proof rules for ODEs. Thus, complex proofs about ODEs likely require greater manual effort in this system. As the name suggests, VeriDrone was used to prove safety of a UAV drone model, specifically safe geofencing: the drone remains within a 3D region constructed as a union of intersections of half-volumes. VeriDrone emphasizes a compositional safety argument where safety of a complex 3D region is reduced to a safety argument for the half-volume primitive. Their semantics use discrete time, which would complicate any potential end-to-end argument because real time is not discrete. Moreover, their proofs are not end-to-end because their experiments are based on control code handwritten in C based on floating-point calculations (Ricketts, 2017, §2.4.2) rather than code automatically generated from a formalization.

In short, while all of the above seek to apply verified controllers on real systems, none of them have our end-to-end guarantees, and neither ROSCoq nor VeriDrone has been successfully applied to ODEs which exercise the full range of reasoning available in the dL family, nor did they directly synthesize correct code from models for which they proved safety in the sense of collision-freedom.

This chapter introduced the thesis statement that Constructive Differential Game Logic (CdGL) enables practical, end-to-end verification of cyber-physical systems. We elaborated on the notions of practicality and end-to-end verification by introducing three characters: the Logician, the Engineer, and the Logic-User. These characters respectively represent the competing needs of theoretical foundations, system implementation, and modeling and verification at scale. We discussed how the thesis' end-to-end verification approach, VeriPhy, resolves these competing needs to significantly greater degree than related work.

Part I

End-to-End Verification of Classical Hybrid Systems

Chapter 2

Formalization of Classical dL

Throughout this thesis, we use *differential dynamic logic* (dL) (Platzer, 2018a, 2008a, 2017a, 2012b) and its relatives to prove safety, liveness, and correctness of hybrid models of cyber-physical systems (CPS's). Much of the value of a formal proof comes from the certainty it provides: if a fact has a proof, we are confident that it is true. We are especially confident when our proof is written in a formal logic which can be mechanically checked by a proof assistant on a computer because, unlike a human, a computer does not grow tired or miss details. A proof system is *sound* if all proved formulas are *valid*, meaning that they are true in all cases.

It is common to prove soundness of a logic using human-readable arguments, and dL has had such a soundness proof since its inception (Platzer, 2007a, 2008a). Specifically, dL is proved sound by introducing a denotational (or model-theoretic) semantics which serves as a ground notion of truth, then proving that dL proof rules are sound with respect to the denotational semantics. However, a human-readable argument for soundness could be incorrect for all the same reasons that a human-readable proof about a CPS could be incorrect: the author might apply a reasoning step incorrectly or skip an important step. The Logician in particular knows that human reasoning is fallible and thus prefers the trustworthiness of formal, machine-checkable reasoning in a proof assistant. Trustworthy proofs are especially important for soundness: the dL proof calculus serves as a gatekeeper for dL proof attempts, so our confidence in every dL proof relies on our confidence in the dL soundness proof. The gatekeeper role served by the dL proof calculus justifies the significant investment of effort required to prove soundness formally: effort is invested in proving soundness *once*, but the benefit of increased trust is paid back *every* time a proof is performed in the dL calculus.

We formalize the soundness theorem of dL in Isabelle/HOL, a general-purpose proof assistant which is strong enough to represent logics and their soundness theorems. The formalization first defines the syntax, semantics, axioms, and rules of dL, then states and proves the soundness theorem. The particular presentation of dL which we formalize is called its *uniform substitution calculus* (Platzer, 2017a), a style of calculus which is noted for its use of concrete formulas as axioms rather than axiom schemata. A strength of uniform substitution calculi is their modularity, a modularity which is kept both by the formalized soundness proof in Isabelle/HOL and the implementation of the KeYmaera X

theorem prover for dL. Our formalization follows KeYmaera X when possible, and we comment on any differences and the reasons for them. To the extent that our uniform substitution calculus agrees with that of KeYmaera X, our effort can also be understood as the *cross-verification* of KeYmaera X in Isabelle/HOL. To demonstrate the extent of our cross-verification, we instrumented KeYmaera X to generate proof-terms which were successfully rechecked in a proofchecker extracted from the formalization (Section 2.8). We discuss the extent to which different parts of KeYmaera X can or cannot be concluded as sound from the proofchecking experiments (Section 2.10).

As reflected in Section 2.10, it is important to take a skeptical perspective when showing soundness of a proof calculus. A particularly skeptical Logician might observe that if it is possible for the dL calculus to contain soundness bugs, it is possible for Isabelle/HOL to contain soundness bugs as well. Likewise, if Isabelle/HOL were cross-verified in another proof assistant, that proof assistant's code would have to be trusted: because soundness of a system can only be shown in a stronger system, *any* formal proof of soundness will rely on soundness of another system. However, the theoretical possibility of a soundness bug in Isabelle/HOL cannot invalidate the demonstrated benefits of our formalization: for example, our effort exposed a previously-unknown and easily-fixed soundness bug in the KeYmaera X prover core (Section 2.10). Because we have written a single Isabelle/HOL proof which gives assurance for all dL proofs, we are far more likely to find and fix a soundness bug in dL than we are to first silently encounter an Isabelle/HOL soundness bug and then fail to address that bug. Our own soundness bug arose in a subtle edge case, reinforcing the folklore wisdom that soundness is most challenging for edge cases, and that proofs which avoid edge cases of the proof calculus are the least likely to encounter proofchecking bugs. If a skeptic remains concerned about the possibility of a soundness bug in Isabelle/HOL compromising our conclusion, the skeptic can further eliminate doubt by providing an independent soundness proof in another proof assistant whose design and foundations differ. Our coauthors did exactly that in one of the publications corresponding to this chapter (Bohrer, Rahli, Vukotic, Völp, & Platzer, 2017), showing soundness of dL in the proof assistant Coq. Subsequent to our work, Platzer independently formalized dGL in Isabelle/HOL (Platzer, 2019a) with an emphasis on simplicity over practicality. His formalization often makes different design decisions from our own, thus it provides a useful point for comparison.

While following chapters of this thesis will introduce new logics which extend dL (Chapter 5, Chapter 6), formalized soundness of dL takes priority over those logics because the reasoning principles used in dL are needed in its extensions as well. In chapters which introduce extensions of dL and prove them sound on paper, we do not mechanize the soundness proofs but instead discuss the challenges that formalization would pose. The major reason that we do not mechanize those results is that formalization of *constructive* games (Chapter 5) and their refinements (Chapter 6) introduces unique foundational challenges beyond those posed by *classical* games (Platzer, 2019a). Conversely, our soundness formalization for dL is also a testament to the usefulness of real analysis and differential equation libraries provided by Isabelle/HOL: those libraries are complete enough that our own work focuses on the specifics of dL rather than general-purpose theorems of differential equations on which soundness relies.

Within the broader context of the thesis, this chapter serves several purposes. Firstly, Classical VeriPhy (Chapter 3) can use the verified proofchecker implemented in this chapter to provide a high degree of confidence in the correctness of a dL proof, which is crucial because Classical VeriPhy's implementation-level correctness guarantees are dependent on the soundness of dL proofchecking. While Constructive VeriPhy (Chapter 8) does not employ the verified proofchecker (because the logic it is based on, called CdGL, differs from dL), a soundness proof for dL helps increase our confidence in fundamental hybrid systems verification principles which arise in multiple logics. Lastly, because this chapter discusses dL formulas and their meaning in detail, this chapter serves a secondary purpose as an introduction to dL. However, the full, formalized uniform substitution calculus for dL also includes some advanced features and implementation details which are not needed to understand the models and proofs used in the rest of the thesis, and thus may be of lesser interest on a first reading. We point out advanced features and implementation details when they arise, but encourage readers unfamiliar with dL to focus on the main features during a first reading.

Section 2.1 introduces basic Isabelle/HOL syntax used in this chapter. Section 2.2 introduces the syntax of the dL uniform substitution calculus we formalized, which is given a denotational semantics in Section 2.3. The dL static semantics of Section 2.4 are used to state side conditions for rules and in a handful of axiom schemata. Section 2.5 introduces the dL axioms and axiom schemata. Section 2.6 introduces the dL rules and rule schemata. Axioms, rules, and schemata are proved sound in Section 2.7. Section 2.8 wraps the proof calculus in a verified proof term checker, from which code is extracted and applied to proof terms extracted from KeYmaera X. Section 2.10 discusses the implications of the formalization for the soundness of dL as implemented in KeYmaera X.

Version Differences. There exist several versions of the Isabelle/HOL formalization discussed here. In addition to multiple published versions (Bohrer et al., 2017, 2018; Bohrer, 2017), some features present in our development repository have not been published to the Archive of Formal Proofs (AFP) as of this writing. The main reason the latest revision has not been published is because its new, more general treatment of identifiers leads to generated code that is unrunnable in practice, while the treatment in the current AFP release can generate runnable code. Our presentation matches the development version most closely because the latest revisions have less syntactic boilerplate than the AFP release. Our code generation and proofchecking experiments are an exception: we use an older version (Bohrer et al., 2018) with the original treatment of identifiers to generate runnable code for the experiments.

In principle, the two versions of the Isabelle/HOL formalization could be unified by adapting the data structures for dL expressions and substitutions to admit efficient representations in the presence of large identifier sets. Specifically, substitutions would be represented as lists of replacement pairs, the same design choice which was taken in the Coq formalization of dL (Bohrer et al., 2017). The expression data structure would be modified to make the arity of functions (Section 2.2.1) and predicates (Section 2.2.4) not depend on the set of available identifiers. While these changes would be useful for the

long-term maintainability of the formalization, they have not been pursued in this chapter because they would constitute a large maintenance task but would not significantly change the conclusions of the chapter.

We explicitly mention when there are major differences between the public release and the development version. In summary, the major implication of the version differences is that when verified proofchecking is desired, one must use the AFP release, which has a greater amount of boilerplate in the formalization and which lacks several convenient extensions provided by the development version such as nondeterministic assignments, strings as identifiers, and a more extensive term language.

2.1 Isabelle/HOL Primer

We introduce basic Isabelle/HOL notation used in this chapter and refer the reader to the literature (Nipkow, Paulson, & Wenzel, 2002; Nipkow & Klein, 2014) for a thorough introduction to Isabelle/HOL. We will use type, term, and predicate definitions, for which purpose we also introduce Isabelle/HOL syntax for terms and formulas. Isabelle/HOL syntax makes heavy use of double quotes `"`, which must be wrapped around all non-trivial top-level terms and formulas. It will suffice an uninitiated reader to read the quotes as parentheses. Whitespace is not significant, except for its standard role of separating identifiers. We typeset mathematical symbols from Isabelle/HOL code using Unicode for readability; Isabelle/HOL's internal representation differs.

Logical notation in Isabelle/HOL requires special attention for two reasons:

- Isabelle/HOL distinguishes terms from propositions
- Isabelle is a *logical framework* which can define many logics. In this framework, different notations are used for the *metalogic* (Isabelle/Pure) and *object logic* (HOL).

The reader of this chapter need not concern themself with the distinctions between Pure and HOL, but each syntax is needed at times for technical reasons, and we introduce all notations that we use. For example, a universal quantifier in Pure is written $\bigwedge$ while a universal quantifier in HOL is written $\forall$. The Pure universal quantifier is easily confused with the HOL conjunction operator $\wedge$. Pure quantifiers and HOL conjunctions can be distinguished by their arity: conjunction ($\wedge$) is an infix binary operator while Pure's universal quantification ($\bigwedge$) uses prefix notation, followed by the quantified variable and quantified formula. An implication in Pure is written with a long double arrow $\Longrightarrow$ while a function uses a shorter double arrow $\Rightarrow$ and HOL implications (likewise, equivalences) use single arrows $\longrightarrow$. Rather than an equality sign, definitions may use the equal-by-definition sign $\equiv$.

Set operations can use standard notations such as membership $\in$, binary union $\cup$, n-ary union $\bigcup$, and binary intersection $\cap$. Set complement is prefix $-$, the set of all values (of some type `'a`) is `UNIV` whose type is `'a set`, and the image of S under function f is `f`S`. The closed interval $[a, b]$ is written `{a..b}`. Set comprehensions are supported. We give several examples with increasing complexity:

- `{x, y}`, the finite set containing exactly x and y,
- `{x. p(x)}`, the set of elements x satisfying characteristic formula `p(x)`, and

- `{x | x y. y² = x}` (with an existentially quantified characteristic formula), the set of values x that are the square of any number y, i.e., the set of nonnegative numbers x. This notation deserves special attention because, compared to the previous examples, it is more distinct from standard textbook notation for mathematical set comprehensions. Formally, an existential set comprehension is defined by introducing a ghost variable standing for the left-hand side, i.e., comprehension `{x | x y. y² = x}` is mathematically defined as $\{u \mid x = u \text{ and } y^2 = x \text{ for some } x, y\}$. All variables which appear in the left-hand side must appear in the list of quantified variables. The set comprehension `{x | x y. y² = x}` should not be confused with the mathematical set $\{u \mid y^2 = x \text{ for some } x, y\}$, which is the universal set $\{u \mid \text{true}\}$.

Product and disjoint sum types are supported with infix notation `*` and `+`. Pairs are constructed `(l, r)` and their components are projected by functions `fst` and `snd`, respectively. The constructors of the sum type are named `Inl` and `Inr` and, like any other datatype, can be discriminated with a `case` expression. Vectors are a wrapper around functions from indices to elements, and new vectors are introduced with an expression `(χ i. e)` which defines each element `e` in terms of its index `i`. The ith element of vector e is projected by writing `e $ i`. A function value is introduced as a λ-expression `(λx. e)` with a body e that can mention an argument x.

Isabelle/HOL supports definitions of inductive (sum-of-products) datatypes using the `datatype` keyword. Datatypes in Isabelle/HOL mirror those of typed functional languages such as Standard ML, but the `datatype` syntax is not identical between the two. The constructors of the datatype are separated by `|` and each constructor name is separated by spaces from its arguments, which are curried:

```
datatype typeName =
   ConstructorA
 | ConstructorB int "int list"
 | ConstructorC int
```

For type constructors with type parameters, the type constructor is written after the argument, as in Standard ML. For example, `int list` is a list of integers. Type `int list` is written with quotes `"` in the datatype definition to distinguish it because it is a non-trivial type family, i.e., not a single identifier. Removing the quotes would result in two arguments whose respective types are `int` and the nonsensical type `list`. The `and` keyword is used to separate any mutually recursive definitions, including datatype definitions:

```
datatype typeNameA =
   ConstructorA
 | ConstructorB typeNameB
and typeNameB =
   ConstructorC
 | ConstructorD typeNameA
```

Names can be given to abbreviate types whose length makes them inconvenient to write in full. Abbreviations use the `type_synonym` keyword. The `type_synonym` keyword, like the `type` definitions of Standard ML and `typedef` keyword of C, abbreviates types

in a non-abstract way that leaves definitions visible during typechecking. In the example below, the right-hand side is wrapped in quotes because, as a rule, Isabelle/HOL requires double quotes around top-level expressions (and type families).

```
type_synonym fiveInts = "int * int * int * int * int"
```

The `type_synonym` keyword should not be confused with Isabelle/HOL's `typedef` keyword, which constructs a new type τ' given an existing type τ and guard predicate P. The new type τ' contains all values $x : \tau$ that satisfy the guard $P(x)$. This chapter can be read without an understanding of `typedef`, but the full formalization uses `typedef` to define an integral numeric type `bword` containing machine words in a bounded range. Every `typedef` comes with `Rep` and `Abs` functions which map τ' to τ and vice-versa. For example, this chapter makes limited use of a function `Rep_bword` which extracts the underlying numeric value of a word. Our treatment of words also uses a function `sint` which converts a machine word to its value as a signed integer.

Recursive functions are introduced with the keywords `fun` or `primrec`[1], which share a common syntax for type signatures and function bodies. Types are annotated with `::`, the keyword `where` separates the signature from the body, and the keyword `and` separates the signatures of mutually recursive functions, before writing the body. In the following example, `functionA` and `functionB` are nonsensical computations on lists which demonstrate mutually recursive function syntax. Used in this example, `[]` is the empty list and `x # xs` prepends element `x` to list `xs`.

```
fun functionA::"int list ⇒ int"
and functionB::"int list ⇒ int"
where
  "functionA [] = 1"
| "functionA (x # []) = 0"
| "functionA (x # y # xs) = x + functionB (y # xs)"
| "functionB [] = 5"
| "functionB (x # xs) = functionB xs * functionA xs"
```

Because the logic of Isabelle/HOL is classical, function definitions are checked for *totality* rather than *termination*. In this chapter, when we speak of termination, we speak specifically of the executable code extracted from a definition.

Isabelle/HOL has an analogous syntax for defining inductive predicates rather than recursive functions. Predicates can be defined inductively by writing the `inductive` keyword and the introduction rules of the predicate. The following example inductively defines what it means for all elements of an integer list to be positive:

```
inductive allPositive::"int list ⇒ bool"
where NilPos:"allPositive []"
| ConsPos:"x > 0 ⇒ allPositive xs ⇒ allPositive (x # xs)"
```

[1]The `fun` keyword is standard practice for most definitions. Our formalization uses `primrec` for recursive functions over datatypes containing functions, as its totality checker happens to handle these recursion patterns better than `fun` does.

The double colon symbol `::` should not be confused with the single colon symbol `:` which serves a very different role. The former is used to ascribe a type to an expression, while the latter is used to assign names. In the `allPositive` example, single colons `:` are used to assign names to each case `NilPos` and `ConsPos`. Naming of cases is used in both functions and inductive predicates so that the name can be used later in proofs to appeal to the definition of the case.

We use the `lemma` keyword to introduce Isabelle/HOL statements which we have proved. Isabelle/HOL's notion of `lemma` and `theorem` are interchangeable from a logical standpoint, and merely indicate significance for the sake of documentation. In the full formalization, the statement is followed by a formal proof.

A `lemma` can optionally use the keywords `fixes`, `assumes`, and `shows` to specify argument terms, assumptions, and conclusions, respectively. The notation `fixes n::int` universally quantifies a variable `n::int`. The notation `assumes pos:"n>0"` introduces an assumption `n>0` which can mention `n` and which can be used as an assumption in a proof step by accessing it via the name `pos`. The notation `shows "n ≥ 0"` says the conclusion is the formula `n ≥ 0`, so the overall theorem statement is `∧ n. (n > 0 ⇒ n ≥ 0)`.

```
lemma posNonneg
  fixes n::int
  assumes pos:"n > 0"
  shows "n ≥ 0"
```

2.2 Syntax

We begin introducing our dL formalization. Any formalization of dL must define the syntax of dL expressions. Expressions in dL are divided into:

- terms, which express real-valued calculations,
- hybrid programs, which are a programmatic syntax for hybrid system models, and
- formulas, which are essential both for stating theorems about hybrid programs and for stating conditions that hold at various points within the program.

In our presentation, we also distinguish a class of *differential programs* which describe the dynamics of ordinary differential equation (ODE) systems; differential programs are often subsumed under hybrid programs in other presentations. As with any uniform substitution calculus, the dL uniform substitution calculus includes explicit syntax for rigid symbols that range over expressions, which in our case are terms, formulas, hybrid programs, and differential programs. This chapter uses Greek letters for metavariables ranging over expressions and bold Latin variables for uniform substitution symbols in the dL object language. Differential programs are the exception, where metavariables begin with ODE and object variables are lowercase Latin variables such as $\mathbf{c}$. We define the syntax of dL both as a recursive grammar and as Isabelle/HOL datatype declarations. *Uniform substitution* symbols, such as function applications $\mathbf{f}$ and functionals $\mathbf{F}$, crucially allow most axioms and rules to be stated with concrete formulas, as in Section 2.5.

As discussed in Section 2.3, program state is a fundamental concept in the semantics of dL. A state assigns a real-number value to each program variable x and differential symbol x'. As a hybrid program is executed, the state is modified. States are discussed in greater detail in Section 2.3; this section discusses state informally to explain the meaning of expressions.

2.2.1 Term Syntax

We discuss the syntax of terms.

Definition 2.1 (Terms of dL)**.** Terms θ, η of dL are defined recursively according to the following grammar:

$$\theta, \eta ::= q \mid x \mid x' \mid \theta + \eta \mid \theta \cdot \eta \mid \theta / \eta \mid -\theta \mid \mathsf{max}(\theta, \eta) \mid \mathsf{min}(\theta, \eta) \mid \mathsf{abs}(\theta) \mid (\theta)' \mid \mathbf{f}(\theta_1, \ldots, \theta_n) \mid \mathbf{F}$$

Here, the literal $q \in \mathbb{Q}$ is a rational constant and $x \in \mathcal{V}$ is a real-valued scalar program variable where $\mathcal{V}$ is the (at most countable) set of all base variable identifiers. In the formalization, $\mathcal{V}$ is further restricted to be an arbitrarily large *finite* set, which simplifies proofs[2] and is no more restrictive in practice. For every base variable $x \in \mathcal{V}$, there is a differential variable $x' \in \mathcal{V}'$ which exists in every state but is primarily used to track the time-derivative of x within an ODE. Our syntax uses distinct constructors for base and differential variables; the choice between distinct constructors (Platzer, 2015b) and a shared constructor (Platzer, 2017a) is an incidental design decision which results in less nesting of proof cases in the former design and fewer top-level proof cases in the latter design. Terms $\theta + \eta$ and $\theta \cdot \eta$ are the sum and product of θ and η. The differential term $(\theta)'$ denotes the total differential of θ, which agrees (Platzer, 2017a, Lem. 35) with the time-derivative of θ within an ODE but has the advantage that it is well-defined in every state. The total differential is the sum of partial derivatives with respect to each variable $x \in \mathcal{V}$, where each partial derivative is scaled by the corresponding x'. The negation of θ is written $-\theta$, quotients are θ/η, and special functions for minimums min, maximums max, and absolute values abs are supported. An uninterpreted function symbol $\mathbf{f}$ stands for an uninterpreted C^1 smooth real function with n arguments, while functionals $\mathbf{F}$ are like functions that depend on the whole state, rather than positional arguments. When describing the substitution data structure, we will want a symbol which refers to the argument(s) of a function. In the written description of the dL calculus, we use $\bullet_i$ as a reserved nullary uninterpreted function symbol which stands for the ith argument of a function[3]. Positive integer exponents θ^k are derived from multiplication as $\theta \cdot \cdots \cdot \theta$ and negative integer exponents are derived with division as $\theta^{-k} = \frac{1}{\theta \cdot \cdots \cdot \theta}$. Roots and rational exponents are not formalized, but are available

[2]Specifically, real vectors can be indexed by variable identifiers. Finite-dimension real vector spaces are always Banach spaces under a Euclidean norm, allowing differentiation of real vector functions, but the set of all infinite real vectors is not a Banach space.

[3]In the formalization, there are always as many arguments as identifiers, and they are distinguished from standard function symbols by a sigil character at the start of their name. Our high, fixed arity was chosen in order to support arbitrarily many arguments without introducing a type system for dL expressions, as may be required when allowing multiple function arities. The downside is that code generation becomes difficult, even practically impossible when the number of variable identifiers is large.

in KeYmaera X. Simple functionals which depend only on base variables x can be derived from functions in the Isabelle/HOL formalization because functions in the formalization can have as many arguments as there are identifiers. We write $\mathbf{f}(\bar{x})$ in the text for simple functionals to represent that their argument vector is the vector $\bar{x}$ of all (base) variables. Simple functionals are only used to formally define a handful of axioms.

Division by zero is prohibited, so that all terms are defined in every state (i.e., total). Presentations of dL vary in their assumptions on totality and continuity. When needed, our formalization will distinguish between *simple terms* which are guaranteed to be C^1 smooth (continuous to the first derivative) and *full terms* which need not be smooth. Simple terms contain only polynomial operations, base variables x, and C^1 function symbols, while full terms can use all term operators. When a full term contains a differential term $(\theta)'$, the differentiated term θ must be simple, so that nested differentials never occur. By distinguishing simple terms from full terms, we provide sound support for differentials without prohibiting non-smooth operators entirely. Simple terms are required to be C^1 rather than merely differentiable so that ODEs with simple terms on their right-hand sides have unique solutions by Picard-Lindelöf.

Throughout the Isabelle/HOL formalization, we use well-formedness predicates to rule out nested differentials. Predicate `dfree θ` says term `θ` is **free** of **d**ifferentials, i.e., simple. Predicate `dsafe θ` says term `θ` is well-formed because it contains no *nested* differentials. Predicates `osafe ODE`, `fsafe φ`, `hpsafe α`, and `ssafe σ` are the respective well-formedness predicates for ODE systems, formulas, hybrid programs, and substitutions. They ensure all terms within the respective expressions are `dsafe`. We describe any additional well-formedness conditions when we describe the corresponding syntactic class. We do not give Isabelle/HOL listings for the full definitions of the well-formedness conditions since the definitions contain many repetitive cases.

In (Bohrer, Fernández, & Platzer, 2019), the present author shows how to define more general terms in an extension of dL with Hilbert's definition description operator, enabling many more term constructs that are convenient in practical proving, the most common of which include roots, trigonometric functions, and conditionals.

In our Isabelle/HOL formalization, dL terms are defined by a datatype `trm`:

```
datatype trm =
  Const lit
| Var ident
| DiffVar ident
| Plus trm trm
| Times trm trm
| Div trm trm
| Neg trm
| Max trm trm
| Min trm trm
| Abs trm
| Differential trm
| Function ident "ident ⇒ trm"
| Functional ident
```

The constructors `Functional`, `Div`, `Neg`, `Max`, `Min`, and `Abs` are currently only available in the development version of the formalization. The type of numeric literals is written `lit` and the type of identifiers is written `ident`. The formalization uses fixed-length words as numeric literals for the sake of generating simple code. Rational-number literals are definable from word literals. The remaining cases of `trm` are in direct correspondence with the recursive grammar definition, except for functions. The arguments of a function symbol are formalized with type `"ident => trm"`, meaning that the formalization treats all functions as having the same number of arguments n, where $n = |\texttt{ident}|$. A fixed arity avoids the need to check type agreement between the types of functions and their argument terms. In the formalization, a derived syntax for low-arity (e.g. unary) functions is defined by setting the remaining arguments to the constant term 0. The major limitation of fixed-arity functions is that when generating code, the space of identifiers must be kept small to avoid allocating unusably-large argument vectors.

The formalization of identifiers is remarkably involved and is a major point of departure between the AFP version and development version. Our formalization requires that the identifier type is finite for two reasons:

- The semantics of differential equations (ODEs) involves calculus over state vectors containing as many elements as there are identifiers. The semantics thus require a Banach space, which is most easily shown by showing finite support. In turn, the easiest way to show finite support is to show that state vectors are finite, meaning there are finitely many identifiers.
- Our dL proofchecker involves computations over data structures which have as many elements as there are identifiers. We can only execute code from our definitions if the type of identifiers is finite[4].

The AFP release uses finite enumeration datatypes for identifiers, while the development version uses bounded-length strings. Even bounded-length strings make code generation impractical, so the code generation experiments (Section 2.9) use the older release.

Setting aside the need for a finite identifier type, substitution (Section 2.6.2) introduces additional considerations for the choice of identifiers. Some cases of the substitution algorithm use reserved identifiers which must not appear elsewhere. One seemingly elegant way to implement reserved identifiers is to treat the identifier type as a type argument and allow the type argument to vary throughout the formalization, with the substitution algorithm using a richer identifier type that can syntactically distinguish reserved symbols. The AFP release takes this approach, but the development version abandons it because of the large number of boilerplate polymorphic type annotations it required. The development version uses a sigil character to distinguish reserved symbols instead, which is closer to the presentation used on paper (Platzer, 2017a) and requires less annotations, but requires additional operations which pack and unpack sigil characters. This chapter assumes the sigil approach, but renames some definitions from the development version for the sake of readability.

[4]It must also satisfy the `enum` locale.

2.2.2 Differential Program Syntax

We now discuss ordinary differential equation systems (ODEs), which are a crucial building block for hybrid systems. Before integrating ODEs into hybrid systems, it is useful for the sake of precision to introduce the class of *differential programs*, which describe an ODE system standing alone. Specifically, a differential program specifies the rates at which variables change in an ODE system:

$$ODE1, ODE2 ::= x' = \theta \mid ODE1, ODE2 \mid \mathbf{c}\{|\text{space}|\}$$

A singleton differential equation is written $x' = \theta$. The right-hand side θ of a singleton ODE $x' = \theta$ must be a simple term so that it does not mention differentials and thus ODEs are always in explicit form. A compound ODE system is built by composing two systems in parallel as $ODE1, ODE2$. Next, we describe differential program constants, which stand for differential programs, and their optional *space constraints*. We write $\mathbf{c}$ for a differential program constant with no space constraint. We write $\mathbf{c}\{|!x|\}$ when the constraint is $\{|!x|\}$, which means $\mathbf{c}$ must not bind[5] variable x, nor x'. Space constraints $\{|!x|\}$ are only used in our formalization of one axiom (DEsys in Fig. 2.1), so they can safely be skipped during a first reading of this chapter, despite being essential for the soundness of DEsys. Unannotated constants $\mathbf{c}$ are also primarily used in axioms of dL, as opposed to theorems stated and proved by users of dL.

Recall that the `type_synonym` keyword in Isabelle/HOL introduces a transparent type synonym, like `typedef` in C or `type` in ML. Space specifier `All` indicates an unannotated symbol $\mathbf{c}$ where all variables may be bound while specifier `NB` indicates $\mathbf{c}\{|!x|\}$ where x and x' are **n**ot **b**ound.

As with the other syntactic classes of dL, ODE systems feature a well-formedness predicate (`osafe ODE` in Section 2.2.1), whose full definition is elided because it is verbose. Predicate `osafe ODE` checks that the right-hand side of each differential equation is simple (`dfree` in Section 2.2.1) and no variable is bound twice by the left-hand sides of two difference equations.

```
type_synonym space = "ident option"
definition All::space = None
definition NB::"ident ⇒ space" = Some

datatype ODE =
  OSing ident trm
| OProd ODE ODE
| OVar ident space
```

Space specifiers are only available in the development version.

[5]In the corresponding KeYmaera X feature, these variables must also not appear as free variables. Both notions are useful, but the formalization uses the weaker notion because it suffices for the rules that have been formalized, with the stronger restriction only needed for differential ghost reasoning on ODE systems containing multiple equations.

2.2.3 Hybrid Program Syntax

Hybrid programs α, β and formulas ϕ, ψ are defined by a simultaneous induction and are modeled with mutually-inductive datatypes. Recall that hybrid programs are a programmatic syntax for the hybrid systems which we use to model CPSs. Hybrid systems thus combine discrete constructs such as assignments, conditionals, branching, and repetition with a continuous construct for evolution of ODEs. Formulas are used to state safety ($[\alpha]\phi$) and liveness ($\langle\alpha\rangle\phi$) theorems of a hybrid system α which respectively say that all executions or some execution of α satisfy postcondition ϕ. Formulas and hybrid programs have a simultaneous inductive definition because formulas also appear in hybrid programs within conditional statements and differential equation statements.

Definition 2.2 (Hybrid programs)**.** Hybrid programs are defined by this grammar:

$$\alpha, \beta ::= x := \theta \mid ?\phi \mid x := * \mid \alpha \cup \beta \mid \alpha; \beta \mid ODE \,\&\, \psi \mid \alpha^* \mid \mathbf{a}$$

Discrete state changes are provided by $x := \theta$, which stores the current value of real-valued (full) term θ in program variable x. Conditionals are expressed with the discrete test $?\phi$, which is a no-op when ϕ is true, or has no executions if ϕ is false. When ϕ is false, one can informally say that the current branch of execution has aborted or failed. Programs can be nondeterministic, and the nondeterministic assignment $x := *$ chooses any value $r \in \mathbb{R}$ to assign to x. In program $\alpha \cup \beta$, either α or β is executed, and the choice between them is made nondeterministically. In program $\alpha; \beta$ the left program α runs first, followed by β, which starts from any resulting state of α. While if-then-else conditional programs are not part of the core dL syntax, they are easily derived from choices, sequencing, and tests: $\mathsf{if}\,(\phi)\{\alpha\}\,\mathsf{else}\,\{\beta\} \equiv ?\phi; \alpha \;\cup\; ?(\neg\phi); \beta$. When ϕ is true, the first branch α has executions and the β branch does not, or vice versa when ϕ is false. Crucially, if a failing test $?\phi$ is understood as an abort, it is a local one: other branches of a surrounding choice can still be executed. Program $ODE \,\&\, \psi$ evolves ODE continuously for a nondeterministically-chosen duration $d \geq 0$ for which ψ remains true as ODE evolves for time d. The formula ψ is called the *domain constraint* and must not mention any differential variables $x' = \mathcal{V}'$. In the iterated program α^*, the body α is run sequentially any finite number of times, including 0. Uniform substitution symbol $\mathbf{a}$ stands for an arbitrary program. We parenthesize hybrid programs $\{\alpha\}$ for clarity and disambiguation as needed.

The Isabelle/HOL definition of programs closely matches the grammar, except that assignments $x' := \theta$ of differential variables x' are given their own constructor[6]. Our program grammar uses the same constructors used in KeYmaera X to represent hybrid programs, albeit with different constructor names.

```
datatype hp =
  EvolveODE ODE formula   (* means x'=f&ψ in dL *)
| Assign ident trm        (* means x:=θ in dL *)
| AssignAny ident         (* means x:=* in dL *)
```

[6]The other incidental difference is that the constructors were written in a different order in the Isabelle/HOL code. This difference has zero impact on the meaning or use of the defined type.

```
| DiffAssign ident trm    (* means x':=θ in dL *)
| Test formula            (* means ?ψ in dL *)
| Choice hp hp            (* means α∪β in dL *)
| Sequence hp hp          (* means α;β in dL *)
| Loop hp                 (* means α* in dL *)
| Pvar ident              (* means a in dL *)
```

Nondeterministic assignments `AssignAny` are only available in the development version, as of this writing.

2.2.4 Formula Syntax

We discuss formulas of dL.

Definition 2.3 (Formulas of dL)**.** Formulas are defined by this (minimal) grammar:

$$\phi, \psi ::= \theta \geq \eta \mid \neg\phi \mid \phi \wedge \psi \mid \exists x\, \phi \mid \langle\alpha\rangle\phi \mid \mathbf{p}(\theta_\mathbf{1}, \ldots, \theta_\mathbf{n}) \mid \mathbf{C}(\phi)$$

Formula $\theta \geq \eta$ compares the terms θ and η, negation is $\neg\phi$, conjunction is $\phi \wedge \psi$, real existential quantifiers are $\exists x\, \phi$, and $\langle\alpha\rangle\phi$ is a dL (diamond, or liveness) modality. Modality $\langle\alpha\rangle\phi$ says there exists some execution of α where ϕ is true in the final state. Program modalities are the distinguishing feature of any dynamic logic, including dL. The predicate symbol $\mathbf{p}(\theta_1, \ldots, \theta_n)$ stands for a predicate with n real arguments, while the context symbol $\mathbf{C}(\phi)$ takes a *formula* for its argument. $\mathbf{C}$ is also sometimes called a predicational or quantifier symbol because, like a quantifier, its truth may depend on the meaning of ϕ in *every* state, not just the current state. Nullary predicational symbols are written $\mathbf{P}, \mathbf{Q}$ and are context symbols that ignore their formula arguments, equivalently they stand for arbitrary formulas. Analogously to simple functionals, simple nullary predicationals can be defined as predicates that depend on all the base variables x. Simple nullary predicationals are implicitly used for the domain constraints of ODEs. They reuse the names $\mathbf{P}, \mathbf{Q}$.

Many other dL connectives are derived from this syntax, including comparisons $\leq, <, =, \neq, >$, first-order connectives, and the dL (box, or safety) modality $[\alpha]\phi$ saying that *all* executions of α end in states satisfying ϕ. Both diamond and box modalities are essential, but our initial examples will emphasize box modalities because they are so widely used in practice. The basic usage of a box modality is a Hoare-like safety specification $\phi \rightarrow [\alpha]\psi$ which says that from every initial state satisfying formula ϕ, every execution of program α satisfies ψ in its final state. We include diamond modalities in the core syntax not because they are more fundamental than box modalities, but because they make certain proofs convenient and because boxes and diamonds are interdefinable in dL:

$$\langle\alpha\rangle\phi \leftrightarrow \neg[\alpha]\neg\phi$$

The type for formulas in Isabelle/HOL differs from the minimal grammar most notably in its representation of predicates. As with functions, the arguments are modeled by a function from identifiers to terms, equivalently a term vector with one entry per legal identifier. This representation supports high-arity predicates; low-arity predicates are derivable by setting the remaining arguments to the term 0. As with functions, the major limitation is that code generation is impractical when the set of legal identifiers is large.

```
and formula =
  Geq trm trm                (* means θ≥η in dL*)
| Not formula                (* means !ψ in dL *)
| And formula formula        (* means ψ&ψ in dL *)
| Exists ident formula       (* means ∃xψ in dL *)
| Diamond hp formula         (* means <α>ψ in dL *)
| Prop ident "ident ⇒ trm"   (* means p(θ1...θn) in dL *)
| InContext ident formula    (* means C(ψ) in dL *)
```

The well-formedness predicates `fsafe ψ` and `hpsafe α` check that every term and differential program contained in a formula or hybrid program is well-formed.

2.2.5 Example Model

Having defined the dL syntax, we take a brief detour to give an example usage of dL syntax before returning to the discussion of the Isabelle/HOL formalization in Section 2.3. For our example, we give a hybrid system model of 1-dimensional robotic driving and its safety specification. While the same example will be studied in Chapter 3, its main purpose in the present chapter is merely to demonstrate how the connectives of dL are used together to express a model and specification. Secondarily, the model presented in this section is also used as a test case for the verified proofchecker which is extracted from the Isabelle/HOL formalization in Section 2.9.

We use a simplified physical model where the robot has instantaneous control over its velocity v and can change its velocity at least once each timestep $T \geq 0$, meaning it is *time-triggered*. The robot's safety goal is to stop while the signed distance d to some destination is nonnegative. That is, we define safety as collision avoidance, which is a common and fundamental safety specification for any mobile CPS.

The driving model follows a common modeling idiom called a *control-plant loop*: hybrid program ctrl implements the control logic that determines velocity v, while a plant is a model of physics which describes continuous system evolution with ODEs. The plant can evolve for bounded time per loop iteration. The loop repeats any finite number of times, including 0. Formula (2.1) expresses safety of the model in a dL formula $\phi \rightarrow [\{\mathsf{ctrl}; \mathsf{plant}\}^*]\psi$ where ϕ is an initial condition and ψ a postcondition. An intuitive reading of $\phi \rightarrow [\{\mathsf{ctrl}; \mathsf{plant}\}^*]\psi$ says that ψ holds for all time, assuming ϕ initially. This intuitive reading is correct only because the nondeterminism in $\{\mathsf{ctrl}; \mathsf{plant}\}^*$ allows it to evolve for an indefinite length of time. In general, well-designed hybrid programs α should be able to evolve indefinitely, to support an intuitive temporal reading.

The initial condition ϕ specifies the signs of system variables and parameters. The signed distance d to the destination, maximum velocity V, and maximum time between control decisions T are all nonnegative. Formula (2.1) says that under these assumptions, all system behaviors result in nonnegative (signed) distance d, representing the notion that the robot has not crashed through its destination.

The controller (2.2) either drives or stops ($\mathsf{drive} \cup \mathsf{stop}$), then resets a local timer $t := 0$ for use in the plant. The drive action is only run when the test $d \geq TV$ passes in (2.3),

which ensures it is safe to keep driving for T time at speed V. In the drive case, the robot can choose any velocity $v := *$ up to the maximum velocity ($?0 \leq v \leq V$). The robot is can always choose to stop (2.4) by setting velocity v to 0.

$$\overbrace{d \geq 0 \land V \geq 0 \land T \geq 0}^{\phi} \rightarrow [\{\mathsf{ctrl}; \mathsf{plant}\}^*] \overbrace{d \geq 0}^{\psi} \tag{2.1}$$

$$\mathsf{ctrl} \equiv \{\mathsf{drive} \cup \mathsf{stop}\};\ t := 0 \tag{2.2}$$

$$\mathsf{drive} \equiv ?d \geq TV;\ v := *;\ ?0 \leq v \leq V \tag{2.3}$$

$$\mathsf{stop} \equiv v := 0 \tag{2.4}$$

$$\mathsf{plant} \equiv \{d' = -v,\ t' = 1 \,\&\, t \leq T\} \tag{2.5}$$

Finally, the plant (2.5) changes the distance according to the chosen velocity v via the differential equation $d' = -v$. Time advances at the rate $t' = 1$, for any duration $t \leq T$. The ODE must finish by time T, after which the program ctrl; plant can optionally repeat and the controller can make its next decision.

2.3 Semantics

The canonical denotational semantics of dL (Platzer, 2008a, 2017a) is a Kripke (Kripke, 1972) semantics, which we use in our formalization. We assume a finite set $\mathcal{V}$ of base identifiers (`ident` in Isabelle/HOL). We define program states ω, ν, μ that assign a real-number value $\omega(x)$ or $\omega(x')$ to every x and x' for $x \in \mathcal{V}$. We also assume interpretations I, J which assign meanings to all rigid symbols $\mathbf{f}, \mathbf{F}, \mathbf{p}, \mathbf{C}, \mathbf{a}$, and $\mathbf{c}$.

In the Isabelle/HOL formalization, the `state` is a pair of maps which respectively assign real numbers to the base variables x and differential variables x'. Each component of the pair is called a `simple_state`, in reference to the fact that only the x component of the state is used when evaluating a simple term. In the Isabelle/HOL formalization, the interpretation is a record. The record fields `Functions`, `Contexts`, `Predicates`, `ODEs`, and `Programs` describe the interpretations of functions, predicationals, predicates, differential programs, and hybrid programs, respectively. The field `ODEBV` is discussed together with the semantics of `EvolveODE`, where it helps faithfully model edge cases of the ODE semantics. In Isabelle/HOL, record fields are accessed with function-like syntax, e.g., `Programs I` for the field of `I` which assigns the interpretations of hybrid programs. Note that the interpretations of functions and predicates accept the `simple_state` type as arguments because functions and predicates have as many arguments in our formalization as there are base variables in the program state.

```
record interp =
  Functions  ::"id ⇒ simple_state ⇒ real"
  Predicates ::"id ⇒ simple_state ⇒ bool"
  Contexts   ::"id ⇒ state set ⇒ state set"
  Programs   ::"id ⇒ (state * state) set"
  ODEs       ::"id ⇒ space ⇒ simple_state ⇒ simple_state"
  ODEBV      ::"id ⇒ id set"
```

The semantic well-formedness predicate for interpretations is named `is_interp`. It ensures that the interpretation of every function symbol is C^1 continuous and that the interpretations of ODEs respect their space constraints, if any. Because the differential of any differentiable function is unique, there is no need to provide an interpretation field for the derivatives of interpretations of functions. Thus, when we investigate Fréchet derivatives in the semantics of differential terms $(\theta)'$, we will ultimately just define an Isabelle/HOL function for the Fréchet derivatives of interpretations of functions, named `FunctionFrechet`. However, there is no harm in thinking of `FunctionFrechet` as a derived field of the interpretation if you find it helpful to do so.

In the dL uniform-substitution calculus, the value $I\omega[\![\theta]\!]$ of a term θ in a state ω and interpretation I is a real number[7]. The semantics of a formula ϕ is written $\omega \in I[\![\phi]\!]$ to say ϕ is true in state ω and interpretation I. The semantics of a hybrid program α is written $(\nu, \omega) \in I[\![\alpha]\!]$ to say it is possible to reach final state ω starting from ν when executing α with interpretation I.

2.3.1 Term Semantics

We discuss the semantics of terms.

Definition 2.4 (Interpretation of dL terms)**.** The (real-number) value of a term θ in state ω, written $I\omega[\![\theta]\!]$, is defined as follows:

1. $I\omega[\![q]\!] = q$
2. $I\omega[\![var]\!] = \omega(var)$ for $var = x$ or x' for some $x \in \mathcal{V}$
3. $I\omega[\![\theta \otimes \eta]\!] = I\omega[\![\theta]\!] \otimes I\omega[\![\eta]\!]$ for $\otimes \in \{+, -, *, /\}$
4. $I\omega[\![-\theta]\!] = -(I\omega[\![\theta]\!])$
5. $I\omega[\![\mathbf{f}(\theta_1, \ldots, \theta_n)]\!] = I(\mathbf{f})(I\omega[\![\theta_1]\!], \ldots, I\omega[\![\theta_n]\!])$
6. $I\omega[\![\mathbf{F}]\!] = I(\mathbf{F})(\omega)$
7. $I\omega[\![(\theta)']\!] = \sum_{x \in \mathcal{V}} \left(\frac{\partial I\omega[\![\theta]\!]}{\partial x} \cdot \omega(x') \right)$

The value of a term is defined inductively. Literals q denote themselves. The values of variables x and x' are looked up in the state. Mathematical operators are evaluated by applying them to the value of each subterm. Functions work like mathematical operators whose meaning is determined by the interpretation. The built-in interpreted functions $\mathsf{min}(\theta, \eta), \mathsf{max}(\theta, \eta)$, and $\mathsf{abs}(\theta)$ are subsumed under the function case $\mathbf{f}(\theta_1, \ldots, \theta_n)$ but their interpretations (e.g. $I(\mathsf{max})$) are fixed to the corresponding mathematical functions. The value of a functional $\mathbf{F}$ is looked up from the interpretation I as with functions $\mathbf{f}$, but takes the entire state as a parameter because functionals represent arbitrary terms and can thus depend on the entire state. The value of a differential term $(\theta)'$ is the total spatial differential of the value of θ, which is the sum of the partial derivatives of $I\omega[\![\theta]\!]$ as each base variable x of ω changes. When the differential $(\theta)'$ appears in the postcondition of an ODE, it is often intuitively read as the time derivative of θ. However, a compositional semantics must define $(\theta)'$ in isolated states and not only in the context of ODEs, for which

[7]Note that later chapters of this thesis will drop the interpretation I because it is used to define the meaning of uniform substitution symbols, which are not a feature of those chapters.

reason the differential variables x' are used to stand in for the rate at which each variable x changes. In isolation, $(\theta)'$ has a well-defined meaning of the rate at which θ changes assuming each x changes at rate $\omega(x')$. The intuitive reading of $(\theta)'$ within an ODE is restored by a lemma (Platzer, 2017a, Lem. 35) showing that an ODE program assigns the time derivative of each bound variable x of the ODE to the corresponding x'.

The Isabelle/HOL formalization of term semantics is divided into two functions which represent two classes of functions. The `sterm_sem` function considers *simple* terms, those whose variable dependencies are only base variables x and which are guaranteed to be C^∞-smooth. The `dterm_sem` function considers full terms which can contain *differentials* and non-smooth operators. A differential term $(\theta)'$ can only differentiate a simple term θ. We distinguish full terms from simple terms to support term operators which do not have differentials, while ensuring every differential term $(\theta)'$ is well-defined. We present `dterm_sem` here; the polynomial cases of `sterm_sem` are identical and the remaining cases of `sterm_sem` are undefined (keyword `undefined` in Isabelle/HOL).

```
primrec dterm_sem::"interp ⇒ trm ⇒ state ⇒ real"
where
  "dterm_sem I (Var x) = (λv. fst v $ x)"
| "dterm_sem I (DiffVar x) = (λv. snd v $ x)"
| "dterm_sem I (Function f args) =
    (λv. Functions I f (χ i. dterm_sem I (args i) v))"
| "dterm_sem I (Neg t) = (λv. - (dterm_sem I t v))"
| "dterm_sem I (Plus t1 t2) =
    (λv. (dterm_sem I t1 v) + (dterm_sem I t2 v))"
| "dterm_sem I (Times t1 t2) =
    (λv. (dterm_sem I t1 v) * (dterm_sem I t2 v))"
| "dterm_sem I (Div t1 t2) =
    (λv. (dterm_sem I t1 v) / (dterm_sem I t2 v))"
| "dterm_sem I (Differential t) =
    (λv. directional_derivative I t v)"
| "dterm_sem I (Functional f) = (λv. Funls I f v)"
| "dterm_sem I (Const b) = (λv. sint (Rep_bword b))"
| "dterm_sem I (Max t1 t2) =
    (λv. max (dterm_sem I t1 v) (dterm_sem I t2 v))"
| "dterm_sem I (Min t1 t2) =
    (λv. min (dterm_sem I t1 v) (dterm_sem I t2 v))"
| "dterm_sem I (Abs t1) = (λv. abs (dterm_sem I t1 v))"
```

The `Var` and `DiffVar` cases project x and x' from the base state `fst v` and differential state `snd v`, respectively, where `fst` and `snd` are the standard Isabelle/HOL functions for projecting components of a pair. Functions and functionals consult the state using function-like record syntax for the interpretation components `Function` and `Funls`. Literals `Const b` interpret the underlying bounded word `b` as a signed integer. The cases for `Neg`, `Plus`, `Times`, `Div`, `Max`, `Min`, and `Abs` apply the respective mathematical operation to the results of recursive calls.

A helper function `directional_derivative` is used in the semantics of differential

terms to bridge the total differentials of dL's semantics with the closely-related concepts of directional differential and Fréchet derivative. When `I` is an interpretation and `t` is a simple term, then `directional_derivative I t` is a function which accepts a state `v` whose components `fst v` and `snd v` assign respective values to each $x \in \mathcal{V}$ and $x' \in \mathcal{V}'$. The helper function `directional_derivative` says that the total derivative in state `v` is equivalent to the directional differential at point `fst v` in direction[8] `snd v`. Because the state space in dL is always some Euclidean space, the directional differential can always be expressed as a Fréchet derivative, which expresses the derivative at point `fst v` as a bounded linear operator over a direction vector.

```
definition directional_derivative::"interp ⇒ trm ⇒ state ⇒ real"
where "directional_derivative I t = (λv. frechet I t (fst v) (snd v))"
```

In this chapter, we refer to different kinds of differentials in order to place a different emphasis: total differentials emphasize the close relationship between differentials in dL and differential forms (Platzer, 2017a), directional differentials emphasize the intuition that `snd v` captures the instantaneous direction and rate of change, and Fréchet derivatives emphasize that dL states are Euclidean vectors which admit vector calculus. The technical differences between Fréchet derivatives and total or directional differential are minor when operating over Euclidean spaces, as the key difference is that total and directional differentials are applicable in more general contexts such as differentiable manifolds.

The `frechet` function determines the Fréchet derivative of (the semantics of) a simple term in a given (simple) state `v`. We define `frechet` by recursion on the term argument; in each case, the derivative is expressed as a bounded linear operator over the direction vector `v'`. Function `frechet` only specifies cases for simple terms `Var`, `Function`, `Plus`, `Times`, and `Const`. We leave the other cases undefined because the remaining terms operators are not guaranteed to preserve differentiability in general.

```
primrec frechet::"interp ⇒ trm ⇒ simple_state ⇒ simple_state ⇒ real"
where
  "frechet I (Var x) v = (λv'. v' • axis x 1)"
| "frechet I (Function f args) v =
    (λv'. FunctionFrechet I f (χ i. sterm_sem I (args i) v)
                              (χ i. frechet I (args i) v v'))"
| "frechet I (Plus t1 t2) v = (λv'. frechet I t1 v v'
                                  + frechet I t2 v v')"
| "frechet I (Times t1 t2) v =
    (λv'. sterm_sem I t1 v * frechet I t2 v v'
        + frechet I t1 v v' * sterm_sem I t2 v)"
| "frechet I (Const r) v = (λv'. 0)"
```

The Fréchet derivative of `Var x` is an operator which returns the value of x' specified by `v'`. For the sake of proof convenience, the `Var x` case is defined by taking the dot product of `v'` with `axis x 1`, where the `axis` function provided by the Isabelle/HOL

[8]The so-called direction vector is not a unit vector in general. Each component x' will be the time-derivative of variable x in typical use, thus their magnitude can be arbitrarily large.

analysis library yields a vector whose `x` component is 1 and whose other components are uniformly zero. The `Plus` and `Time` cases are the standard rules for derivatives of sums and products. The `Function f args` case relies on a symbol `FunctionFrechet` which represents the Fréchet derivative of the *interpretation* of `f` in `I` as a function of the `args` and their Fréchet derivatives. Throughout the formalization, proofs about term derivatives explicitly assume that interpretations `I` always interpret function symbols as C^1 smooth functions, so our formalization expresses `FunctionFrechet` as a definite description, i.e., as *the* function which is the derivative of the interpretation of `f`:

```
fun FunctionFrechet::"interp⇒ident⇒simple_state⇒simple_state⇒real"
  where "FunctionFrechet I i =
    (THE f'. ∀ x. (Functions I i has_derivative f' x) (at x))"
```

where `(_ has_derivative _) at _` is a mixfix notation from the analysis library (Hölzl, Immler, & Huffman, 2013) which we use to express that a function has a specific derivative at a certain point.

The correctness lemma for `frechet` says that `frechet` yields the Fréchet derivative of the interpretation of a term, assuming that all functions have C^1 interpretations (`is_interp`) and the term is simple (`dfree`).

```
lemma frechet_correctness:
  fixes I::interp and ν::simple_state
  assumes "is_interp I"
  assumes "dfree θ"
  shows "((sterm_sem I θ) has_derivative (frechet I θ ν)) (at ν)''
```

The Fréchet derivative correctness lemma is useful because it allows us to characterize the derivative of a term by cases.

This completes the discussion of the interpretation of terms as functions from states to real numbers. A major challenge when formalization terms was tracking which terms are differentiable and reasoning about their derivatives. The following sections employ the term semantics in defining the semantics of formulas and hybrid programs. In particular, differentiability is fundamental to defining the semantics of ODEs.

2.3.2 Formula Semantics

Definition 2.5 (Interpretation of dL formulas)**.** Truth of dL formula ϕ in state ν and interpretation I, written $\nu \in I[\![\phi]\!]$, is defined as follows:

1. $\nu \in I[\![\theta_1 \geq \theta_2]\!]$ iff $I\nu[\![\theta_1]\!] \geq I\nu[\![\theta_2]\!]$
2. $\nu \in I[\![\neg\phi]\!]$ if $\nu \notin I[\![\phi]\!]$,
3. $\nu \in I[\![\phi \wedge \psi]\!]$ iff $\nu \in I[\![\phi]\!]$ and $\nu \in I[\![\psi]\!]$,
4. $\nu \in I[\![\exists x\, \phi]\!]$ iff $\omega \in I[\![\phi]\!]$ for some state ω that agrees with ν except for the value (in $\mathbb{R}$) of x
5. $\nu \in I[\![\langle\alpha\rangle\phi]\!]$ iff $\omega \in I[\![\phi]\!]$ for some ν with $(\nu, \omega) \in I[\![\alpha]\!]$
6. $\nu \in I[\![\mathbf{p}(\theta_1, \ldots, \theta_n)]\!]$ iff $I(\mathbf{p})(I\nu[\![\theta_1]\!], \ldots, I\nu[\![\theta_n]\!])$
7. $\nu \in I[\![\mathbf{C}(\phi)]\!]$ iff $\nu \in I(\mathbf{C})(I[\![\phi]\!])$

A formula ϕ is *valid*, written $\vDash \phi$, if $\nu \in I[\![\phi]\!]$ for all states ν and interpretations I.

The semantics of first-order connectives agree with their usual definitions: comparison formulas compare the values of terms while propositional connectives compose the truth values of subformulas. While our definition of the quantifier $\exists x\, \phi$ is also standard, we remark on the fact that several standard treatments of quantifiers exist, owing to different treatments of variables:

- The dL semantics treat the set of variables as fixed, with every variable defined in every state. Quantifiers simply modify the value of a variable which already exists. Variables in dL work like mutable variables in a program: if a variable x is bound twice, the state's assignment for x is updated each time. A mutable understanding of variables allows us to faithfully model looping programs which modify a variable multiple times.
- In logics where the set of variables is not fixed, quantifiers can be understood as introducing fresh variables. When necessary, quantifiers $\exists x\, \phi(x)$ are α-renamed to $\exists y\, \phi(y)$ for some fresh variable y. The semantics of $\exists y\, \phi(y)$ can then assign a value for the fresh variable y. A fresh understanding of variables is natural for lexically-scoped variables, such as function arguments in functional programs or variables in mathematical quantifiers.
- Because global variables are well-suited to imperative programs and lexical variables are well-suited to mathematical quantifiers, some dynamic logics (Ahrendt et al., 2016) employ both semantics and differentiate between a class of (global) program variables and a class of (lexical) mathematical variables. This approach is not taken in dL so that two distinct classes of variables do not become a source of confusion.

The Isabelle/HOL formalization of formula semantics depends on several helper functions: `dterm_sem` is the interpretation of a term, `repv v x r` updates the value of base variable x to r in base state v, and $(\chi i.\ e)$ introduces a vector by defining each element e in terms of its index i. Recall that function-like syntax is used to access fields (`Predicates` and `Contexts`) of the interpretation, which is a record. Recall that `is_interp` is the well-formedness condition for interpretations which ensures for example that the interpretations of functions are C^1 smooth. It also ensures that the interpretations of ODE symbols respect their space constraints, i.e., `is_interp` captures all well-formedness constraints for interpretations. The `Exists` and `Diamond` cases use Isabelle/HOL's syntax for existentially-quantified set comprehensions. We recall the meaning of existentially-quantified set comprehensions by using the `Exists` case as an example. Let $p(v, r)$ be the mathematical predicate which holds when `(repv v x r) ∈ fml_sem I ψ`, then `{v | v r. (repv v x r) ∈ fml_sem I ψ}` represents the mathematical set $\{u \mid u = v \text{ and } p(v, r), \text{ exists } v, r\}$ which simplifies to $\{v \mid p(v, r) \text{ for some } r\}$ as desired.

The semantics of formulas and programs are described by `fml_sem` and `prog_sem`, whose definitions are mutually recursive. For the sake of exposition, we discuss the formula cases first and return to this definition again when we discuss the program cases.

```
fun fml_sem::"interp ⇒ formula ⇒ state set"
and prog_sem::"interp ⇒ hp ⇒ (state * state) set"
where
```

```
  "fml_sem I (Geq t1 t2) = {v. dterm_sem I t1 v ≥ dterm_sem I t2 v}"
| "fml_sem I (Not ψ) = {v. v ∉ fml_sem I ψ}"
| "fml_sem I (And ψ ψ) = fml_sem I ψ ∩ fml_sem I ψ"
| "fml_sem I (Exists x ψ) = {v | v r. (repv v x r) ∈ fml_sem I ψ}"
| "fml_sem I (Diamond α ψ) =
   {v | v ω. (v, ω) ∈ prog_sem I α ∧ ω ∈ fml_sem I ψ}"
| "fml_sem I (Prop P terms) =
   {v. Predicates I P (χ i. dterm_sem I (terms i) v)}"
| "fml_sem I (InContext c ψ) = Contexts I c (fml_sem I ψ)"
| ...

definition valid::"formula ⇒ bool"
where "valid ψ ≡ (∀ I. ∀ v. is_interp I → v ∈ fml_sem I ψ)"
```

A syntactic proof of a dL formula shows validity, meaning it shows that a formula is true in every state and every interpretation. For that reason, our dL soundness proof will show validity, i.e., it will show that provable formulas are true everywhere.

2.3.3 Hybrid Program Semantics

We discuss the semantics of hybrid programs.

Definition 2.6 (Transition semantics of hybrid programs)**.** The transition relation $I[\![\alpha]\!]$ specifies which states ω are reachable from a state ν by operations of α under an interpretation I. It is defined as follows:

1. $(\nu,\omega) \in I[\![x := \theta]\!]$ iff $\omega(x) = I\nu[\![\theta]\!]$, and for all other variables $z \neq x$, $\omega(z) = \nu(z)$
2. $(\nu,\omega) \in I[\![x := *]\!]$ iff $\omega(z) = \nu(z)$ for all variables $z \neq x$
3. $(\nu,\omega) \in I[\![?\psi]\!]$ iff $\nu = \omega$ and $\nu \in I[\![\psi]\!]$
4. $I[\![\alpha \cup \beta]\!] = I[\![\alpha]\!] \cup I[\![\beta]\!]$
5. $I[\![\alpha;\beta]\!] = \{(\nu,\omega) : (\nu,\mu) \in I[\![\alpha]\!], (\mu,\omega) \in I[\![\beta]\!], \text{ for some } \mu\}$
6. $I[\![\alpha^*]\!] = \big(I[\![\alpha]\!]\big)^*$, the transitive, reflexive closure of $I[\![\alpha]\!]$
7. $(\nu,\omega) \in I[\![ODE\,\&\,\psi]\!]$ iff exists solution $\varphi{:}[0,r] \to \mathcal{S}$ for $r \geq 0$ which satisfies the conditions $\varphi(0) = \nu$ on $\{x' \mid \text{ODE binds } x'\}^{\complement}$, $\varphi(r) = \omega$, and $\varphi \models ODE\,\&\,\psi$ where $\varphi \models ODE\,\&\,\psi$ means there exists a duration $d \geq 0$ such that for all $s \in [0,d]$ the vector field of ODE and time derivative of φ agree at s and $\varphi(s) \in I[\![\psi]\!]$
8. $(\nu,\omega) \in I[\![\mathbf{a}]\!]$ iff $(\nu,\omega) \in I(\mathbf{a})$

The corresponding Isabelle/HOL function is `prog_sem`, which is defined by mutual recursion with `fml_sem`. The Isabelle/HOL code introduces several helper functions: `repd` is the counterpart to `repv` which updates a differential variable x' and the function `O` is relation composition. The differential program semantics `ODE_sem` recursively defines the vector field of the differential program `ODE` at a given state. In `ODE_sem`, ODE constants $\mathbf{c}\{|\text{space}|\}$ derive their meaning from the interpretation I. For ODE constants $\mathbf{c}\{|\text{space}|\}$ where the optional "space" constraint $\{|!x|\}$ is given, a well-formed interpretation must interpret $\mathbf{c}$ as a differential program which does not modify x or x'. In the `EvolveODE` case, the `mk_v` helper function (mnemonic: make v, i.e., make a state)

implements edge cases regarding the final values of differential variables. The relation `solves_ode` is provided by the ODE library (Immler & Traut, 2016). The relation `(sol solves_ode (λt. der t)) {0..t} X` holds if for all $s \in [0, t]$ we have that `der s` is the time derivative of `sol` at time `s` and `sol s` is an element of `X`. That is, `sol` is the solution function, `(λt. der t)` is the vector field of an ODE which is permitted to depend on time, `{0..t}` is the time interval on which the ODE is solved, and X is a set representing a domain constraint. The equation `sol 0 = fst ν` indicates that the base variables of the initial state must agree with the solution; the values of differential variables are allowed to differ, however. The cases for `AssignAny` and `EvolveODE` use existentially-quantified set comprehensions, which have their standard meaning.

```
fun fml_sem::"interp ⇒ formula ⇒ state set"
and prog_sem::"interp ⇒ hp ⇒ (state * state) set"
where
   ...
| "prog_sem I (Assign x t) = {(ν, ω). ω = repv ν x (dterm_sem I t ν)}"
| "prog_sem I (DiffAssign x t) = {(ν, ω). ω =
     repd ν x (dterm_sem I t ν)}"
| "prog_sem I (AssignAny x) = {(ν, ω) | ω ν r. ω = repv ν x r}"
| "prog_sem I (Test φ) = {(ν, ν). ν ∈ fml_sem I φ}"
| "prog_sem I (Choice α β) = prog_sem I α ∪ prog_sem I β"
| "prog_sem I (Sequence α β) = prog_sem I α O prog_sem I β"
| "prog_sem I (Loop α) = (prog_sem I α)*"
| "prog_sem I (EvolveODE ODE φ) =
  ({(ν, mk_v I ODE ν (sol t)) | ν sol t.
      t ≥ 0 ∧
      (sol solves_ode (λ_. ODE_sem I ODE)) {0..t}
        {x. mk_v I ODE ν x ∈ fml_sem I φ} ∧
      sol 0 = fst ν})"
| "prog_sem I (Pvar p) = Programs I p"
```

The helper function `mk_v` implements the rule that when x is not mentioned in ODE, then x' is never modified, else it is modified so that the ODE (for instance, $x' = \theta$) is a true equality. Importantly, unmentioned variables are treated differently from variables x' where the ODE contains the equation $x' = 0$, because the latter program sets x' to 0 while the other leaves it unchanged. In the following discussion, we call the first class of variables *implicitly constant* because they are not mentioned in the program text, while the latter class are explicitly constant. In Section 2.4, it is important that the final state of an ODE only updates x' for explicitly-mentioned variables, not implicit constants, so that the set of *bound variables* modified by a program is not needlessly large and can admit an intuitive recursive definition. The exact definition of `mk_v` is surprisingly subtle because implicit and explicit constants must be distinguished both syntactically *and semantically*. Because the differential program symbol `OVar c sp` must range over *arbitrary* differential programs, the interpretation `I` must distinguish implicit and explicit constant variables. Specifically, the interpretation field `ODEBV` specifies for each differential program symbol **c** the set of bound (base) variables. For all variables x specified by `ODEBV`, the semantics

of **c** will update x' as if x were an explicit variable. This subtle treatment is important because it means that when substitution (Section 2.6.2) replaces **c** with an ODE that does explicitly bind x, the meaning of the ODE is preserved. The full definition of `mk_v` is divided into four definitions.

The function `ODE_vars` computes the set of (base) variables bound by a given ODE in a given interpretation.

```
fun ODE_vars::"interp ⇒ ODE ⇒ ident set"
  where
  "ODE_vars I (OVar c sp) = ODEBV I c sp"
| "ODE_vars I (OSing x θ) = {x}"
| "ODE_vars I (OProd ODE1 ODE2) = ODE_vars I ODE1 ∪ ODE_vars I ODE2"
```

The function `semBV` extends the bound variable set to include both base variables (encoded as left injections) and differential variables (right injections). Recall that the notation `f ` S` stands for the image of `S` under function `f`, where `Inl` and `Inr` are the left and right injection constructors.

```
fun semBV::"interp ⇒ ODE ⇒ (ident + ident) set"
  where "semBV I ODE = Inl `(ODE_vars I ODE) ∪ Inr `(ODE_vars I ODE)"
```

The function `mk_xode` computes the hypothetical final state which would arise if *every* variable were explicitly bound.

```
fun mk_xode::"interp ⇒ ODE ⇒ simple_state ⇒ state"
where "mk_xode I ODE sol = (sol, ODE_sem I ODE sol)"
```

The function `mk_v` assembles the final state by determining which variables are bound, taking the bound variables from the state proposed by `mk_xode`, and taking all other variables from the initial state `v`. A definite description (keyword `THE`) defines the final state `ω` as the state that agrees (`Vagree`) with initial state `v` on variables that are not semantically bound and agrees with `(mk_xode I ODE sol)` on all other variables.

```
definition mk_v::"interp ⇒ ODE ⇒ finite state ⇒ simple_state ⇒ state"
where "mk_v I ODE ν sol = (THE ω.
  Vagree ω ν (- semBV I ODE)
∧ Vagree ω (mk_xode I ODE sol) (semBV I ODE))"
```

The choice to define `mk_v` is not an essential one. This definition was used because it is common to reason about the final state of an ODE program by comparing it to the initial state `v` and to the simplified semantics described by `mk_xode`. Though the definition of ODE semantics is subtle, past work (Platzer, 2017a) has demonstrated the value of the present semantics: practical use of substitution (Section 2.6.2) demands strong notions of variable binding. Moreover, by dividing the semantics of ODEs into several helper functions, we make it possible for our formalization to prove lemmas about each helper function and thus minimize proof redundancy.

2.4 Static Semantics

A significant soundness advantage of uniform substitution (Platzer, 2015b, 2017a) is that axioms are not schemata with subtle side conditions, but rather side conditions are captured once and for all in the substitution rule[9]. Those side conditions employ notions of free and bound variables and signatures, which we collectively call the *static semantics* of dL. The functions described in this section are the same functions used in prior work (Platzer, 2015b, 2017a), expressed in Isabelle/HOL.

The *signature* $\Sigma(e)$ of an expression e is the set of uniform substitution identifiers which appear free in e. The Isabelle/HOL formalization distinguishes term symbols, formula symbols, and hybrid-program-or-differential-program symbols using disjoint sums (+): In the signature definition, the comprehension `{Inl x | x. x ∈ (⋃i. SIGT (args i))}` represents (for example) the set of all `Inl x` where the argument `x` belongs to the n-ary union `(⋃i. SIGT (args i))`.

The signature function for terms is given first:

```
primrec SIGT::"trm ⇒ ident set"
where
  "SIGT (Var var) = {}"
| "SIGT (Const r) = {}"
| "SIGT (Function var args) = {var} ∪ (⋃i. SIGT (args i))"
| "SIGT (Functional var) = {var}"
| "SIGT (Plus t1 t2) = SIGT t1 ∪ SIGT t2"
| "SIGT (Times t1 t2) = SIGT t1 ∪ SIGT t2"
| "SIGT (Div t1 t2) = SIGT t1 ∪ SIGT t2"
| "SIGT (Max t1 t2) = SIGT t1 ∪ SIGT t2"
| "SIGT (Min t1 t2) = SIGT t1 ∪ SIGT t2"
| "SIGT (Abs t1) = SIGT t1"
| "SIGT (DiffVar x) = {}"
| "SIGT (Differential t) = SIGT t"
| "SIGT (Functional var) = {var}"
```

The signature function for ODEs follows the function for terms:

```
primrec SIGO::"ODE ⇒ (ident + ident) set"
where
  "SIGO (OVar c _) = {Inr c}"
| "SIGO (OSing x θ) = {Inl x | x. x ∈ SIGT θ}"
| "SIGO (OProd ODE1 ODE2) = SIGO ODE1 ∪ SIGO ODE2"
```

The signature functions for programs and formulas are mutually recursive:

```
primrec SIGP::"hp      ⇒ (ident + ident + ident) set"
and     SIGF::"formula ⇒ (ident + ident + ident) set"
where
```

[9]Our calculus is not purely uniform-substitution based, in the sense that a few axioms must be formulated as schemata for soundness.

```
  "SIGP (Pvar var) = {Inr (Inr var)}"
| "SIGP (Assign var t) = {Inl x | x. x ∈ SIGT t}"
| "SIGP (DiffAssign var t) = {Inl x | x. x ∈ SIGT t}"
| "SIGP (AssignAny var) = {}"
| "SIGP (Test p) = SIGF p"
| "SIGP (EvolveODE ODE p) = SIGF p ∪
    {Inl x | x. Inl x ∈ SIGO ODE} ∪
    {Inr (Inr x) | x. Inr x ∈ SIGO ODE}"
| "SIGP (Choice a b) = SIGP a ∪ SIGP b"
| "SIGP (Sequence a b) = SIGP a ∪ SIGP b"
| "SIGP (Loop a) = SIGP a"
| "SIGF (Geq t1 t2) = {Inl x | x. x ∈ SIGT t1 ∪ SIGT t2}"
| "SIGF (Prop var args) = {Inr (Inr var)} ∪
                          {Inl x | x. x ∈ (Ui. SIGT (args i))}"
| "SIGF (Not p) = SIGF p"
| "SIGF (And p1 p2) = SIGF p1 ∪ SIGF p2"
| "SIGF (Exists var p) = SIGF p"
| "SIGF (Diamond a p) = SIGP a ∪ SIGF p"
| "SIGF (InContext var p) = {Inr (Inl var)} ∪ SIGF p"
```

The *free variables* $\mathrm{FV}(e)$ of an expression e are those variables x that appear *free*, i.e., not under a binder of x. Base variables x are distinguished from differential variables x'. Individual variables are collected at leaves and collected inductively. Differential terms depend on both base variables and their primed counterparts, which indicate the change rates of each base variable. Symbols which stand for arbitrary terms, formulas, or programs may depend on every variable. In a modality, a free variable of the postcondition is no longer free if it is uniquely determined (must-bound) by the modal program, likewise for sequential composition programs.

The free variable function for terms is given first. In the differential term case $(\theta)'$, recall that the value of a differential term depends on both base variables and differential variables; for that reason, the differential term case ensures for all free variables of θ that both the base variable and differential variable are considered free variables of $(\theta)'$.

```
primrec FVT::"trm ⇒ (ident + ident) set"
where
  "FVT (Var x) = {Inl x}"
| "FVT (Const x) = {}"
| "FVT (Function f args) = (Ui. FVT (args i))"
| "FVT (Functional f) = UNIV"
| "FVT (Plus f g) = FVT f ∪ FVT g"
| "FVT (Times f g) = FVT f ∪ FVT g"
| "FVT (Div f g) = FVT f ∪ FVT g"
| "FVT (Max f g) = FVT f ∪ FVT g"
| "FVT (Min f g) = FVT f ∪ FVT g"
| "FVT (Abs f) = FVT f"
| "FVT (Differential f) =
    (Ux∈{x. Inl x ∈ (FVT f)}. {Inl x,Inr x})
```

```
   ∪(⋃x∈{x. Inr x ∈ (FVT f)}. {Inl x,Inr x})"
| "FVT (DiffVar x) = {Inr x}"
```

The free variable function for ODEs is given second:

```
primrec FVO::"ODE ⇒ ident set"
where
  "FVO (OVar c sp) = UNIV"
| "FVO (OSing x θ) = {x} ∪ {x. Inl x ∈ FVT θ}"
| "FVO (OProd ODE1 ODE2) = FVO ODE1 ∪ FVO ODE2"
```

The free variable functions for formulas and programs are mutually recursive:

```
primrec FVF::"formula ⇒ (ident + ident) set"
and     FVP::"hp ⇒ (ident + ident) set"
where
  "FVF (Geq f g) = FVT f ∪ FVT g"
| "FVF (Prop p args) = (⋃i. FVT (args i))"
| "FVF (Not p) = FVF p"
| "FVF (And p q) = FVF p ∪ FVF q"
| "FVF (Exists x p) = FVF p - {Inl x}"
| "FVF (Diamond α p) = FVP α ∪ (FVF p - MBV α)"
| "FVF (InContext C p) = UNIV"
| "FVP (Pvar a) = UNIV"
| "FVP (Assign x θ) = FVT θ"
| "FVP (DiffAssign x θ) = FVT θ"
| "FVP (AssignAny x) = {}"
| "FVP (Test ψ) = FVF ψ"
| "FVP (EvolveODE ODE ψ) = BVO ODE ∪ (Inl ` FVO ODE) ∪ FVF ψ"
| "FVP (Choice α β) = FVP α ∪ FVP β"
| "FVP (Sequence α β) = FVP α ∪ (FVP β - MBV α)"
| "FVP (Loop α) = FVP α"
```

The bound variables $\mathrm{BV}(\alpha)$ of a program α (or formula ϕ) are those modified on at least one execution path, while the must-bound variables $\mathrm{MBV}(\alpha)$ are bound on every path.

The bound variables of ODEs are given first:

```
fun BVO::"ODE ⇒ (ident + ident) set"
where
  "BVO (OVar c (Some x)) = -{Inl x, Inr x}"
| "BVO (OVar c None) = UNIV"
| "BVO (OSing x θ) = {Inl x, Inr x}"
| "BVO (OProd ODE1 ODE2) = BVO ODE1 ∪ BVO ODE2"
```

The bound variables of programs are given second:

```
fun BVP::"hp ⇒ (ident + ident) set"
where
  "BVP (Pvar a) = UNIV"
| "BVP (Assign x θ) = {Inl x}"
```

```
| "BVP (DiffAssign x θ) = {Inr x}"
| "BVP (AssignAny x θ) = {Inl x}"
| "BVP (Test ψ) = {}"
| "BVP (EvolveODE ODE ψ) = BVO ODE"
| "BVP (Choice α β) = BVP α ∪ BVP β"
| "BVP (Sequence α β) = BVP α ∪ BVP β"
| "BVP (Loop α) = BVP α"
```

The bound variables of formulas are given next. In contrast to free variables of formulas, bound variables of formulas will not be essential to the proof of soundness.

```
fun BVF::"formula ⇒ (ident + ident) set"
where
  "BVF (Geq f g) = {}"
| "BVF (Prop p dfun_args) = {}"
| "BVF (Not p) = BVF p"
| "BVF (And p q) = BVF p ∪ BVF q"
| "BVF (Exists x p) = {Inl x} ∪ BVF p"
| "BVF (Diamond α p) = BVP α ∪ BVF p"
| "BVF (InContext C p) = UNIV"
```

The must-bound variables of a program are given last:

```
fun MBV::"hp ⇒ (ident + ident) set"
where
  "MBV (Pvar a) = {}"
| "MBV (Choice α β) = MBV α ∩ MBV β"
| "MBV (Sequence α β) = MBV α ∪ MBV β"
| "MBV (Loop α) = {}"
| "MBV (EvolveODE ODE _) =
    (Inl ` (ODE_dom ODE)) ∪ (Inr ` (ODE_dom ODE))"
| "MBV α = BVP α"
```

Program symbols `a` might bind any variable, but need bind none. Assignments and quantifiers bind their left-hand side. Loops only bind variables sometimes, because they might run for zero iterations. The must-bound variables of an ODE are those that explicitly appear on the left-hand side of some singleton equation, a notion expressed by the (omitted) helper function `ODE_dom`. A differential program symbol **c** has no must-bound variables, but *may* bind any variable that is not explicitly taboo (`Some x`).

2.5 Axioms

A dL formula is proved by decomposing a formula with axioms until the proof is complete or any remaining goals belong to decidable languages such as first-order arithmetic. In the dL uniform substitution calculus, axioms are simply individual formulas, many of which are listed in Fig. 2.1. Axiom [·] says that the diamond and box modalities are interdefinable, so it suffices to give axioms for one of $[\mathbf{a}]\mathbf{P}$ or $\langle\mathbf{a}\rangle\mathbf{P}$. The following rules decompose hybrid

programs syntactically, in harmony with their semantics. Axioms $\langle := \rangle$ and $\langle :* \rangle$ respectively reduce deterministic assignments using substitution and reduce nondeterministic assignments to quantifiers. Tests (axiom $\langle ? \rangle$) and choices (axiom $\langle \cup \rangle$) reduce propositionally, and sequential composition becomes a nested modality (axiom $\langle ; \rangle$). Loops are more subtle. Simple proofs can case on whether the loop evolves for at least one iteration (axiom $\langle * \rangle$), but loop proofs often need (co-)inductive reasoning (axiom I), in which case soundness demands that the (co-)inductive step holds no matter how many executions have already occurred. We present the (coinductive) invariant-based axiom for box loop properties $[\mathbf{a}^*]\mathbf{P}$, which is interderivable with an inductive variant-based axiom for diamonds $\langle \mathbf{a}^* \rangle \mathbf{P}$. Modal modus ponens (axiom K) applies implications under modalities and vacuity (axiom V) says that *nullary* predicates are preserved under modalities. Axiom V is an excellent example of how predicates and predicationals differ. It would be unsound if $\mathbf{p}()$ were replaced by a predicational $\mathbf{P}$ because most dL modalities modify program variables, and do not preserve the truth of *all* formulas ϕ, but they *do* preserve formulas such as $1 > 0$ that mention no variables. Axiom V is much more useful when realizing that the substitution rule (Section 2.6.2) permits $\mathbf{p}()$ to mention variables so long as they are not modified in $\mathbf{a}$.

In contrast to some connectives, there is no single axiom that captures all reasoning for ODEs, so a combination of axioms are used. Axioms DC, DI, DEsys, and DW are often used together: we identify a series of invariants for an ODE, then *cut* each invariant into the domain constraint, prove it by *induction*, then conclude the postcondition by applying any differential *effect* and finally *weakening* away the ODE, which is no longer needed once the domain constraint is strong enough to entail the postcondition. The differential effect axiom DEsys uses the syntax $\mathbf{c}\{|!x|\}$ to formalize the intuitive fact that the composed system $x' = \mathbf{f}(\bar{x}), \mathbf{c}\{|!x|\}$ cannot bind x twice. In DEsys, it is important that the space specifier $\mathbf{c}\{|!x|\}$ allows free occurrences of x, since DEsys is applied to a wide variety of ODEs, many of which feature such free occurrences. In DEsys, $\mathbf{f}(\bar{x})$ is a simple functional which can depend on all base variables and no differential variables. The explicit space $\mathbf{c}\{|!x|\}$ is necessary in the semantic soundness proof of DEsys but not in the KeYmaera X implementation of DEsys where syntactic data structure invariants ensure uniqueness of bound ODE variables. Advanced proofs can use differential ghosts (axiom DG) to introduce continuously-changing variables for the sake of the proof, which are often used to establish invariants regarding exponential decay properties. In DG, note that $\mathbf{a}$ and $\mathbf{b}$ are unary function symbols which are used to define a linear equation, not to be confused with the typical use of these identifiers as program symbols. The KeYmaera X core features a generalization of (axiom DG) to systems, which has not been formalized as of this writing. Axiom DI is a sound, simplified axiom for ODEs of a single variable. In contrast to past descriptions of dL on paper (Platzer, 2017a), our Isabelle/HOL formalization cannot directly express the simplified axiom DI because it uses a notion of differential *formula* $(\phi)'$ which we lack. The absence of differential formulas $(\phi)'$ is overcome using the observation that DI arguments for formulas of form $\theta > \eta$ and $\theta \geq \eta$ suffice to derive (Platzer & Tan, 2020; Platzer, 2018a) all instances of the general DI axiom. A deeper issue, however, is that the natural generalization of axiom DI to ODE systems, used in KeYmaera X, is unsound unless a side condition is added. Adding those side conditions results in an axiom

$$([\cdot]) \quad [\mathbf{a}]\mathbf{P} \leftrightarrow \neg\langle\mathbf{a}\rangle\neg\mathbf{P} \qquad (\langle;\rangle) \quad \langle\mathbf{a};\mathbf{b}\rangle\mathbf{P} \leftrightarrow \langle\mathbf{a}\rangle\langle\mathbf{b}\rangle\mathbf{P}$$

$$(\langle:=\rangle) \quad \langle x := \mathbf{f}()\rangle\mathbf{p}(x) \leftrightarrow \mathbf{p}(\mathbf{f}()) \qquad (\langle*\rangle) \quad \mathbf{P} \vee \langle\mathbf{a}\rangle\langle\mathbf{a}^*\rangle\mathbf{P} \rightarrow \langle\mathbf{a}^*\rangle\mathbf{P}$$

$$(\langle:*\rangle) \quad \langle x := *\rangle\mathbf{p}(x) \leftrightarrow \exists x\, \mathbf{p}(x) \qquad (\mathrm{I}) \quad \mathbf{P} \wedge [\mathbf{a}^*](\mathbf{P} \rightarrow [\mathbf{a}]\mathbf{P}) \rightarrow [\mathbf{a}^*]\mathbf{P}$$

$$(\langle?\rangle) \quad \langle ?\mathbf{Q}\rangle\mathbf{P} \leftrightarrow (\mathbf{Q} \wedge \mathbf{P}) \qquad (\mathrm{V}) \quad \mathbf{p}() \rightarrow [\mathbf{a}]\mathbf{p}()$$

$$(\langle\cup\rangle) \quad \langle\mathbf{a} \cup \mathbf{b}\rangle\mathbf{P} \leftrightarrow \langle\mathbf{a}\rangle\mathbf{P} \vee \langle\mathbf{b}\rangle\mathbf{P} \qquad (\mathrm{K}) \quad [\mathbf{a}](\mathbf{P} \rightarrow \mathbf{Q}) \rightarrow ([\mathbf{a}]\mathbf{P} \rightarrow [\mathbf{a}]\mathbf{Q})$$

$$(\mathrm{DW}) \quad [\mathbf{c}\,\&\,\mathbf{Q}](\mathbf{Q} \rightarrow \mathbf{P}) \leftrightarrow [\mathbf{c}\,\&\,\mathbf{Q}]\mathbf{P}$$

$$(\mathrm{DC}) \quad \big([\mathbf{c}\,\&\,\mathbf{Q}]\mathbf{P} \leftrightarrow [\mathbf{c}\,\&\,\mathbf{Q} \wedge \mathbf{C}]\mathbf{P}\big) \leftarrow [\mathbf{c}\,\&\,\mathbf{Q}]\mathbf{C}$$

$$(\mathrm{DEsys}) \quad [x' = \mathbf{f}(\bar{x}), \mathbf{c}\{|\mathrm{!x}|\}\,\&\,\mathbf{Q}]\mathbf{P} \leftrightarrow [\mathbf{c}\{|\mathrm{!x}|\}, x' = \mathbf{f}(\bar{x})\,\&\,\mathbf{Q}][x' := \mathbf{f}(\bar{x})]\mathbf{P})$$

$$(\mathrm{DI}) \quad [x' = \mathbf{f}(x)\,\&\,\mathbf{p}(x)]\mathbf{p}(x) \leftarrow \big(\mathbf{p}(x) \rightarrow \mathbf{p}(x) \wedge [x' = \mathbf{f}(x)\,\&\,\mathbf{p}(x)](\mathbf{p}(x))'\big)$$

$$(\mathrm{DIsys}{>}) \quad [ODE\,\&\,\psi]\theta > \eta \leftarrow \big((\psi \rightarrow \theta > \eta) \wedge [ODE\,\&\,\psi](\theta)' \geq (\eta)'\big)\ ^{1}$$

$$(\mathrm{DIsys}{\geq}) \quad [ODE\,\&\,\psi]\theta \geq \eta \leftarrow \big((\psi \rightarrow \theta \geq \eta) \wedge [ODE\,\&\,\psi](\theta)' \geq (\eta)'\big)\ ^{1}$$

$$(\mathrm{DG}) \quad [x' = \mathbf{f}(x)\,\&\,\mathbf{p}(x)]\mathbf{p}(x) \leftrightarrow \exists y\, [x' = \mathbf{f}(x), y' = \mathbf{a}(x)y + \mathbf{b}(x)\,\&\,\mathbf{p}(x)]\mathbf{p}(x)$$

$$(\mathrm{DS}) \quad [x' = \mathbf{f}()\,\&\,\mathbf{p}(x)]\mathbf{p}(x) \leftrightarrow \forall t{\geq}0\, \big((\forall 0{\leq}s{\leq}t\, \mathbf{p}(x + \mathbf{f}()s)) \rightarrow [x := x + \mathbf{f}()t]\mathbf{p}(x)\big)$$

$$(q') \quad (q)' = 0 \qquad (x') \quad (x)' = x' \qquad (+') \quad (\mathbf{f}(\bar{x}) + \mathbf{g}(\bar{x}))' = (\mathbf{f}(\bar{x}))' + (\mathbf{g}(\bar{x}))'$$

$$(-') \quad (\mathbf{f}(\bar{x}) - \mathbf{g}(\bar{x}))' = (\mathbf{f}(\bar{x}))' - (\mathbf{g}(\bar{x}))' \qquad (\cdot') \quad (\mathbf{f}(\bar{x}) \cdot \mathbf{g}(\bar{x}))' = (\mathbf{f}(\bar{x}))' \cdot \mathbf{g}(\bar{x}) + \mathbf{f}(\bar{x}) \cdot (\mathbf{g}(\bar{x}))'$$

[1] $\mathrm{FV}(\theta) \cup \mathrm{FV}(\eta) \subseteq \mathrm{MBV}(ODE) \cap \mathcal{V}$

Figure 2.1: Axioms of dL.

schema. To ensure soundness while remaining within the formalized fragment of dL, our formalization contains two axiom schemata DIsys$>$ and DIsys$\geq$ which soundly implement the $>$ and $\geq$ cases of differential induction, from which the general case is derivable. Axioms q', x', $+'$, $-'$, and $\cdot'$ syntactically compute the differential of a term, which is a key step for completing a differential invariant argument of axiom DI. Because only simple terms have differentials, axioms $+'$, $-'$, and $\cdot'$ use simple functionals $\mathbf{f}(\bar{x})$ which can depend on all base variables and no differential variables. While KeYmaera X supports an axiom for syntactic differentiation of division terms, we do not formalize that axiom because division lies outside the simple fragment, i.e., not every division term has a derivative in every state. Reasoning about an ODE by inspecting its solution is a natural alternative to invariant reasoning, but an ODE solver is far too complicated to make ODE-solving an axiom in a minimalistic calculus, so axiom DS expresses the solution of a *constant, singleton* ODE.

Instead of providing a single axiom schema for ODE solving, the dL uniform substitution calculus allows solution reasoning to be derived from the core ODE axioms (Platzer, 2017a), so that ODE solving is not soundness-critical. KeYmaera X uses the core ODE axioms to implement a solution rule for ODEs that do have some simple closed-form solution $y(t)$:

$$(\langle' \rangle) \quad \langle x' = \theta \rangle \phi \leftrightarrow \exists t{\geq}0 \, \langle x := y(t) \rangle \phi \quad \text{where } y'(t) = \theta$$

Solution reasoning is convenient for systems with simple solutions, though invariant reasoning scales better to ODEs whose solutions are complex or not closed-form. In KeYmaera X, a large library of tactics (Fulton et al., 2017) provides extensive automation for invariant proofs too, ranging from automation of steps such as DEsys to generating invariants (Sogokon et al., 2019). The Isabelle/HOL formalization of the dL axioms is straightforward: each axiom is declared as a formula, which is proved valid in Section 2.7.

2.6 Rules

The dL substitution calculus has a handful of deduction rules, which can be classified as axiomatic rules, sequent rules, renaming, and substitution. Like axioms, the axiomatic rules are concrete, not schematic, and are defined in the Isabelle/HOL formalization by listing the conclusion and premise formulas. The axiomatic rule data structure is also used as KeYmaera X's notion of proof state[10]: the conclusion represents the theorem being proved, while premises represent the open goals of the proof. A proof is finished when no open goals remain, represented by a rule with no premises (i.e., a derived axiom). In every rule except substitution[11], the premises and conclusions are *local validity* statements. A formula ϕ is locally valid in interpretation I if $\omega \in I[\![\phi]\!]$ for all states ω. A rule is locally sound if for all interpretations I, local validity of all premises in interpretation I implies local validity of the conclusion in interpretation I. That is, the premises and conclusion use the same interpretation, but free program variables are universally quantified in the premises and universally quantified again in the conclusion. Every locally sound rule is

[10]also a called a `Provable` in KeYmaera X terminology

[11]In substitution, the premise and conclusion are simple validity statements, meaning their interpretations may differ.

$$\text{(G)}\quad \frac{\mathbf{P}}{[\mathbf{a}]\mathbf{P}} \qquad\qquad \text{(CE)}\quad \frac{\mathbf{F}=\mathbf{G}}{\mathbf{p}(\mathbf{F})=\mathbf{p}(\mathbf{G})}$$

$$\text{(M)}\quad \frac{\mathbf{P}\vdash\mathbf{Q}}{[\mathbf{a}]\mathbf{P}\vdash[\mathbf{a}]\mathbf{Q}} \qquad\qquad \text{(CQ)}\quad \frac{\mathbf{P}\leftrightarrow\mathbf{Q}}{\mathbf{C}(\mathbf{P})\leftrightarrow\mathbf{C}(\mathbf{Q})}$$

Figure 2.2: Axiomatic rules.

sound. Universal quantification of free program variables is essential to the correct reading of certain rules. Rule G is one example: if $\mathbf{P}$ is true in every state, then $[\mathbf{a}]\mathbf{P}$ is true in every state. Local soundness means that $\mathbf{P}$ stands for the same formula in the premise and conclusion. Universal quantification is essential to soundness of G because, in general, a formula $\mathbf{P}$ could be true initially and become false after running $\mathbf{a}$. That is, the formula $\mathbf{P}\rightarrow[\mathbf{a}]\mathbf{P}$ is not a valid formula of dL. Soundness of rule G relies crucially on the assumption that $\mathbf{P}$ is true in *every* state and thus in every final state of program $\mathbf{a}$.

Monotonicity M says valid implications can be moved under modalities. In KeYmaera X, both rule G and rule M are primarily used internally as generalization principles inside some other high-level user-facing proof tactic. Likewise, in paper proofs, rules G and M are often not written explicitly but are used to derive high-level versions of other dL reasoning principles. Rules CE and CQ say that valid equalities and equivalences can be applied in context, and in practice are mostly used for performance.

Modus ponens is conspicuously absent: while modus ponens can be expressed as an axiomatic rule, it can be (and in practice, is) derived from the dL sequent rules. Like the axiomatic rules, the sequent rules could be formulated concretely in principle, but operate on sequents rather than formulas. A rule application specifies which formula a sequent rule is applied to, and we formulate sequent rules as schemata so they can be applied to arbitrary formulas from the proof context rather than the formula at a fixed position. Propositional (classical) sequent calculus is standard, so we give only a few examples here. In classical sequent calculus, the antecedent and succedent are both contexts, and the sequent $\Gamma\vdash\Delta$ is equivalent to the formula $\bigwedge\Gamma\rightarrow\bigvee\Delta$.

$$(\rightarrow\text{L})\quad \frac{\Gamma\vdash\Delta,\phi\quad\psi,\Gamma\vdash\Delta}{\phi\rightarrow\psi,\Gamma\vdash\Delta} \qquad (\wedge\text{R})\quad \frac{\Gamma\vdash\phi,\Delta\quad\Gamma\vdash\psi,\Delta}{\Gamma\vdash\phi\wedge\psi,\Delta}$$

$$(\rightarrow\text{R})\quad \frac{\Gamma,\phi\vdash\Delta,\psi}{\Gamma\vdash\phi\rightarrow\psi,\Delta} \qquad (\wedge\text{L})\quad \frac{\Gamma,\phi,\psi\vdash\Delta}{\phi\wedge\psi,\Gamma\vdash\Delta}$$

Figure 2.3: Selected classical sequent calculus rules.

For example, implications are proved on the left (rule $\rightarrow$L) by proving the assumption in order to gain the conclusion, or proved on the right (rule $\rightarrow$R) by assuming the assumption to prove the conclusion. Conjunctions are proved on the left (rule $\wedge$L) by decomposing the assumed conjunction into two assumptions, or proved on the right (rule $\wedge$R) by proving each conjunct. The sequent rules are widely used both for propositional reasoning in practical proofs and to discharge assumptions when applying an axiom. The implementation

of the sequent rules is concise, comprising only 212 lines of the KeYmaera X core (Mitsch & Platzer, 2020). Nonetheless, the soundness proofs for sequent rules require care (Section 2.8). For example, the sequent calculus rules allow random access to an antecedent or succedent represented as an ordered list, so the soundness proofs must carefully capture the impact of list operations on the semantics of a sequent.

2.6.1 Renaming Rules

The *uniform renaming* rule says that valid formulas and sequents remain valid when renaming a variable x to a variable y and vice versa. That is, we rename by swapping (or *transposing*) the names of variables, so that uniform renaming is sound with no side conditions. The notation $e\frac{y}{x}$ represents expression e with variables x and y swapped. We also support *bound renaming*, analogous to α-conversion, which renames a variable binder (assignment) and the bound occurrences. The idea of bound renaming can be generalized to any binder, but is most often used for assignments in practice. Note that the renaming rules use Greek variables because they are rule schemata which take expressions as arguments, rather than axiomatic rules defined by concrete formulas. In contrast to uniform

$$\text{(UR)}\quad \frac{\phi}{\phi\frac{y}{x}} \qquad\qquad \text{(BR)}\quad \frac{\Gamma \vdash [x := \theta]\phi, \Delta}{\Gamma \vdash [y := \theta]\phi\frac{y}{x}, \Delta}\ {}^{1}$$

[1] $\{y, y', x'\} \cap \mathrm{FV}([x := \theta]\phi) = \emptyset$

Figure 2.4: Renaming rules.

renaming, bound renaming has a soundness side condition expressed in terms of the free variables $\mathrm{FV}(\phi)$ of the postcondition ϕ. Notably, a soundness bug in the implementation of the bound renaming rule in KeYmaera X was discovered and fixed in the process of formalization (Bohrer et al., 2017). We discuss the bug and the fix in Section 2.10.

2.6.2 Substitution Rule

The uniform substitution (rule US) rule is the heart of the dL proof calculus. We first describe the substitution rule and its side conditions at a high level, then give its non-mechanized definition and finally the formal Isabelle/HOL definition. We begin with a brief description because the definitions are long and the formalization longer still.

Uniform substitution is most easily understood as it applies to formulas: a valid formula remains valid under any admissible substitution σ (rule US), where the substitution σ is a partial mapping from uniform substitution symbols to their replacements. The substitution algorithm works by structural recursion on expressions. We write $\sigma(e)$ with parentheses for the result of applying the substitution σ uniformly throughout e. Differential program symbols ($\mathbf{c}\{|\text{space}|\}$) are replaced with differential programs, program symbols $\mathbf{a}$ are replaced with programs, predicates and predicationals $\mathbf{p}(\theta_1, \ldots, \theta_n)$ and $\mathbf{C}(\phi)$ are replaced with formulas, and functions and functionals $\mathbf{f}(\theta_1, \ldots, \theta_n)$ and $\mathbf{F}$ are replaced with

terms. We write $\sigma\,\mathbf{f}$ without parentheses for the term that replaces function $\mathbf{f}$ in σ, likewise for other uniform substitution symbols. We write $Dom(\sigma)$ for the set of replaced symbols.

$$\text{(US)}\quad \frac{\phi}{\sigma(\phi)} \quad \text{where } \sigma \text{ is admissible for } \phi$$

In short, the substitution of σ in ϕ is admissible if for every symbol $\mathbf{f}$ (likewise $\mathbf{p}$) that gets replaced in $\sigma(\phi)$, the replacement $\sigma(\mathbf{f})$ does not introduce new free references to variables that are bound in the surrounding context. Because functionals, predicationals, and program symbols already treat every variable as free, substituting them would never introduce a *new* free reference. Admissibility is implemented as a recursive syntactic check.

Recall that a proof state consists of open goals and a conclusion (a derived axiomatic rule), thus it is useful to apply substitution to entire rules (Platzer, 2017a, Thm. 27).

Lemma 2.1 (Rule substitution)**.** *If a rule* $\dfrac{\phi_1 \cdots \phi_n}{\psi}$ *is locally sound and* $\mathrm{FV}(\sigma) = \emptyset$ *then rule* $\dfrac{\sigma(\phi_1) \cdots \sigma(\phi_n)}{\sigma(\psi)}$ *is locally sound.*

For example, axiomatic rules are applied to a premise by first applying Lemma 2.1 to the axiomatic rule, then applying the resulting rule to the premise.

Substitution and Admissibility Definitions. We present the substitution algorithm (based on (Platzer, 2017a)) in Fig. 2.5 with inline admissibility checking at each recursive call. We call each step of the admissibility check a U-admissibility check: when U is a set of variables, the substitution σ is U-admissible for expression e if applying the substitution σ to e introduces no free references to any variable $x \in U$. Every time we substitute inside a connective that binds variables, we let U be the set of variables bound by the connective. Thus, the sum total of the U-admissibility checks amounts to checking that no substitution ever introduces a free reference to a program variable under a binder of that variable, which is what admissibility requires to ensure sound substitutions.

The $\exists x\, \phi$ case demonstrates an admissibility check in its purest form: the formula ϕ appears under a binder of x, thus the substitution must introduce no new free reference to x in ϕ. In the sequential composition $\alpha;\beta$, program β is executed after α, thus it is "under the binders" of α. For that reason, the $\alpha;\beta$ case checks that σ is $\mathrm{BV}(\sigma(\alpha))$-admissible for β, meaning that *no* bound variable of (the result of substituting in) α may have a free reference introduced when substituting in β. The cases for α^* and $\langle\alpha\rangle\phi$ reflect the same intuition: in the former, α may be run repeatedly in sequence, while in the latter case ϕ is under all binders of α because it is a *post*condition. The ODE case follows the same intuition as a loop which repeats continuously often, i.e., the ODE checks $\mathrm{BV}(ODE)$-admissibility[12] because the vector field of an ODE depends on the state, which in turn depends on the past evolution of the ODE. The $(\theta)'$ case has a very strict admissibility

[12]We check $\mathrm{BV}(ODE)$-admissibility for consistency with the Isabelle/HOL formalization, though a tighter check for $\mathrm{BV}(\sigma(ODE))$-admissibility would also be sound. While the tighter condition is useful both in a theoretical development and to ensure generality of the implementation, it did not arise in the experiments discussed in this chapter.

Case	Replacement	Admissible when:
$\sigma(q)$	$= q$	
$\sigma(x)$	$= x$	
$\sigma((\theta)')$	$= (\sigma(\theta))'$	σ $\mathcal{V} \cup \mathcal{V}'$-adm. in θ
$\sigma(\theta + \eta)$	$= \sigma(\theta) + \sigma(\eta)$	
$\sigma(\theta \cdot \eta)$	$= \sigma(\theta) \cdot \sigma(\eta)$	
$\sigma(\theta / \eta)$	$= \sigma(\theta) / \sigma(\eta)$	
$\sigma(\theta - \eta)$	$= \sigma(\theta) - \sigma(\eta)$	
$\sigma(\max(\theta, \eta))$	$= \max(\sigma(\theta), \sigma(\eta))$	
$\sigma(\min(\theta, \eta))$	$= \min(\sigma(\theta), \sigma(\eta))$	
$\sigma(\mathsf{abs}(\theta))$	$= \mathsf{abs}(\sigma(\theta))$	
$\sigma(\mathbf{f}(\vec{\theta}))$	$= \{\overrightarrow{\cdot_i \mapsto \sigma(\theta_i)}\}(\sigma\mathbf{f}), \mathbf{f} \in Dom(\sigma)$, else $\mathbf{f}(\overrightarrow{\sigma(\theta)})$	
$\sigma(\mathbf{F})$	$= \sigma\mathbf{F}, \mathbf{F} \in Dom(\sigma)$, else $\mathbf{F}$	
$\sigma(\mathbf{c}\{\|!\mathrm{x}\|\})$	$= \sigma\,\mathbf{c}\{\|!\mathrm{x}\|\}, \mathbf{c}\{\|!\mathrm{x}\|\} \in Dom(\sigma)$, else $\mathbf{c}\{\|!\mathrm{x}\|\}$	$x \notin \mathrm{BV}(\sigma\,\mathbf{c})$
$\sigma(\mathbf{c})$	$= \sigma\,\mathbf{c}, \mathbf{c} \in Dom(\sigma)$, else $\mathbf{c}$	
$\sigma(ODE1, ODE2)$	$= \sigma(ODE1), \sigma(ODE2)$	$\mathrm{BV}(ODE1) \cap \mathrm{BV}(ODE2) = \emptyset$
$\sigma(\{x' = \theta\})$	$= \{x' = \sigma(\theta)\}$	
$\sigma(\mathbf{a})$	$= \sigma\mathbf{a}, \mathbf{a} \in Dom(\sigma)$, else $\mathbf{a}$	
$\sigma(x := \theta)$	$= x := \sigma(\theta)$	
$\sigma(x := *)$	$= x := *$	
$\sigma(?(\phi))$	$= ?(\sigma(\phi))$	
$\sigma(\{ODE \,\&\, \psi\})$	$= \{x' = \sigma(\theta) \,\&\, \sigma(\psi)\}$	σ $\mathrm{BV}(ODE)$-adm. in ODE, ψ
$\sigma(\alpha; \beta)$	$= \sigma(\alpha); \sigma(\beta)$	σ $\mathrm{BV}(\sigma(\alpha))$-adm. in β
$\sigma(\alpha \cup \beta)$	$= \sigma(\alpha) \cup \sigma(\beta)$	
$\sigma(\alpha^*)$	$= \sigma(\alpha)^*$	σ $\mathrm{BV}(\sigma(\alpha))$-adm. in α
$\sigma(\theta \geq \eta)$	$= \sigma(\theta) \geq \sigma(\eta)$	
$\sigma(\mathbf{p}(\vec{\theta}))$	$= \{\overrightarrow{\theta_i \mapsto \sigma(\theta_i)}\}(\sigma\mathbf{p}), \mathbf{p} \in Dom(\sigma)$,else $\mathbf{p}(\overrightarrow{\sigma(\theta)})$	
$\sigma(\mathbf{P})$	$= (\sigma\mathbf{P}), \mathbf{P} \in Dom(\sigma)$, else $\mathbf{P}$	
$\sigma(\neg\phi)$	$= \neg\sigma(\phi)$	
$\sigma(\phi \wedge \psi)$	$= \sigma(\phi) \wedge \sigma(\psi)$	
$\sigma(\exists x\, \phi)$	$= \exists x\, \sigma(\phi)$	σ $\{x\}$-adm. in ϕ
$\sigma(\langle\alpha\rangle\phi)$	$= \langle\sigma(\alpha)\rangle\sigma(\phi)$	σ $\mathrm{BV}(\sigma(\alpha))$-adm. in ϕ

Figure 2.5: Uniform substitution algorithm.

condition: no free reference to any variable can be introduced. Intuitively, every variable is bound in $(\theta)'$ because its semantics is defined as a differential. The differential of a function at a state is determined by finding the partial derivatives as the state varies in *any* dimension.

In contrast to the other cases, the function and predicate cases require a more careful totality argument to reason about the substitution of concrete arguments for parameters of functions and predicates. We call the argument substitution a *first-order* substitution whereas general substitutions σ are called *second-order* substitutions. In the cases for functions and predicates, $\vec{\theta}$ is the vector of arguments, $\overrightarrow{\sigma(\theta)}$ is the vector of second-order substitution results from applying σ to the arguments, and $\{\overrightarrow{\cdot_i \mapsto \sigma(\theta_i)}\}$ is a first-order substitution which maps each distinguished function argument symbol $\cdot_i$ to the corresponding $\sigma(\theta_i)$. Taking the function case as an example, the recursive call $\{\overrightarrow{\cdot_i \mapsto \sigma(\theta_i)}\}(\sigma\mathbf{f})$ features a term $\sigma\,\mathbf{f}$ which may be larger than $\mathbf{f}(\vec{\theta})$. The substitution function is total: even if $\sigma\mathbf{f}$ is larger than $\mathbf{f}(\vec{\theta})$, the substitution has become simpler because first-order substitution $\{\overrightarrow{\cdot_i \mapsto \sigma(\theta_i)}\}$ only specifies replacements of nullary function symbols $\cdot_i$, but σ must have featured a function $\mathbf{f}$ which had arguments, else we would not have entered the function argument substitution case. By definition of a nullary function symbol, the recursive call on $\{\overrightarrow{\cdot_i \mapsto \sigma(\theta_i)}\}(\sigma\mathbf{f})$ will never recursively re-enter the function argument substitution case.

Our use of the phrases *second-order* and *first-order* substitution is inspired by substitution in NuPRL (Constable et al., 1986; Anand & Rahli, 2014; Rahli & Bickford, 2016). Second-order substitution functions are commonly organized in two ways:

- In the *second-order substitution* approach, there is a single, second-order substitution function, which handles all substitutions, including argument substitutions. In this approach, inductive proofs about substitution must employ a lexicographic complexity metric: substitution complexity followed by expression complexity.
- In the *first-order substitution* approach, a separate helper function handles first-order substitution and is called by the main second-order substitution function. In this approach, simple induction over expressions can be used. In Isabelle/HOL, no special metric is needed for such inductive principles. The downside is that multiple substitution functions must be written, each with their own theorem statements, admissibility conditions, and proofs.

The second-order approach is used in Platzer's dGL formalization (Platzer, 2019a), in the original paper description of the dL uniform substitution calculus (Platzer, 2017a, Lem. 23-25), and in KeYmaera X to reduce the number of function definitions and lemmas. The presentation in Fig. 2.5 implies a second-order approach, but can be read as a first-order approach if one assumes that the function argument substitutions $\{\overrightarrow{\cdot_i \mapsto \sigma(\theta_i)}\}(\sigma\mathbf{f})$ are performed by a separate helper function. The formalization in this chapter uses the first-order approach so that simple induction techniques suffice. We will see that the increase in the number of substitution functions and lemmas may outweigh the benefit of enabling simple induction principles, but at the very least our formalization provides concrete evidence of the cost of the first-order approach.

2.6.3 Isabelle/HOL Formalization

We give the Isabelle/HOL function definitions for the substitution algorithm first, followed by the admissibility predicates. In contrast to the paper presentation, we perform the admissibility checks out-of-line with the substitution. Both approaches are equally viable; we chose our approach in hopes of avoiding excessive case analyses in proofs about substitution, while the inline admissibility checks of the paper likely generate more efficient code. While admissibility is fundamental to soundness of substitution, we present the substitution formalization first, primarily since the Isabelle/HOL formalization of admissibility requires a large number of incidental helper predicates, which may be more understandable after reading the corresponding substitution helper functions.

Substitution functions are separated by syntactic class and separated between second-order and first-order. The second-order substitution functions are `Fsubst`, `Psubst`, `Osubst`, and `Tsubst`, which respectively implement substitution of formulas, programs, ODEs, and terms. The entry point is `Fsubst`, which is used in rule US. The first-order functions `FsubstFO`, `PsubstFO`, `OsubstFO`, and `TsubstFO` perform first-order replacement of function or predicate arguments in formulas, programs, ODEs, and terms, respectively. The first-order substitution of *predicational* arguments is implemented by `PPsubst` and `PFsubst` for programs and formulas, respectively. We give the second-order functions first.

The Isabelle/HOL type of substitutions is a record whose fields `SFunctions`, `SFunls`, `SPredicates`, `SContexts`, `SPrograms`, and `SODEs` specify replacements of functions, functionals, predicates, predicationals, programs, and ODEs, respectively. The well-formedness predicate for substitutions is named `ssafe σ`. It checks that every right-hand side (i.e., every expression replacement) in σ satisfies the well-formedness predicate for its syntactic class. In contrast to expressions elsewhere, replacements of functions, predicates, and predicationals in a substitution can mention reserved identifiers which represent the arguments of functions, predicates, and predicationals. As is standard in Isabelle/HOL, we access the fields using function application syntax.

```
primrec Tsubst::"trm ⇒ subst ⇒ trm"
where
  "Tsubst (Var x) σ = Var x"
| "Tsubst (DiffVar x) σ = DiffVar x"
| "Tsubst (Const r) σ = Const r"
| "Tsubst (Function f args) σ =
  (case SFunctions σ f of Some f' ⇒ TsubstFO f' | None ⇒ Function f)
  (λ i. Tsubst (args i) σ)"
| "Tsubst (Functional f) σ =
  (case SFunls σ f of Some f' ⇒  f' | None ⇒ Functional f)"
| "Tsubst (Neg θ1) σ = Neg (Tsubst θ1 σ)"
| "Tsubst (Plus θ1 θ2) σ = Plus (Tsubst θ1 σ) (Tsubst θ2 σ)"
| "Tsubst (Times θ1 θ2) σ = Times (Tsubst θ1 σ) (Tsubst θ2 σ)"
| "Tsubst (Div θ1 θ2) σ = Div (Tsubst θ1 σ) (Tsubst θ2 σ)"
| "Tsubst (Max θ1 θ2) σ = Max (Tsubst θ1 σ) (Tsubst θ2 σ)"
| "Tsubst (Min θ1 θ2) σ = Min (Tsubst θ1 σ) (Tsubst θ2 σ)"
```

```
| "Tsubst (Abs θ1) σ = Abs (Tsubst θ1 σ)"
| "Tsubst (Differential θ) σ = Differential (Tsubst θ σ)"
```

The function and functional cases check whether the given uniform substitution symbol is replaced by the substitution. If a functional symbol is found which has a replacement in the substitution, the replacement is immediately returned. If a function is found which has a replacement, we apply second-order substitution to each argument, then apply first-order substitution to replace the arguments within the replacement of the function symbol. The anonymous function `(λ i. Tsubst (args i) σ)` is a first-order substitution (data structure); as we will see in the first-order substitution algorithm, we represent first-order substitutions as total functions from identifiers to replacements. We do so throughout the formalization. The other cases for composite terms apply substitution homomorphically to subterms, while the base cases return the input.

Function `Osubst` substitutes in ODEs.

```
primrec Osubst::"ODE ⇒ subst ⇒ ODE"
where
  "Osubst (OVar c sp) σ =
    (case SODEs σ c sp of Some c' ⇒ c' | None ⇒ OVar c)"
| "Osubst (OSing x θ) σ = OSing x (Tsubst θ σ)"
| "Osubst (OProd ODE1 ODE2) σ = OProd (Osubst ODE1 σ) (Osubst ODE2 σ)"
```

The `OVar c sp` case checks whether the substitution defines a replacement for the given differential program symbol and returns the replacement if there is one. Note that the substitution treats `OVar c sp1` and `OVar c sp2` as entirely independent symbols when `sp1` differs from `sp2`. We do not recommend depending on this behavior, which was added only because it was easy to do so.

The substitution functions for programs and formulas are mutually recursive.

```
fun Psubst::"hp ⇒ subst ⇒ hp"
and Fsubst::"formula ⇒ subst ⇒ formula"
where
  "Psubst (Pvar a) σ =
    (case SPrograms σ a of Some a' ⇒ a' | None ⇒ Pvar a)"
| "Psubst (Assign x θ) σ = Assign x (Tsubst θ σ)"
| "Psubst (AssignAny x) σ = AssignAny x"
| "Psubst (DiffAssign x θ) σ = DiffAssign x (Tsubst θ σ)"
| "Psubst (Test ψ) σ = Test (Fsubst ψ σ)"
| "Psubst (EvolveODE ODE ψ) σ = EvolveODE (Osubst ODE σ) (Fsubst ψ σ)"
| "Psubst (Choice α β) σ = Choice (Psubst α σ) (Psubst β σ)"
| "Psubst (Sequence α β) σ = Sequence (Psubst α σ) (Psubst β σ)"
| "Psubst (Loop α) σ = Loop (Psubst α σ)"

| "Fsubst (Geq θ1 θ2) σ = Geq (Tsubst θ1 σ) (Tsubst θ2 σ)"
| "Fsubst (Prop p args) σ =
  (case SPredicates σ p of
    Some p' ⇒ FsubstFO p' (λi. Tsubst (args i) σ)
```

```
  | None ⇒ Prop p (λi. Tsubst (args i) σ))"
| "Fsubst (Not ψ) σ = Not (Fsubst ψ σ)"
| "Fsubst (And ψ ψ) σ = And (Fsubst ψ σ) (Fsubst ψ σ)"
| "Fsubst (Exists x ψ) σ = Exists x (Fsubst ψ σ)"
| "Fsubst (Diamond α ψ) σ = Diamond (Psubst α σ) (Fsubst ψ σ)"
| "Fsubst (InContext C ψ) σ =
  (case SContexts σ C of Some C' ⇒ PFsubst C' (λ _. (Fsubst ψ σ))
  | None ⇒ InContext C (Fsubst ψ σ))"
```

Every constituent formula, program, ODE, and term has the corresponding substitution function applied. The program constant case returns the replacement of the program symbol, if any. When the predicate case finds a replacement for a given predicate, first-order substitution is used to replace the arguments in the replacement of the predicate symbol. Note that the `FsubstFO` function used in the predicate case belongs to the same family of functions as the `TsubstFO` function used in the function symbol case, which serves as a helper function to it. That is, both functions replace term arguments in an expression. In contrast, the `InContext` case uses `PFsubst` for first-order replacement of a *formula* argument in a formula. Recall that every predicational in the core grammar is unary. The first-order substitution `(λ _. (Fsubst ψ σ))` is expressed as a function for the sake of symmetry with first-order term substitutions, but the domain of a first-order substitution data structure for a unary symbol is the unit type, meaning the argument can safely be ignored with a wildcard pattern `_`.

We give a select subset of the first-order substitution functions. The `TsubstFO` function performs replacement of function arguments in terms.

```
primrec TsubstFO::"trm ⇒ (id ⇒ trm) ⇒ trm"
where
  "TsubstFO (Var v) σ = Var v"
| "TsubstFO (DiffVar v) σ = DiffVar v"
| "TsubstFO (Const r) σ = Const r"
| "TsubstFO (Function f args) σ =
    (case (as_reserved_or_other f)
      Inl reserved ⇒ σ reserved
    | Inr other ⇒ Function other (λ i. TsubstFO (args i) σ))"
| "TsubstFO (Functional f) σ =
    (case (as_reserved_or_other f) of
       Inl reserved ⇒  σ reserved
     | Inr other ⇒ (Functional other))"
| "TsubstFO (Plus θ1 θ2) σ = Plus (TsubstFO θ1 σ) (TsubstFO θ2 σ)"
| "TsubstFO (Times θ1 θ2) σ = Times (TsubstFO θ1 σ) (TsubstFO θ2 σ)"
| "TsubstFO (Div θ1 θ2) σ = Div (TsubstFO θ1 σ) (TsubstFO θ2 σ)"
| "TsubstFO (Max θ1 θ2) σ = Max (TsubstFO θ1 σ) (TsubstFO θ2 σ)"
| "TsubstFO (Min θ1 θ2) σ = Min (TsubstFO θ1 σ) (TsubstFO θ2 σ)"
| "TsubstFO (Abs θ) σ = Abs (TsubstFO θ σ)"
| "TsubstFO (Differential θ) σ = Differential (TsubstFO θ σ)"
```

The crucial case is the `Function` case, which checks whether the given function symbol is a reserved function argument symbol, and replaces it with the replacement specified by the first-order substitution, if so. The development version of the formalization uses sigil characters to distinguish arguments from non-arguments. The function `as_reserved_or_other`[13] parses an identifier and handles the sigil character or lack thereof. Platzer's dGL formalization (Platzer, 2019a) and previous paper presentations (Platzer, 2019a) take an analogous approach, but only support unary functions and thus have a single reserved identifier. In contrast, the AFP version (Bohrer, 2017) treats the identifier type as type variable which locally assumes a disjoint union type within the substitution algorithm. In the AFP version, the function `as_reserved_or_other` can be eliminated, because the substitution already distinguishes reserved and unreserved symbols as left and right injections, respectively. The `Functional` case is written by analogy to the `Function` case, though its `reserved` case is dead code because function arguments are never functionals.

The definitions of the `PsubstFO` and `FsubstFO` functions are not listed here because each function simply applies `TsubstFO` to every constituent term.

The mutually recursive functions `PPsubst` and `PFsubst` perform first-order substitution of formulas into programs and formulas.

```
fun PPsubst::"hp ⇒ (id ⇒ formula) ⇒ hp"
and PFsubst::"formula ⇒ (id ⇒ formula) ⇒ formula"
where
  "PPsubst (Pvar a) σ = Pvar a"
| "PPsubst (Assign x θ) σ = Assign x θ"
| "PPsubst (DiffAssign x θ) σ = DiffAssign x θ"
| "PPsubst (AssignAny x) σ = AssignAny x"
| "PPsubst (Test ψ) σ = Test (PFsubst ψ σ)"
| "PPsubst (EvolveODE ODE ψ) σ = EvolveODE ODE (PFsubst ψ σ)"
| "PPsubst (Choice α β) σ = Choice (PPsubst α σ) (PPsubst β σ)"
| "PPsubst (Sequence α β) σ = Sequence (PPsubst α σ) (PPsubst β σ)"
| "PPsubst (Loop α) σ = Loop (PPsubst α σ)"

| "PFsubst (Geq θ1 θ2) σ = (Geq θ1 θ2)"
| "PFsubst (Prop p args) σ = Prop p args"
| "PFsubst (Not ψ) σ = Not (PFsubst ψ σ)"
| "PFsubst (And ψ ψ) σ = And (PFsubst ψ σ) (PFsubst ψ σ)"
| "PFsubst (Exists x ψ) σ = Exists x (PFsubst ψ σ)"
| "PFsubst (Diamond α ψ) σ = Diamond (PPsubst α σ) (PFsubst ψ σ)"
| "PFsubst (InContext C ψ) σ =
  (case as_reserved_formula_or_other of
    Inl C' ⇒ InContext C' (PFsubst ψ σ) | Inr p' ⇒ σ p')"
```

Substitutions are performed in the case `InContext`. As with first-order substitutions, the function `as_reserved_formula_or_other` stands for a helper function that distinguishes the reserved argument symbol from other symbols. In the AFP version, symbols

[13]The function has been renamed in this chapter for the sake of readability.

are already distinguished as disjoint unions, but the repository version distinguishes them using identifier sigils.

The other cases map through. This completes the substitution function listings.

We now describe the (extensive list of admissibility) predicates which capture the notions of U-admissibility, first order substitution admissibility, and second-order substitution admissibility. We present a representative subset of them here.

The definitions `TUadmit`, `PUadmit`, and `Fadmit` define U-admissibility for terms, programs, and formulas respectively. There is not a separate U-admissibility definition for ODEs, because all necessary checks are captured in the main ODE admissibility predicate. In the Isabelle/HOL formalization, `SDom σ` implements $Dom(\sigma)$ and `SFV σ i` represents the subset of $\mathrm{FV}(\sigma)$ containing free variables introduced by $\sigma(i)$ for identifier i. That is, the function `FVS` which implements $\mathrm{FV}(\sigma)$ is defined as a union over `SFV`:

```
definition FVS::"subst ⇒ (ident + ident) set"
where "FVS σ = (Ui. SFV σ i)"
```

As usual, the sum `ident + ident` distinguishes base and differential variables.

```
definition TUadmit::"subst ⇒ trm ⇒ (ident + ident) set ⇒ bool"
where "TUadmit σ θ U ↔
  ((Ui ∈ SIGT θ.
    (case SFunctions σ i of Some f' ⇒ FVT f' | None ⇒ {})) ∩ U) = {}"

definition PUadmit::"subst ⇒ hp ⇒ (ident + ident) set ⇒ bool"
where "PUadmit σ θ U ↔ ((Ui ∈ (SDom σ ∩ SIGP θ). SFV σ i) ∩ U) = {}"

definition FUadmit::"subst ⇒ formula ⇒ (ident + ident) set ⇒ bool"
where "FUadmit σ θ U ↔ ((Ui ∈ (SDom σ ∩ SIGF θ). SFV σ i) ∩ U) = {}"
```

A family of predicates including `TadmitFO` ensure first-order substitution admissibility for terms and other classes, respectively. A first-order substitution is modeled by type `ident ⇒ trm` which replaces every argument function symbol with an argument term. The only case with a U-admissibility check is `TadmitFO_Diff`. The remaining cases just inductively check first-order admissibility of subterms.

```
inductive TadmitFO::"(ident ⇒ trm) ⇒ trm ⇒ bool"
  TadmitFO_Diff:"TadmitFFO σ θ ⇒ NTUadmit σ θ UNIV ⇒
    dfree (TsubstFO θ σ) ⇒ TadmitFO σ (Differential θ)"
| ...
```

The `TadmitFO_Diff` case has several helper functions. The checks `TadmitFFO σ θ` and `dfree (TsubstFO θ σ)` are redundant with each other and one of them could be safely removed, but they provide an opportunity to compare different ways of ensuring that substitution preserves various syntactic constraints. Recall that differentials must not be nested in our formalization of dL. While the substitution algorithm assumes that the input is well-formed and thus free of nested differentials, we must ensure through the admissibility checks that substitution does not introduce nested differentials where there were none before. One approach (`TadmitFFO σ θ`) instruments the admissibility predicate to

check that whenever a function argument symbol is mentioned in θ, the replacement of the argument in σ is simple. The second approach (`dfree (TsubstFO θ σ)`) just substitutes σ in θ and checks that the result is simple. Code generated from the latter could be slower in theory if substitution greatly increases the size of a term, but the former is likely slower in practice due to our naïve representation of substitutions. The helper `NTUadmit` says that no replacement which occurs under a differential symbol can introduce any free variables, which is required for soundness in both first-order and second-order substitution. We do not give definitions for `TadmitFFO` and `NTUadmit` because they provide little new insight but do contain boilerplate for managing identifiers which could distract the reader.

The second-order term substitution admissibility predicate `Tadmit` uses a helper predicate `TUadmit` to check that no free variables are introduced under differentials and that all first-order admissibility checks succeed for all function applications. The fields `SFunctions` and `SFunls` express the replacements of functions and functionals in substitution σ, respectively. The function cases `TadmitF_Fun1` and `TadmitF_Fun2` respectively check admissibility of functions that are or are not replaced by the substitution. Both function cases include conditions that are best understood as well-formedness conditions: only unreserved symbols can be replaced by a *second-order* substitution, and their length must be *strictly less* than the global maximum `MAX_STR`, because proofs of identifier lemmas are easier if we assume there is space left for a sigil.

```
definition TUadmit::"subst ⇒ trm ⇒ (id + id) set ⇒ bool"
where "TUadmit σ θ U ↔ ((⋃ i ∈ SIGT θ.
  (case SFunctions σ i of Some f' ⇒ FVT f' | None ⇒ {})) ∩ U) = {}"

inductive Tadmit::"subst ⇒ trm ⇒ bool"
where
  Tadmit_Diff:
    "Tadmit σ θ ⇒ TUadmit σ θ UNIV ⇒ Tadmit σ (Differential θ)"
| TadmitF_Fun1:"(∀i. TadmitF σ (args i)) → SFunctions σ f = Some f' →
   is_not_reserved f → ilength f < MAX_STR →
   (∀i. dfree(Tsubst(args i)σ))→TadmitFFO(λ i. Tsubst(args i) σ) f' →
    TadmitF σ (Function f args)"
| TadmitF_Fun2:"(∀i. TadmitF σ (args i)) → SFunctions σ f = None →
  is_not_reserved f → ilength f < MAX_STR →
  TadmitF σ (Function f args)"
| Tadmit_Funl:"SFunls σ f = Some f' ⇒ Tadmit σ f'
  ⇒ Tadmit σ (Functional f)"
| Tadmit_Plus:"Tadmit σ θ1 ⇒ Tadmit σ θ2 ⇒ Tadmit σ (Plus θ1 θ2)"
| Tadmit_Times:"Tadmit σ θ1 ⇒ Tadmit σ θ2 ⇒ Tadmit σ (Times θ1 θ2)"
| Tadmit_Div:"Tadmit σ θ1 ⇒ Tadmit σ θ2 ⇒ Tadmit σ (Div θ1 θ2)"
| Tadmit_Max:"Tadmit σ θ1 ⇒ Tadmit σ θ2 ⇒ Tadmit σ (Max θ1 θ2)"
| Tadmit_Min:"Tadmit σ θ1 ⇒ Tadmit σ θ2 ⇒ Tadmit σ (Min θ1 θ2)"
| Tadmit_Abs:"Tadmit σ θ1 ⇒ Tadmit σ (Abs θ1)"
| Tadmit_DiffVar:"Tadmit σ (DiffVar x)"
| Tadmit_Var:"Tadmit σ (Var x)"
| Tadmit_Const:"Tadmit σ (Const r)"
```

ODE admissibility is defined as an inductive predicate `Oadmit`. Helper `TadmitF` treats second-order substitutions whose results must be simple, thus it is similar to the first-order helper `TadmitFFO`. Field `SODEs` specifies the differential program symbol substitutions provided by substitution σ.

```
inductive Oadmit::"subst ⇒ ODE ⇒ (id + id) set ⇒ bool"
where
  Oadmit_Var:"Oadmit σ (OVar c) U"
| Oadmit_VarNB:
  "(case SODEs σ c (Some x) of
    Some ode ⇒ Inl x ∉ BVO ode
  | None ⇒ False) ⇒ Oadmit σ (OVar c (Some x)) U"
| Oadmit_Sing:"TUadmit σ θ U ⇒ TadmitF σ θ ⇒ Oadmit σ (OSing x θ) U"
| Oadmit_Prod:"Oadmit σ ODE1 U ⇒ Oadmit σ ODE2 U ⇒
  ODE_dom (Osubst ODE1 σ) ∩ ODE_dom (Osubst ODE2 σ) = {} ⇒
  Oadmit σ (OProd ODE1 ODE2) U"
```

In contrast to the term admissibility predicate, `Oadmit` is parameterized by the set U of bound variables of the ODE: because all equations of an ODE system evolve in parallel, every equation must respect the bound variable restrictions imposed by the others, the result being that a single set U of bound variables can be used when checking the entire system. The `Oadmit_Var` case says that differential program constants with no space specifiers are always admissible. The `Oadmit_VarNB` case for differential program constants with space specifiers assumes that the constant is replaced[14] and requires that the taboo variable `x` is not a bound variable of the replacement of $\mathbf{c}\{|!x|\}$. The `Oadmit_Prod` case checks admissibility of each component of a product system and also checks that no variable is bound twice in the same system.

We give the main admissibility predicates for programs and formulas, only mentioning the helpers briefly because their design is similar to those for terms. The helpers `PUadmit` and `FUadmit` implement U-admissibility for programs and formulas. Helper `PFadmit` is admissibility for the first-order substitution of formulas into formulas and `NFadmit` is admissibility for first-order substitution of terms into formulas. Fields `SPredicates` and `SContexts` contain the replacements for predicates and predicationals. Predicates `hpsafe`, `fsafe`, and `dsafe` are the standard well-formedness predicates for hybrid programs, formulas, and terms. In each case, we apply admissibility checks inductively, check well-formedness of expressions when needed, and apply the same U-admissibility checks specified by the paper presentation of admissibility.

```
inductive Padmit::"subst ⇒ hp ⇒ bool"
and Fadmit::"subst ⇒ formula ⇒ bool"
where
  Padmit_Pvar:"Padmit σ (Pvar a)"
| Padmit_Sequence:"Padmit σ a ⇒ Padmit σ b ⇒
  PUadmit σ b (BVP (Psubst a σ))⇒ hpsafe (Psubst a σ) ⇒
  Padmit σ (Sequence a b)"
```

[14]This condition is incidental to our formalization and thus is not mentioned in the paper presentation of the admissibility condition.

```
| Padmit_Loop:"Padmit σ a ⇒ PUadmit σ a (BVP (Psubst a σ)) ⇒
  hpsafe (Psubst a σ) ⇒ Padmit σ (Loop a)"
| Padmit_ODE:"Oadmit σ ODE (BVO ODE) ⇒ Fadmit σ ψ ⇒
  FUadmit σ ψ (BVO ODE) ⇒ Padmit σ (EvolveODE ODE ψ)"
| Padmit_Choice:"Padmit σ a ⇒ Padmit σ b ⇒ Padmit σ (Choice a b)"
| Padmit_Assign:"Tadmit σ θ ⇒ Padmit σ (Assign x θ)"
| Padmit_AssignAny:"Padmit σ (AssignAny x)"
| Padmit_DiffAssign:"Tadmit σ θ ⇒ Padmit σ (DiffAssign x θ)"
| Padmit_Test:"Fadmit σ ψ ⇒ Padmit σ (Test ψ)"

| Fadmit_Geq:"Tadmit σ θ1 ⇒ Tadmit σ θ2 ⇒ Fadmit σ (Geq θ1 θ2)"
| Fadmit_Prop1:"(∧i. Tadmit σ (args i)) ⇒ SPredicates σ p = Some p' ⇒
  NFadmit (λ i. Tsubst (args i) σ) p' ⇒
  (∧i. dsafe (Tsubst (args i) σ)) ⇒
  Fadmit σ (Prop p args)"
| Fadmit_Prop2:"(∧i. Tadmit σ (args i)) ⇒ SPredicates σ p = None ⇒
  Fadmit σ (Prop p args)"
| Fadmit_Not:"Fadmit σ ψ ⇒ Fadmit σ (Not ψ)"
| Fadmit_And:"Fadmit σ ψ ⇒ Fadmit σ ψ ⇒ Fadmit σ (And ψ ψ)"
| Fadmit_Exists:"Fadmit σ ψ ⇒ FUadmit σ ψ {Inl x} ⇒
  Fadmit σ (Exists x ψ)"
| Fadmit_Diamond:"Fadmit σ ψ ⇒ Padmit σ a ⇒
  FUadmit σ ψ (BVP (Psubst a σ)) ⇒ hpsafe (Psubst a σ) ⇒
  Fadmit σ (Diamond a ψ)"
| Fadmit_Context1:"Fadmit σ ψ ⇒ FUadmit σ ψ UNIV ⇒
  SContexts σ C = Some C' ⇒ PFadmit (λ _. Fsubst ψ σ) C' ⇒
  fsafe(Fsubst ψ σ) ⇒ Fadmit σ (InContext C ψ)"
| Fadmit_Context2:"Fadmit σ ψ ⇒ FUadmit σ ψ UNIV ⇒
  SContexts σ C = None ⇒ Fadmit σ (InContext C ψ)"
```

This completes the listing of admissibility predicates, each of which is used in correctness lemmas for the corresponding substitution functions. The large number of substitution functions and admissibility predicates exposes a major downside of providing separate functions for first-order substitutions and a downside of prohibiting nested differentials: both design choices greatly increase the number of inductive predicate definitions and lemmas about them required. This is a limitation of our approach of separating the definition of admissibility from the definition of substitution.

2.7 Soundness Proof

The soundness proof establishes soundness of individual axioms and axiomatic rules, lemmas about the static semantics and admissibility, and soundness of the renaming and soundness rules. In Section 2.8, the axioms and rules are packaged into a proof term checker and the soundness results of this section are composed to show that the entire checker is sound.

2.7.1 Static Semantics Proofs

The *coincidence* lemma (Platzer, 2017a, Lem. 10-12, Lem. 17) says that states and interpretations which agree on the free variables and signature of a formula will assign it the same truth value. Recall that `fsafe` is a syntactic well-formedness ("safety") condition for formulas and `is_interp` is a semantic well-formedness condition for interpretations. Predicates `Iagree` and `Vagree` check the agreement of two interpretations or two states on a set of identifiers. Agreement properties are crucial for the static semantics lemmas.

```
lemma coincidence_formula:
 "∧ν ν' I J.
  fsafe (ψ::formula) ⇒ is_interp I ⇒ is_interp J ⇒
  Iagree I J (SIGF ψ) ⇒ Vagree ν ν' (FVF ψ) ⇒
  (ν ∈ fml_sem I ψ ↔ ν' ∈ fml_sem J ψ)"
```

The *bound effect* lemma (Platzer, 2017a, Lem. 9, Lem. 17) says that only bound variables can be modified by programs. Here, `hpsafe` is the well-formedness condition on hybrid programs, which ensures that only simple terms are differentiated.

```
lemma bound_effect:
  fixes I::"interp"
  assumes :"is_interp I"
  shows "∧ν::state. ∧ω::state. hpsafe α ⇒ (ν, ω) ∈ prog_sem I α
    ⇒ Vagree ν ω (-(BVP α))"
```

2.7.2 Adjoints and Substitution

The `adjoint` function captures the effect of a substitution as an interpretation, which is called the *adjoint* interpretation. The adjoint lemma (Platzer, 2017a, Cor. 22) says that the semantics of formula ψ is preserved between the adjoint interpretations of σ with respect to two states ω and ν so long as ω and ν differ only on some taboo set U (`Vagree`) such that σ is U-admissible for formula ψ as encoded by predicate `FUadmit` (Section 2.6.3). Here `fsafe` and `ssafe` are well-formedness conditions for formulas and substitutions: formulas should only ever differentiate differentiable terms, while substitutions must replace symbols only with well-formed expressions.

```
lemma uadmit_fml_adjoint:
  assumes "fsafe ψ"
  assumes "ssafe σ"
  assumes "is_interp I"
  assumes "FUadmit σ ψ U"
  assumes "Vagree ν ω (-U)"
  shows "fml_sem (adjoint I σ ν) ψ = fml_sem (adjoint I σ ω) ψ"
```

These results culminate in the *substitution* lemma (Platzer, 2017a, Lem. 23-25), from which soundness (Platzer, 2017a, Lem. 26) of rule US follows directly. The lemma for formula substitution `Fsubst` states that a substituted formula holds in an interpretation when the non-substituted formula holds in the adjoint.

```
lemma subst_fml:
  fixes I::interp and ν::state
  assumes good_interp:"is_interp I"
  assumes Fadmit:"Fadmit σ ψ"
  assumes fsafe:"fsafe ψ"
  assumes ssafe:"ssafe σ"
  shows
    "(ν ∈ fml_sem I (Fsubst ψ σ)) = (ν ∈ fml_sem (adjoint I σ ν) ψ)"
```

Lemma `subst_fml` is proved by induction simultaneously with substitution for programs. In general, every substitution helper function (Section 2.6.3) has its own substitution lemma which assumes the corresponding notion of admissibility and proves that syntactic substitutions have the same effect as adjoint interpretations. Lemmas about first-order substitutions employ a corresponding notion of adjoint interpretations of first-order substitutions.

While the volume of substitution functions, admissibility predicates, and substitution lemmas is large, all lemmas implement the same notion that substitutions have the same effect as adjoints. Our first-order substitution approach accepts a larger implementation in order to ensure simple induction principles suffice to show the substitution lemmas. In each inductive proof, we apply the adjoint and static semantics lemmas to show the claim. The second-order approach taken in Platzer's subsequent dGL formalization (Platzer, 2019a) and elsewhere likely provides a better tradeoff by minimizing the number of functions and lemmas. Nonetheless, it is useful that both formalizations exist so that the cost of a first-order approach can be seen clearly. Moreover, Isabelle/HOL does not care whether a proof is long or short; in either case, the substitution lemmas are true.

The formula substitution lemma is worth the effort invested because it yields soundness of the substitution rule (US) as a corollary. Uniform substitution calculi are designed to offload as much complexity as possible into the substitution rule, so that the rest of the calculus can employ conceptually simple concrete axioms instead of axiom schemata with nontrivial side conditions. Soundness of rule US and validity of the dL axioms amount to the largest share of a comprehensive soundness proof for dL. Having completed the heart of the proof, we turn to packaging the soundness proof in a verified proofchecker, from which executable code can be extracted and used to cross-check real dL proof terms.

2.8 Proof Checker

We define a datatype of dL proof terms. We instrumented (Bohrer et al., 2018) the KeYmaera X prover to record proofs in our proof term format and cross-check them using a generated, verified proofchecker. The formalization of the proofchecker begins by enumerating the names of rules and axioms, with arguments, if any. Composing the axioms and rules will allow us to define a proof term datatype `pt` representing an entire dL proof.

The datatype definitions for sequents, rules, axioms, and proof terms are based on corresponding data structures from the KeYmaera X implementation. As in KeYmaera X,

we operate over sequents rather than formulas. A classical sequent[15] contains a list of antecedent formulas and a list of succedent formulas. As in KeYmaera X, we represent the state of an ongoing proof as a derived rule: every proof state lists the conclusion of the proof and a list of open goals which, if proved, entail validity of the conclusion. A finished proof is simply a derived rule with zero premises, whose conclusion is thus a valid sequent.

```
type_synonym sequent = "formula list * formula list"
type_synonym rule = "sequent list * sequent"
```

The truth value of a sequent $(\Gamma \vdash \Delta)$ is defined as the truth value of the corresponding formula $\bigwedge \Gamma \vdash \bigvee \Delta$:

```
fun seq2fml::"sequent ⇒ formula"
where
"seq2fml (ante,succ) = Implies (foldr And ante TT) (foldr Or succ FF)"

fun seq_sem::"interp ⇒ sequent ⇒ state set"
where "seq_sem I S = fml_sem I (seq2fml S)"
```

The proofchecker maintains the invariant that the current proof state is always *locally* sound, meaning that a common interpretation is used to interpret the uniform substitution symbols of the premises and the conclusion. The Isabelle/HOL predicate `sound` captures local soundness specifically.

```
definition sound::"rule ⇒ bool"
where "sound R ↔ (∀I. is_interp I → (∀i. i ≥ 0 → i < length (fst R) →
    seq_sem I (nth (fst R) i) = UNIV) → seq_sem I (snd R) = UNIV)"

definition seq_valid
where "seq_valid S ≡ ∀I. is_interp I → seq_sem I S = UNIV"
```

Sound derived rules with no premises are valid derived axioms:

```
lemma sound_valid:
  assumes "sound ([], C)"
  shows "seq_valid C"
```

We note several minor differences between our representation and the representation used by KeYmaera X. KeYmaera X provides an object-oriented interface where proof steps are applied by invoking methods of a proof object; our representation is functional. KeYmaera X elides proof terms in the core and provides an optional user-space wrapper around the core which remembers proof terms; our functional checker operates over proof terms. Both we and KeYmaera X represent the built-in proof rules as a datatype, though we factor our definitions differently: KeYmaera X defines rule constructors with positional arguments, while we factor out positioning so that it can be treated only once in the proof.

We now enumerate the datatypes used in our proofchecker. The types `lrule` and `rrule` enumerate the left and right sequent calculus rules, such as first-order reasoning

[15] In contrast to constructive sequents which have a single formula in the succedent

and bound renaming. Type `axRule` enumerates the axiomatic rules and type `axiom` enumerates the axioms. In addition to the core hybrid program axioms, we include several axioms which are derivable. Their proofs are simple, but if we did not wish to prove these axioms semantically, we could add a definition mechanism to our proof term language and export the KeYmaera X proofs of these derived axioms. The vast majority of axioms and rules are identical to their counterparts in KeYmaera X so that KeYmaera X proofs can be replayed in the verified checker. Differential invariants are an exception: our formalization necessarily differs because we do not formalize differential formulas $(\phi)'$, and we will use schemata instead of axioms for soundness reasons.

```
datatype lrule = ImplyL | AndL | HideL | FalseL | NotL
  | CutLeft formula | EquivL | BRenameL ident ident | OrL

datatype rrule = ImplyR | AndR | CohideR | TrueR | EquivR | Skolem
  | NotR | HideR | CutRight formula | EquivifyR | CommuteEquivR
  | BRenameR ident ident | ExchangeR nat | OrR

datatype axRule = CQ | CE | G | monb

datatype axiom =
   AloopIter | AI | Atest | Abox | Achoice | AK | AV | Aassign
 | Adassign | Advar | AdConst | AdPlus | AdMult | ADW | ADE | ADC
 | ADS | AEquivReflexive | ADiffEffectSys | AAllElim | ADiffLinear
 | ABoxSplit | AImpSelf | Acompose | AconstFcong | AdMinus
 | AassignEq | AallInst  AassignAny | AequalCommute | ATrueImply
 | Adiamond | AdiamondModusPonens | AequalRefl | AlessEqualRefl
 | Aassignd | Atestd | Achoiced | Acomposed | Arandomd
```

A lemma `axiom_valid` ensures that all supported axioms have been proved valid.

```
lemma axiom_valid:"valid (get_axiom axiom)"
```

Type `ruleApp` represents one rule application of any kind. The natural number arguments of the constructors `RightRule` and `LeftRule` indicate the index of the respective succedent or antecedent formula to which the right rule or left rule is applied.

Note that the KeYmaera X axiom for DI over systems with more than one ODE has a known, albeit contrived soundness exploit. For this reason our checker uses axiom schemata which check an additional side condition to ensure soundness. The KeYmaera X proof term exporter translates applications of the unsound KeYmaera X axioms into applications of the sound schemata, and likewise translates proofs that use differential formulas $(\phi)'$ into ones that do not, because differential formulas are not in the language of our checker.

```
datatype ruleApp =
  URename ident ident
| RightRule rrule nat
| LeftRule lrule nat
| CloseId nat nat
| Cohide2 nat nat
```

```
| Cut formula
| DIGeqSchema ODE trm trm
| DIGrSchema ODE trm trm
| DIEqSchema ODE trm trm
```

Type `pt` captures an entire KeYmaera X proof. An important limitation of our checker is that it does not check first-order real arithmetic proofs, but rather treats them as assumptions, represented by constructor `FOLRConstant`. We treat such proofs as out of scope because verified arithmetic solvers are presently not competitive with unverified ones despite extensive research (Harrison, 2007; Platzer, Quesel, & Rümmer, 2009; McLaughlin & Harrison, 2005; Mahboubi, 2007; Li, Passmore, & Paulson, 2019; Cohen, 2012; Narkawicz, Muñoz, & Dutle, 2015; Cordwell, Tan, & Platzer, 2021). We find it more fruitful to simply make our arithmetic assumptions explicit, so that they might be checked in the solver of our choice. A proof is started by declaring the conclusion as an open subgoal in constructor `Start`. Rules are applied with `RuleApplication`, while `AxiomaticRule` and `Ax` state that every axiomatic rule and axiom are sound proof states and valid, proved theorems, respectively. The numeric argument to `RuleApplication` indicates which premise the rule is applied to. In the case that the `ruleApp` argument is a left or right rule, the argument will further specify which antecedent or succedent formula the rule is applied to. Uniform substitution is applied to proof states (rules) in `PrUSubst`. Proof term `Sub l r i` is the main tool for composing multi-step proofs: first the proof terms `l` and `r` are checked, then the proof rule resulting from `r` is applied to subgoal `i` of the proof state resulting from `l`. The remaining constructors `FNC` and `Pro` are additional composition principles, similar to `Sub`.

```
datatype pt =
  FOLRConstant formula
| Start sequent
| RuleApplication pt ruleApp nat
| AxiomaticRule axRule
| Ax axiom
| PrUSubst pt subst
| FNC pt sequent ruleApp
| Pro pt pt
| Sub pt pt nat
```

The proofchecker is represented as function `pt_result` which traverses a proof term and returns an option which, if successful, contains the final proof state.

```
fun pt_result::"pt ⇒ rule option"
```

Finally, we can state soundness of the proofchecker as a whole. Proofs which pass the checker result in sound rules as proof states, so long as arithmetic assumptions hold:

```
lemma proof_sound:"pt_result pt=Some rule ⇒ QEs_hold pt ⇒ sound rule"
```

The predicate `QEs_hold pt` captures the assumption that arithmetic assumptions hold. It is an inductively defined predicate over proof terms which says that for ev-

ery constructor `FOLRConstant formula` that appears in the proof term, we assume `valid formula`. Because `FOLRConstant` assumes a formula without proof, it is crucial that `FOLRConstant` is only ever applied to formulas which belong to decidable fragments for which trustworthy decision procedures are available, i.e., first-order real arithmetic.

Lemma `proof_sound` extends past dL soundness results (Platzer, 2017a, Lem. 23-25) by additionally proving soundness of all the rules we formalized, including bound renaming and sequent calculus rules which were not explicitly proved sound in the paper presentation of the dL uniform substitution calculus.

2.9 Code Generation

Here we discuss the minutiae of generating code from our Isabelle/HOL formalization and integrating that code with KeYmaera X. Isabelle/HOL supports several target languages for generated code, of which we have performed experiments with Scala and limited experiments with Standard ML.

To generate code, we must ensure all definitions use constructs that are supported by the generator. For example, Isabelle/HOL only permits quantifiers to be code-generated for types whose values can be finitely enumerated, represented by the locale[16] `enum`. Our formalization is parameterized by an enumerable identifier type, so a type of identifiers is specified as part of the code generation process. The latest formalization allows bounded-length strings for identifiers, but experiments were performed with bounded integer identifiers. Any experiment using strings would require broader changes to the representation of expressions and substitutions, as some constructors consume space linear in the number of available identifier names. The substitution data structure, for example, contains total functions from identifiers to options of replacements, thus the space required to represent the structure is linear in the number of available identifier names. A production-quality proofchecker would represent a substitution as a list containing only substituted terms, independent of the number of identifiers available. In some cases, simpler types are chosen for stylistic reasons, even though Isabelle/HOL allows generating a more complicated type. Specifically, we use machine words to represent numeric literals in order to better match the formalization of interval arithmetic used in VeriPhy (Chapter 3). The rational literals which are common in dL can be reconstructed using the division operator.

In addition to Scala, Standard ML was explored as a code generation target because recent work (Hupel, 2019b) has developed a verified Standard ML code extractor for Isabelle/HOL, which would provide a strong argument that the results of executing our proofchecker are trustworthy. To the best of the author's knowledge (Hupel, 2019a), however, that extractor does not support the full range of features used by the dL formalization in Isabelle/HOL, as our formalization relies heavily on the predicate compiler, locales, `typedef`, and code generation for finite set data structures, all at the same time.

After configuring the code generator, we extracted the function `pt_result` to Scala. The Scala function is a function from a proof term to (an option containing) the final proof

[16]Locales are an Isabelle/HOL feature which support modularity and which can be loosely compared to typeclasses that can contain theorems.

state. Our proofchecker soundness theorem says that the proofchecking function in Scala is correct, assuming that code generation and Scala compilation are correct. In order to run the proofchecker, we still need to construct the Scala representation of a proof term.

In order to allow the proofchecker to run independently from KeYmaera X, we developed a textual format for proof terms and wrote by hand a trusted parser which builds the Scala representation of the proof term in memory. Because we do not formalize or verify the parser, our proofchecking program is only sound under the assumption that the parser is correct. If we wish to eliminate this assumption, we could have KeYmaera X call our generated code directly rather than generating a text file to be read by the checker. While the parser is trusted, the code is simple since we represent proofs in an S-expression-like format, i.e., in prefix notation with full parenthesization. Because all expressions and substitutions are represented as S-expressions, the format is so verbose that a full example would not be instructive. Instead, we give small proof term fragments as syntax examples.

Program variables are numeric: the nth variable x_i is written `in`, that is, it is prefixed with the letter `i`. The term `(Geq (Var i2) (Var i1))` represents the formula $x_2 \geq x_1$. Recall that the `Start` rule is used to start a proof. Specifically, a proof term of form `(Start ((a1 ... aN) (s1 ... sN)))` is used to start a proof of the sequent whose antecedent formulas are all the formulas `a1 .. aN` and whose succedent formulas are the formulas `s1 ... sN`. As another example, suppose we want to check the steps contained in some proof term `(pt)` first, then apply `AndL` to some position `(pos)` of premise `(prem)` as the next step. Then we would write the proof term syntax `(RuleApp (pt) (Lrule (AndL) (pos)) (prem))`. While our proof term format is verbose, the parser is fast enough that verbosity is not the bottleneck in practice. The largest proof term generated in our tests was 56 `MiB` when represented in our textual format, but parsed in less than a second on the author's workstation. Less time was spent parsing the proof term than checking it. The speed is attributed to the fact that the parser is one-pass and tail-recursive. Copying of large strings is also avoided, i.e., the parser scans through the input string once and maintains the index of the current location as it goes, rather than splitting the input string into substrings and allocating large amounts of memory in the process.

While proof term parsing was simple, the greater challenge was to generate a proof term to serve as input. We instrumented the KeYmaera X core to record every proof step and generate the corresponding steps of a proof term. While the axioms and rules of the Isabelle/HOL formalization follow the KeYmaera X implementation as closely as possible, important challenges and limitations arise wherever the two differ:

- The (systems) differential invariant axiom of KeYmaera X has a soundness bug. While that bug is only a concern for pathological proofs, a verified checker needs sound rules. To ensure soundness, we use schemata for differential invariants rather than an axiom.
- The Isabelle/HOL formalization does not have differential formulas $(\phi)'$ because there is not an obvious general-case semantic definition for them. Instead, we implement the $>$ and $\geq$ cases of $(\phi)'$ separately. The other cases are derivable (Platzer & Tan, 2020; Platzer, 2018a). In short, the Isabelle/HOL formalization uses a family of

schemata rather than a single schema. While proofs using the schemata are not significantly more difficult, we must automatically translate differential formula-based KeYmaera X proofs to use the family of schemata. That translation is tightly coupled with the implementation details of KeYmaera X tactics related to differential invariants, and would likely break if those tactics ever change.

- The verified proofchecker expects numeric identifiers, but KeYmaera X uses alphanumeric identifiers. The exporter translates alphanumeric identifiers to numeric identifiers. The translation is not technically challenging, but poses a serious maintainability issue. Recall that every KeYmaera X axiom is implemented as a concrete formula and that our proof term format does not explicitly write the *content* of the axioms, only their names. Because a proof term only specifies the *names* of axioms, the axiom *definitions* are tightly coupled between KeYmaera X and the checker.

  The issue is best demonstrated with a contrived example: suppose that KeYmaera X contained two axioms $x^2 \geq 0$ and $x^2 \geq 0 \rightarrow x^2 \geq -1$ and we wished to check a proof of $x^2 \geq -1$ which uses both axioms. The implementer of the proofchecker must remember that both axioms use the same variable name x in KeYmaera X, ensure that both axioms use a single identifier `iK` in the Isabelle/HOL formalization, and make sure that the proof exporter consistently translates variable x in the KeYmaera X proof to `iK` in the proof term. If they forget to do so, the proofchecker may (for example) read the axioms as $y^2 \geq 0$ and $z^2 \geq 0 \rightarrow z^2 \geq -1$, in which case the proof term will fail to check[17]. In practice, the exporter's identifier conversion logic is tightly coupled to the choices of program variable names used in axioms of the KeYmaera X core and Isabelle/HOL formalization. If variable names in either implementation were to change without corresponding changes in the exporter, almost all proofs that use the offending axiom will fail to check.
- If proof terms were adapted to use alphanumeric identifiers, the proof term exporter would still need to know every axiom name and its corresponding proof term constructor name. That being said, a mapping from axiom names to constructor names requires less surprising implementation coupling than a mapping for all identifiers.
- The Isabelle/HOL formalization uses a minimal language, knowing that many connectives are definable. The exporter is responsible for expanding defined operators. Many of the expansions are simple, e.g. $(\phi \vee \psi) \leftrightarrow \neg(\neg\phi \wedge \neg\psi)$.
- Recall that axiom DEsys is more precise in our calculus than in KeYmaera X because the syntactic treatment in KeYmaera X can rely on data structure invariants to rule out obviously unreasonable axiom instances, but our semantic treatment cannot. Axioms such as DEsys must have these details filled in during the translation. For the same reason, some of the formalized axioms constrain their arguments to be free of primed variable dependencies x' when the KeYmaera X axiom does not. The

[17]When performing a proof in KeYmaera X, axiom instances are automatically renamed to use whatever variable name is necessary. The same renaming rule is available in the proofchecker. Regardless, the proofchecker only performs renaming when a proof term tells it explicitly to do so, for which reason the example given here would fail to check.

translator must resolve these axiom discrepancies.

The translation between KeYmaera X proof terms and the verified proofchecker's input language is not theoretically deep, but as the preceding points suggest, it is quite brittle because it depends on many implementation details of KeYmaera X and the formalization. For that reason, the proof term exporter is not part of the standard KeYmaera X release and remains an experiment, albeit an insightful one.

In Section 3.4, the proof term exporter is used to export proof terms from proofs of safety of the sandbox controllers generated by the VeriPhy synthesis tool. Those proof terms contain $\approx 10^5$ proof steps, placing them on the same order of magnitude as recent research case studies performed in KeYmaera X (Mitsch et al., 2017). The proof terms were successfully rechecked by the verified proofchecker in ≈ 1 second on the author's workstation, which is less than 2 % the total time taken to check recent case studies of similar scale (Mitsch et al., 2017). The speed of the proofchecker is attributed to the fact that it assumes rather than checks the truth of arithmetic properties, whereas arithmetic solving can dominate proofchecking time in KeYmaera X.

Not only did the proofchecker process proof terms of nontrivial size at a fast rate, but it exercised a broad range of proof rules available in KeYmaera X because the VeriPhy safety proofs use a broad range of proof rules. Notable exceptions include *differential ghosts* and *differential solutions* for ODE systems as well as (derivable) liveness rules for diamond modalities $\langle\alpha\rangle\phi$. Despite the fragility of the proof term exporter, these experimental results indicate that the fragment of the KeYmaera X core we verified is significant and that the verified proofchecker scales to proof terms of significant size.

Trusted Code Base. Having presented the verified proofchecker, we summarize the *trusted computing base* on which its correctness lies. We make standard assumptions on the soundness of Isabelle/HOL and its code generator, as well as the correctness of the Scala compiler and runtime. While verifying such components is interesting future work, it is both a significant undertaking in its own right and orthogonal to the work discussed in this chapter; if such components are verified in the future, the verified tools could simply be applied to the work we have presented. We also assume correctness of a hand-written parser for proof terms. As discussed earlier in this section, we deem the parsing assumption acceptable because of the simplicity of the proof term format.

2.10 Discussion

Prior works (Platzer, 2015b, 2017a) proved on paper that the dL uniform substitution calculus is sound (Platzer, 2017a), but because paper proofs are written and checked by humans, they are vulnerable to the following errors: *i)* soundness errors by human logical blunder and *ii)* completeness errors by omission: paper proofs often focus on the simplest cases of the proof, but do not show that a prover implementation will continue to be correct in the most difficult cases, exactly when it is most in need of formal assurance. Even relative completeness results for the dL calculus (Platzer, 2017a) do not preclude errors of omission. The issue is not whether our proof calculus can prove every dL formula; the

issue is whether we have proved soundness for every proof principle which we implement. The first kind of error was addressed by formalizing the soundness proof in Isabelle/HOL: a proofchecker will reliably check our proof for logical missteps, so long as we have not encountered a soundness bug in Isabelle/HOL itself, and so long as we define the semantics of dL correctly. The definitions of the dL semantics are ≈100 lines of easy Isabelle/HOL code which build on existing real analysis and ODE libraries. Thus, the dL semantics are short enough to be checked by hand, so long as the reader is also careful to check that the usage of those libraries in the semantics is correct. The second kind of error is more subtle: now that we have formalized dL in another prover, how do we know that our formalization did not leave out some important, yet unsound feature? After all, if we forget to include a certain feature in our paper proof, we might very easily forget it in a mechanized proof as well, and we cannot catch a soundness error in a rule which we have not even stated. In the following discussion, we use the phrase "errors of the second kind" specifically for axioms and rules which were not formalized, *but* are unsound. However, we are also generally interesting in ensuring exhaustiveness *because* an inexhaustive checker might contain errors of the second kind.

Errors of the second kind are addressed by Section 2.9: we instrumented the KeYmaera X prover for dL to export a proof term for each proof, which we then replayed in an auto-generated proofchecker. If we forgot to formalize a key feature, we would notice because our proofchecker would fail to check the terms produced from KeYmaera X. Then, we are never left to wonder about the exhaustiveness of our formalization, because even if the Isabelle/HOL proof forgot some feature of KeYmaera X, the implementation of KeYmaera X certainly cannot forget. While the work involved in bringing a mechanization up to speed with the practical implementation can be significant, there is also a second payoff in theory: the resulting proofchecker is guaranteed sound by construction and can be used as a backup for the standard KeYmaera X prover core, effectively removing the core from the trusted computing base.

2.10.1 KeYmaera X Soundness Bug

One might ask whether errors of the first and second kind are both significant in practice. Our experience suggests that both kinds are important, specifically our experience with finding a soundness bug in the KeYmaera X implementation of the bound renaming rule (rule BR). The verification of rule BR was planned before exhaustiveness test were performed, yet there was no prior informal (Platzer, 2017a) proof nor formal proof. Within the context of our formalization, the bug in rule BR is an error of the first kind because we intended to verify this rule from the start. However the fact that previous proofs (Platzer, 2017a) did not address rule BR demonstrates that errors of the second kind are equally crucial; had we neglected to formalize rule BR at first, its soundness bug would be an error of the second kind.

We now discuss the details of the BR bug. The bound renaming of x to y in postcondition ϕ is sound when:

$$\{y, y', x'\} \cap \mathrm{FV}(\phi) = \emptyset$$

The need to include x' in this condition is counter-intuitive and, until the Isabelle/HOL formalization was developed, KeYmaera X was unsound because x' was not checked. Once the bug is discovered, it is straightforward to construct an example where this bug leads to a soundness violation, for example:

$$\not\leftarrow \frac{[x := x']x = x'}{[y := x']y = y'}$$

The premise is valid, but the conclusion is not. Thankfully, no existing code in KeYmaera X relied on the presence of this bug, so changing the precondition as indicated above was sufficient to fix the bug.

Note that the bug we found is in rule BR, which was *not* proved sound in prior paper proofs (Platzer, 2017a). This highlights the importance of exhaustiveness (and avoiding errors of the second kind). It is crucial to identify and verify all rules which will be used in practice. Next, we will discuss how the BR bug was found, which will lead into a discussion of how errors of the second kind are addressed.

Before developing a verified proofchecker, we identified a list of axioms and rules which are known to be used frequently in KeYmaera X. That list happened to include rule BR, in which sense the BR bug is an error of the first kind. Upon starting the main dL soundness proof, Isabelle/HOL will require us to prove a case for the validity of each axiom and a case for the soundness of each rule. Typically, we organize the proof of each case as a separate lemma. Our initial proof attempt for BR tried to show that the soundness condition previously used in KeYmaera X ($\{y, y'\} \cap \mathrm{FV}(\phi) = \emptyset$) makes BR sound. As is typical, our proof attempt proceeded by manually stating several key intermediate formulas and attempting an interactive proof of each. The crucial step which failed was one which appealed to the static semantics lemmas (Section 2.4), each of which have assumptions on variable occurrences. By inspecting the proof state, it was clear that a stronger variable occurrence condition $\{y, y', x'\} \cap \mathrm{FV}(\phi) = \emptyset$ was required to prove the step, from which the proof of the lemma followed. While inspecting the failed proof attempt required human expertise, the experience of debugging a proof both suggested a fix for the code and suggested a strategy for finding a counterexample to the old rule: choose any formula which satisfies the original side condition but not the revised one.

Because prior proofs (Platzer, 2017a) omit rules such as BR which we proved, it is natural to ask whether our formalization initially omitted other important rules and whether those rules contained errors of the second kind. Our formalization did omit rules which get used in practice, e.g., rules which will be used in Chapter 3. Testing the verified proofchecker led us to formalize such rules, all of which were sound and thus not errors of the second kind. We believe such omissions to be common because KeYmaera X contains many axioms beyond the core dL axioms (Platzer, 2017a). Firstly, KeYmaera X features many *derived axioms* which are implemented in terms of the core dL axioms and thus are not soundness-critical, but are often used for convenience. Secondly, prototypes of new KeYmaera X features may employ experimental axioms which are not intended for widespread use. Because omissions of axioms could continue to occur, it remains essential that exhaustiveness testing is employed to prevent errors of the second kind, even if we

caught BR without the use of testing. At the same time, it is also valuable that we manually assessed which dL rules to verify, because even the proofchecker would not identify a missing rule if that rule were not used in the cases which are tested.

Our verified proofchecker provides important cross-checking for the soundness of dL proof rules, but it is important to acknowledge the one proof principle which is not verified by the soundness theorem `proof_sound`: real arithmetic solving. The pragmatic decision to assume rather than prove validity of real-arithmetic goals is motivated by the known difficulty of verifying efficient real arithmetic solvers. On the other hand, the soundness-critical core of KeYmaera X is less than 2,000 lines of code (Fulton et al., 2015), which is significantly less than tools such as Mathematica that are commonly used for practical real arithmetic solving. While the exact size of Mathematica's trusted computing base is not known, the Logician's skeptical principles tell us we should not assume *any* code is correct until we have formal, positive evidence of correctness. Thus, it is not our goal to claim that a soundness violation could never occur in KeYmaera X, rather by verifying the dL proof calculus we have verified the parts of the KeYmaera X codebase which are under our direct control[18]. Arithmetic confidence could be increased by cross-checking several solvers, proving formulas interactively, or, when possible, by using verified solvers.

2.10.2 Implications for Developing and Formalizing Provers

Just as the formalization presented in this chapter is not the first work on the verification of theorem provers (Section 2.11), it is unlikely to be the last. We discuss lessons learned which are applicable to future theorem prover verification efforts.

We start by discussing our theorem prover verification approach at a high level, i.e., discussing our decision to extract proof terms from an existing prover implementation and cross-check them in a separate, verified program. A major benefit of this approach is that it can be pursued with minimal changes to the existing implementation of a prover, so long as that prover can be readily instrumented to record proof terms, as was the case with KeYmaera X. While leaving the implementation of KeYmaera X intact, this approach gave us complete control over the source code of the verified proofchecker, which is crucial because the code of a verified program is typically optimized for simplicity of its correctness argument and thus may differ significantly from other implementations.

Our approach was made feasible by two key facts: KeYmaera X is implemented with a small core consisting of simple proof steps (Fulton et al., 2015) and the uniform substitution calculus which underlies KeYmaera X is well-studied (Platzer, 2017a). Because the verification of code is labor-intensive, the difficulty of verification increases with the size of a prover core. For example, the proofchecker introduced in Chapter 7 takes a large-core approach, which would complicate any effort towards verification of its soundness. Even for simple code, verification can be challenging when the correctness of simple code relies on deep mathematical results. Soundness arguments for logics are often complex; notably, dL's soundness argument relies on non-trivial theorems of real analysis (Platzer, 2017a).

[18]Notwithstanding a known bug in the KeYmaera X implementation of differential induction, which discussed in Section 2.5

The availability of a non-mechanized proof greatly accelerated the development of a mechanized proof. A general benefit of proving soundness on paper first is that one can avoid wasting great effort attempting a mechanized proof of a false statement whose falsehood might be determined more quickly on paper. That being said, one may wish to attempt a formalized proof of a new logic in order to identify mistakes or deepen their understanding, but one must be prepared for the possibility that changes to logic would require rewriting large sections of formal proofs.

When developing an executable, verified proofchecker (Section 2.9), the greatest challenges arose where the formalization differed from KeYmaera X. By its very nature, the Isabelle/HOL formalization is based on the theory of the dL uniform substitution calculus, so it is also the case that the greatest challenges arose where the implementation of the theorem prover differed from the underlying theory. For example, special handling was required for rules which reason about systems of (more than one) ODE and for differential formulas $(\phi)'$, both of which are crucial in KeYmaera X but have received less theoretical attention than other features of dL.

One particularly subtle takeaway is that the soundness formalization will likely diverge from the prover wherever the prover provides a feature whose *semantics* are difficult to define formally. For example, differential formulas $(\phi)'$ may be easily explained on paper, but the task of defining their semantics was so subtle, particularly in the presence of uniform substitution symbols, that the formalization opted to omit differential formulas from the core language of dL expressions. Such divergences greatly complicate maintenance, limiting the long-term applicability of our proofchecker despite its support for non-trivial dL proofs. For our approach to result in a maintainable proofchecker, we would want the axioms, rules, *and semantics* from the theory to be general enough to support *all* proofs that the theorem prover does. For provers such as KeYmaera X which currently extend their underlying logic, our experience could serve as motivation for extending the underlying logic to catch up with the features used in practice.

Alternative approaches include the verification of an existing codebase and the development of an entirely new, production-quality verified codebase. Verification of existing codebases is often complicated by the fact that most existing programs contain bugs. Development of entirely new verified programs, on the other hand, is complicated by the simple fact that existing theorem prover codebases typically represent many person-years of engineering effort. By discarding the entire codebase of a prover, one is likely to discard too much. For example, the codebases of mature provers typically include features, such as user interfaces, which provide important conveniences but are orthogonal to soundness concerns and thus ought not be discarded in the name of soundness proofs. Though these alternative approaches come with significant challenges, the benefits of their (successful) application would also be notable. By verifying a prover's soundness in its entirety, one eliminates the need to maintain two separate proofcheckers and eliminates the possibility that the soundness proof is inexhaustive with respect to the prover implementation.

No matter which approach is taken, mechanized soundness proofs for theorem provers are nontrivial undertakings. Yet, our experience shows that the process of mechanizing the soundness proof can provide valuable insights. A mechanized proof can catch soundness bugs which have evaded detection despite extensive use (Section 2.10.1), explain the reason

the code is unsound, and thus suggest counterexamples and solutions. Beyond improving the soundness of a theorem prover, the formalization process demands confronting any differences between a prover's theory and implementation. By confronting those differences, one might inspire generalizations of the theory or even implementation simplifications.

2.11 Related Work

Formal verification has been pursued for several theorem provers because their role of proving other systems correct is critical. We compare our work to other works on theorem prover verification. In general, the differences among formalizations mirror the differences among the underlying logics, so the unique features of our work include real analysis proofs, ordinary differential equations (ODEs), and an explicit treatment of substitution.

Barras and Werner (Barras & Werner, 1997) verified a typechecker for a fragment of Coq in Coq. Harrison (Harrison, 2006b) verified (1) a weaker version of HOL Light's kernel in HOL Light and (2) HOL Light in a stronger variant of HOL Light. Myreen et al. have extended this work, verifying HOL Light in HOL4 (Myreen, Owens, & Kumar, 2013; Kumar et al., 2016) and using their verified compiler CakeML (Kumar et al., 2014) to ensure these guarantees apply at the machine-code level. Myreen and Davis proved the soundness of the ACL2-like theorem prover Mitawa in HOL4 (Myreen & Davis, 2014). Anand, Bickford, and Rahli (Anand & Rahli, 2014; Rahli & Bickford, 2016) proved the relative consistency of NuPRL's type theory (Constable et al., 1986; Allen et al., 2006) in Coq with the goal of generating a verified prover core. Twelf (Pfenning & Schürmann, 1999) was used to formalize the LF (Harper, Honsell, & Plotkin, 1993) logical framework on which it is based (Martens & Crary, 2012). These works show that formalization is feasible for a wide array of logics, but careful attention must be given to the foundations of the host prover and the libraries it provides.

Like the above works, the goal of this chapter is to verify the correctness of a theorem prover (KeYmaera X). This goal is one part of the thesis' broader goal of bulletproof foundations: formal proofs are only as trustworthy as their foundations. Since proof in dL is a linchpin of the VeriPhy approach, it is crucial to show the soundness of dL's foundations. Chapter 2 exposed unique technical challenges because the foundations of dL differ greatly from those of the other provers. Soundness for the differential equations of dL involved significant proofs involving concepts of real analysis. The KeYmaera X core is based on a uniform substitution calculus (Platzer, 2017a), so the formalization also includes significant proofs about substitution.

Our verified checker supports a significant fragment of the KeYmaera X core, including enough to recheck proofs of safety for monitor-based synthesized controllers (≈100K steps). Together, these two pieces provide a multifaceted argument how we can trust hybrid systems proofs as done in KeYmaera X. Recently, an independent formalization of classical dGL has been made (Platzer, 2019a), which emphasizes uniform substitution and simplicity of the formalization over suitability for extraction of a verified checker.

The verified dL proofchecker assumes rather than proves that first-order real arithmetic subgoals are valid. While first-order arithmetic is known to be decidable (Tarski,

1951), it has a tight (Davenport & Heintz, 1988) doubly-exponential (G. E. Collins, 1998) time bound. The high complexity of first-order arithmetic means that optimized implementations are essential in practice, and the wide array of verified real arithmetic procedures (Harrison, 2007; Platzer et al., 2009; McLaughlin & Harrison, 2005; Mahboubi, 2007; Li et al., 2019; Cohen, 2012; Narkawicz et al., 2015; Cordwell et al., 2021) have not achieved performance that can compete with the techniques used in production-quality unverified solvers (G. E. Collins & Hong, 1991; Strzebonski, 2006). Because verification of high-performance real arithmetic solvers is a major, long-standing open problem in its own right, we simply identify the real-arithmetic assumptions so that one might check them with verified solvers, unverified solvers, or interactive proofs as desired.

Chapter 3

Monitor Synthesis for Classical Hybrid Systems

This chapter introduces VeriPhy, a design approach and synthesis tool which provides the first implementation of end-to-end verification for dL. We use the name VeriPhy to refer to both the tool and the basic design architecture. In developing VeriPhy, we give the Logician solid formal foundations while giving the Engineer a verified controller which can formally guarantee safety of an untrusted implementation. After developing the approach and a case study on wheeled ground robotics, we identify how the remainder of the thesis can additionally satisfy the Logic-User's needs for robustness, flexibility, traceability without compromising the Engineer's needs and while maintaining a clear basis in logic that suggests how the Logician's needs could be met. The implementation of VeriPhy presented in this chapter is called *classical* VeriPhy because it uses the classical logic dL as its foundation. A second implementation will be presented in Chapter 8 with the goal of addressing classical VeriPhy's limitations. The second implementation will be called *constructive* VeriPhy because of its constructive foundations.

Recall from Chapter 1 that end-to-end verification must resolve the needs of the Logician, Engineer, and Logic-User: the Engineer needs a controller implementation which works in practice despite the presence of complex untrusted code, the Logician demands a foundational argument why the controller is correct at every step of the way, and the Logic-User wants to perform proofs at a high level of abstraction (e.g., hybrid systems) without an excessive proof burden. This chapter contributes a synthesis approach and implementation for dL aimed at resolving these conflicting needs. We evaluate the tool and discuss the limitations that remain (Section 3.8). In short, classical VeriPhy places strong emphasis on pleasing the Logician with a chain of formal proof artifacts, but imposes nontrivial restrictions on the models, proofs, and code supported, which can leave the Engineer and Logic-User wanting when the approach is used in nontrivial applications. Constructive VeriPhy (Chapter 8) will emphasize the opposite design priorities, with the hope that having both implementations available will best meet the needs of all three characters.

To extract implementation-level guarantees from high-level models while maintaining strong foundations, VeriPhy takes a pipeline approach which divides the synthesis process into steps, allowing us to gradually bridge the gap between hybrid systems models and

executable code. The gap is bridged by starting from both ends and working towards the middle: on the front end, we translate the safe input model into another safe model called a *sandbox* which is easier to execute; on the back end, we show that executions of compiled code simulate executions of the safe sandbox model and thus inherit its safety properties.

The inputs of the pipeline are a hybrid system model and its proof in dL. Each front-end step of the VeriPhy pipeline brings its input closer to executable code while maintaining formal proofs. In an informal sense, the correctness theorem for each of these steps is like a refinement property that reduces a safety theorem of the output to safety of the input. The result is a provably safe dL model which is easier to execute than the input because its plant model only uses discrete constructs whose execution is well-understood and its control model explicitly captures the desired sandbox algorithm. Each back-end step shows a simulation argument: any program transition that occurs in compiled code simulates a transition in the source code, which simulates a discrete semantics of dL and finally the standard real-valued semantics of the sandbox in dL. Because the sandbox is proved safe in dL, the ending state of each transition satisfies the postcondition of the safety theorem. Assumptions on safe sensing imply that the real state of the physical world is one such simulated dL state and thus inherits the safety property in the physical world. The formal proofs described here satisfy the Logician's desire for formal evidence of correctness.

The correctness of some steps has been proved in advance in the general case; the other steps use automated proofs. Automation of transformation correctness proofs is important because we wish to reduce work for the Logic-User: ideally, synthesis and its correctness proof should be fully automatic once the input model has been proved. We will discuss (Section 3.8) the challenges of conclusively achieving full automation in the classical setting of dL and propose how the following chapters will alter their approach to achieving this goal. Our other crucial goal is to minimize difficulty for the Engineer. An important feature of our basic design is that VeriPhy is agnostic to low-level implementation details because synthesis and correctness proofs are driven entirely by a dL model and its correctness proof, which are free to ignore many low-level details from implementation code. By designing VeriPhy to be agnostic to low-level details, we make it easy for the Engineer to change those details on their own without having to change a model or its proof (or wait for the Logic-User to do so). That is not to say that all difficulties for the Engineer have been resolved. For example, the representation of arithmetic in classical VeriPhy (Section 3.4) will prove to be an obstacle for the Engineer when models grow more complex.

VeriPhy takes a provable runtime monitoring approach. A key benefit of the approach is that it yields a loose coupling between proofs and code, so that the Engineer's code is untrusted and can change as frequently as desired without additional verification work. This loose coupling also leaves the Engineer free to use the language of their choice. For instance, tools such as Stateflow/Simulink (*Stateflow Documentation*, 2021; *Simulink Documentation*, 2021) have extensive libraries that are widely used in control software, but are not easily verified. Specifically, we monitor whether the implementation complies to the source model. The Logic-User wants source model verification to be as simple as possible and the Logician wants safety of the model to closely match an intuitive notion of real-world safety. For these reasons, the source model is a hybrid program, verified in KeYmaera X with dL.

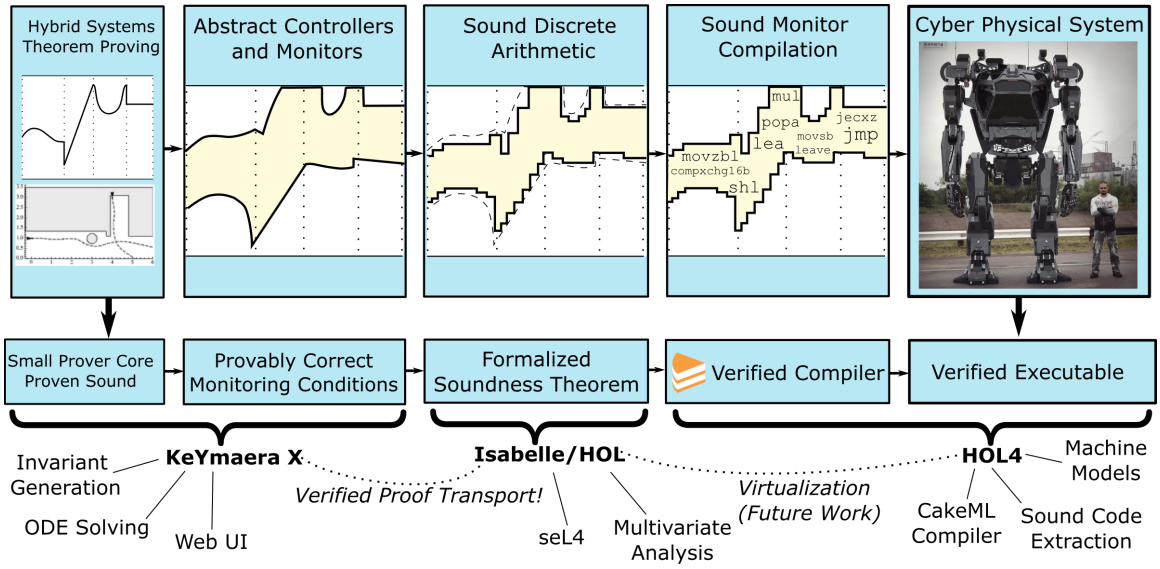


Figure 3.1: High assurance artifacts and steps in the VeriPhy verification pipeline.

We list the steps of the (classical) VeriPhy pipeline. We use ⇒ as the bullet on steps whose correctness arguments move in the *forward* direction by showing that safety of an output model follows from safety of an input, while we write ⇐ on steps whose correctness argument works *backward* by showing that execution of the output simulates execution of the input. We use a standard bullet on steps which are not on the critical path of the pipeline. Pipeline steps, tools, and artifacts are graphically summarized in Fig. 3.1. The *Future Work* section of Fig. 3.1 refers to the future work of virtualizing HOL4 inside Isabelle/HOL for a smaller trusted code base if virtualization techniques (Immler, Rädle, & Wenzel, 2019), which are discussed in Section 3.1, become mature enough to support our formalization.

⇒ The ModelPlex (Mitsch & Platzer, 2016b) tool, which is implemented as a feature of KeYmaera X, is invoked to synthesize a *control monitor formula* and *plant monitor formula*, which determine whether control decisions and physical evolutions respectively satisfy the assumptions made of them by the model. The use of monitors is crucial for showing that implementation-level behavior matches a model and thus inherits its safety guarantees. Automating monitor formula synthesis reduces the amount of manual modeling and proof effort required by the Logic-User. Recall that the *plant* models physics using systems of ordinary differential equations (ODEs).

⇒ The monitors are combined with a proven-safe *fallback controller* (which is provided as an input) to form a *sandbox controller*. Fallback controllers are crucial for allowing safe control even when the control decisions of some untrusted controller implementation (henceforth called the *external controller*) fall outside the proven-safe model. The sandbox controller applies the external controller's decisions whenever those decisions satisfy the controller monitor and otherwise invokes the fallback controller. Sandbox controllers crucially allow extending proven safety guarantees from models

to implementations that incorporate external controllers.

If the *plant* monitor fails, it means the laws of physics do not match modeling assumptions, so an alarm is raised[1].

$\Rightarrow$ The sandbox controller is automatically proven safe in KeYmaera X, in part by reusing the contents of the hybrid system safety proof. Automating the sandbox safety proof is crucial for reducing the Logic-User's proof effort. Reusing components of the system safety proof is a powerful approach for automating proofs despite the undecidability of safety.

- The sandbox safety proof is optionally rechecked in the verified proofchecker (Chapter 2) to eliminate the KeYmaera X core from the trusted base. Though optional, removing the KeYmaera X core from the trusted base is useful to a Logician who wants to trust fewer lines of code. Chapter 2 used these proof terms to assess exhaustiveness of the dL formalization.

$\Leftarrow$ The exact real arithmetic from hybrid systems semantics ($[\![\cdot]\!]$) is conservatively recast as fixed-point interval arithmetic ($[\![\cdot]\!]$), and this recasting is formally proven in Isabelle to be sound. We used fixed-point interval arithmetic because its time, space usage, and behavior are extremely predictable, which matters for application domains such as aviation. Additionally, fixed-point numbers can be implemented in every major theorem prover and programming language. Arithmetic translation is an important compilation step, and its soundness proof is important to the correctness of compilation. By automatically compiling high-level real arithmetic to low-level arithmetic, we free the Engineer from implementing low-level arithmetic herself.

$\Leftarrow$ The resulting program is automatically proven equivalent (in HOL4) to a CakeML program (Kumar et al., 2016). The heart of the proof says that sensing and actuation in the model can be modeled by a CakeML state machine (Ho et al., 2018). By transitioning to CakeML, we enable the use of the following verified compilation steps. Proof automation avoids excessive proof effort.

$\Leftarrow$ The equivalent CakeML program ($[\!\{\texttt{cmlSandbox}\}\!]$) is compiled to machine code in the CakeML verified compiler. CakeML's verified compilation guarantees are one crucial step of the end-to-end correctness argument.

$\Leftarrow$ The machine code ($\{\!|\texttt{CML(cmlSandbox)}|\!\}$) is linked with the Engineer's trusted sensing and actuation code. By automating the synthesis and compilation of the sandbox controller, we free the Engineer from writing the sandbox logic herself.

By combining forward and backward steps, we solve a subtle aspect of end-to-end verification: we wish to show an implementation safe by showing it complies with a model, yet a hybrid system may be so strict that an implementation does not match it exactly. For example, an ODE specifies the trajectory of each of its bound variables exactly, yet real-world sensor values typically contain noise and will not match the trajectory exactly,

[1]In practice, the fallback controller can also be applied in this case, but the important point is that a formal safety guarantee is fundamentally impossible when physics violates our assumptions. Instead, fallback would serve as a best effort.

even when they come close. The forward steps transform the input model into a sandbox model that is *more permissive*, thus easier for implementations to comply with, yet still proven safe. The backward steps then ensure the implementation's compliance with the relatively permissive sandbox model.

Each step is a provable reduction: forward steps reduce safety of an output to safety of an input, while backwards steps (listed in Fig. 3.2) show executions of the output simulate executions of the input. The reductions culminate in the fact that any execution of the machine-code running on the actual CPS corresponds to some execution of the sandbox. Because all executions of the sandbox are proved to satisfy the same safety property as the source hybrid system, then the execution of the actual CPS is also known to be safe. The differing brackets at different levels represent different formal semantics owing to a change in programming language, state type, or both. For example, the change from $[\![\cdot]\!]$ to $(\!|\cdot|\!)$ captures the change from real-valued variables to interval-valued variables. The simulation theorem for this step bridges the gap between reals and intervals by showing that the interval semantics of a program conservatively simulates the real-valued semantics. The CakeML sandbox (`cmlSandbox`) is generated by importing the monitor conditions and fallback used in the dL sandbox.

Note that the sensor and actuator drivers, which are provided by the Engineer, are trusted. While verification of sensing and actuation is an important topic, it is also a moving target and presents deep epistemological issues: Not only does hardware change frequently, but any model of hardware requires assumptions on the physics of hardware components, which would become the new trusted base. Our purpose is not to dismiss the usefulness of sensing and actuation verification, but to present an approach whose sensing and actuation assumptions are cleanly isolated so that our approach will stand to benefit from any future advances in those areas. Our sensing and actuation interface consists of a handful of CakeML foreign-functions (Ho et al., 2018) for reading from and writing to the external world, which have precise specifications in HOL4 and can then be implemented in the Engineer's language of choice, often C.

Fig. 3.2 shows the chain of simulation properties proven for the (backward) steps given in Fig. 3.1. The chain of properties in Fig. 3.2 highlights a fundamental challenge of end-to-end verification: we must cross multiple levels of abstraction, from hybrid systems down to discrete arithmetic and finally machine code. Any approach which seeks to prove compliance between low-level code and high-level models, regardless of how exactly in divides an end-to-end proof into steps, would have to confront the same fundamental challenge of crossing abstraction levels.

The use of multiple abstraction layers motivates our use of multiple theorem provers. The KeYmaera X theorem prover for dL provides powerful automation for proofs of hybrid systems, yet dL is domain-specific and thus has no hope of reasoning about other abstraction layers such as discrete arithmetic machine code. Formalizations of proof steps which relate these different layers to one another rely on the ability to formalize languages and their semantics, a task which is best-suited to general-purpose theorem provers. Conversely, general-purpose theorem provers do not provide the extensive out-of-the-box hybrid systems automation that KeYmaera X does, despite their ability to formalize hybrid systems in depth, in principle. Because the domain-specific theorem prover KeYmaera X has

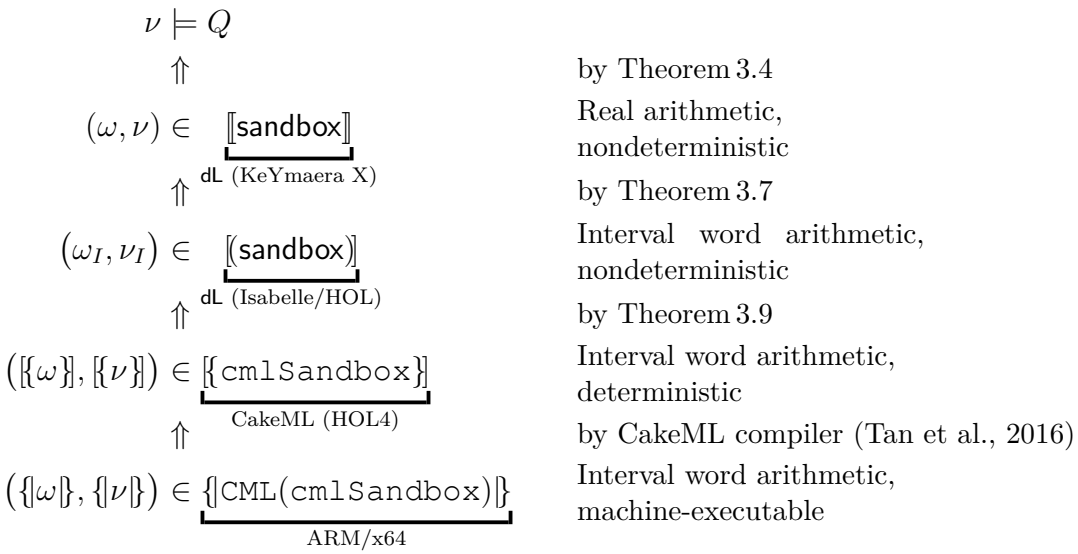


Figure 3.2: End-to-end proof chain for end-to-end result.

strengths and weaknesses that complement general-purpose theorem provers, our choice to employ multiple provers is a natural one.

The three specific provers we used are KeYmaera X, Isabelle/HOL, and HOL4. Our choice to use *both* Isabelle/HOL and HOL4 is motivated by an accident of history: despite both provers sharing closely-related foundations in higher-order logic, Isabelle/HOL has a more comprehensive treatment of real analysis whereas HOL4 has access to unique work on compiler verification. We use all three provers to benefit from each prover's strengths:

- The input of VeriPhy is a dL model which has been verified in KeYmaera X.
- Isabelle/HOL is used to bridge the semantic gap between real numbers and intervals.
- HOL4 and its verified CakeML compiler are used to implement the interval program in ML and compile it to machine code.

The downsides of this approach are an increased trusted base and increased conceptual complexity. Another potential downside of using multiple provers is the need for expertise in the use of each prover, but that was not a major obstacle in our case because each author of VeriPhy contributed expertise with different provers.

In our biased opinion, the advantages of the approach outweighed the negatives. KeYmaera X features extensive domain-specific automation for hybrid systems which does not exist at present in the other provers. Because the translation between reals and intervals must reason about the semantics of dL, it must use a general-purpose prover which can define the semantics of logics. Among existing provers, Isabelle/HOL was a natural choice because a formalization of dL semantics is already available (Chapter 2) and because Isabelle/HOL has strong support for real analysis and sufficient support for discrete arithmetic. CakeML was chosen as a backend not only because of the VeriPhy team's expertise, but because it has good support for foreign functions (Ho et al., 2018) and its correctness

proofs (Tan et al., 2016) go all the way to machine code. Once the decision had been made to use CakeML, the use of HOL4 was a given. If we wish to verify sensing and actuation in the future, we could do so by proving in HOL4 that the hardware drivers implement their specifications.

Different combinations of provers could have been used, with different tradeoffs. If KeYmaera X were not used, we expect that the effort required for hybrid systems proofs would increase. If only HOL4 or only Isabelle/HOL were used, we expect that significant effort would need to be expended on the formalization of results from real analysis or verified compilation, respectively.

To validate our claim that VeriPhy enables end-to-end verification for dL, we have applied VeriPhy to a hardware implementation of a 1D robot model (Section 3.6) and a software implementation of a 2D robot model (Section 3.7.3). In this chapter, we: *i)* present the design and implementation of the VeriPhy pipeline *ii)* evaluate the pipeline on hardware and in software with several models *iii)* discuss limitations of the VeriPhy implementation from the perspective of the Logician, Logic-User, and Engineer, and how the remaining thesis chapters address the limitations.

Contributions. This chapter is based on joint works with Yong Kiam Tan, Stefan Mitsch, Andrew Sogokon, Magnus O. Myreen, and André Platzer (Bohrer et al., 2018; Bohrer, Tan, Mitsch, Sogokon, & Platzer, 2019).

3.1 Related Work

We discuss related works, specifically works on verified compilation, verification of machine arithmetic, and simulation.

Verified Compilation. We use the CakeML (Tan et al., 2016) verified ML compiler and its associated verification tools (Myreen & Owens, 2012; Guéneau, Myreen, Kumar, & Norrish, 2017) based on HOL4, which is part of our trusted computing base. Recent work (Hupel & Nipkow, 2018) provides verified extraction of CakeML code from Isabelle/HOL proofs as opposed to HOL4, but to our knowledge (Hupel, 2019a), its present iteration does not support all features used in our formalization of dL, such as locales, inductively defined predicates, and the built-in set type. Another approach (Immler et al., 2019) allows running a verified HOL4 core inside of Isabelle/HOL, but to the best of our knowledge it did not support all features used by CakeML as of the time this work was performed. Thus, neither effort would allow us to fully remove HOL4 from our trusted base at present, but potentially could in the future.

CakeML is higher-level than other languages that have verified compilers, such as CompCert (Leroy, 2006) for C (for floating-point support, see (Boldo, Jourdan, Leroy, & Melquiond, 2013)) or Jinja (Klein & Nipkow, 2006) for a Java-like language. The smaller gap between CakeML source and Isabelle/HOL definitions makes the verification of sandbox implementation in CakeML against hybrid systems semantics painless. We chose

CakeML compilation over, e.g., translation validation with unverified compilers (Sewell, Myreen, & Klein, 2013) since translation validation can be brittle.

Lustre (Halbwachs, Lagnier, & Ratel, 1992) is a reactive, synchronous language intended for use in safety-critical CPS. It has static analyzers based on abstract interpretation such as NBac (Jeannet, 2003), which eliminate many bugs in practice, but are fundamentally conservative and also consider only the control code, not the physics in the plant. Lustre has a verified compiler, Vélus (Bourke et al., 2017), but verified compilation *alone* does not negate the fact that CPS verification must account for physics. One could choose Lustre as an intermediate compilation target instead of CakeML, but the functional style of CakeML is a much closer match to Isabelle/HOL definitions, while Lustre's reactive model is an entirely different paradigm from both HOL and hybrid programs.

Stateflow/Simulink is a prominent tool for modeling and simulation of hybrid systems. Several approaches generate executable code from Stateflow/Simulink models (Toom et al., 2008; Zou, Zhan, Wang, Fränzle, & Qin, 2013; Yan, Jiao, Wang, Wang, & Zhan, 2020; Tripakis, Sofronis, Caspi, & Curic, 2005), but none of them provide foundational end-to-end guarantees. Of the above approaches, one (Tripakis et al., 2005) is restricted to discrete models. An approach based on Hybrid CSP (Zou et al., 2013; Yan et al., 2020) allows Stateflow/Simulink models to be imported into Hybrid Hoare Prover and verified, after which Hybrid Hoare Prover can export SystemC code. A strength that their approach shares with VeriPhy is the ability to deductively prove safety properties of the system, but their SystemC code generator and the SystemC compiler are not proved correct. In principle, VeriPhy could be extended to support Stateflow/Simulink models as well by using an existing (trusted) tool which translates Stateflow/Simulink models (Liebrenz, Herber, & Glesner, 2018) to dL, but this approach would only work if the resulting hybrid program models match the format expected by VeriPhy, which is unlikely. Gene-Auto (Toom et al., 2008) does not consider safety proofs for Stateflow/Simulink models; its strengths include its emphasis on additionally supporting Stateflow/Simulink's open-source competitor Scicos and its emphasis on generating code within a simple subset of C which would be easy to reason about.

The Facade (Pit-Claudel, Wang, Delaware, Gross, & Chlipala, 2020) intermediate language was developed to provide an extensible approach in Coq for verified compilation from nondeterministic functional programs to machine-code programs which can interact with external code using foreign-function interfaces (FFIs). Because hybrid programs are nondeterministic and because external code is crucial in the implementation of CPSs, Facade would be a natural choice for the development of VeriPhy-like tools based on Coq. On its own, however, Facade would not address the challenges of proving a source (hybrid system) model correct, nor the challenge of soundly sandboxing a CPS implementation against a model. Facade's approach is based on generating proof scripts for correctness of the compilation of each individual program. The authors note that robustness of proof script generation was a challenge. Their conclusion is consistent with our own conclusion that the design of classical VeriPhy, as presented in this chapter, makes the robustness of code synthesis and proof generation a challenge.

Machine Arithmetic Verification. Machine arithmetic correctness verification is a major VeriPhy component. We verify arithmetic soundness foundationally. This is an active research area with libraries available in HOL Light (Harrison, 2006a), in Coq (Boldo & Melquiond, 2011; Daumas, Rideau, & Théry, 2001; Boldo, Filliâtre, & Melquiond, n.d.; Melquiond, 2012), and in Isabelle/HOL (L. Yu, 2013), for examples. The main results we need are results saying that basic arithmetic operations round in the direction specified by the rounding mode. While it is possible others have proved such results, only PFF in Coq (Daumas et al., 2001) has documented such results explicitly. For this reason, we proved rounding results ourselves in Isabelle/HOL using the seL4 (Klein et al., 2010) machine word library. We chose Isabelle/HOL and HOL4 over Coq because their combination of cutting-edge analysis libraries (Immler & Traut, 2016), mature formalization of dL (Bohrer et al., 2017), proof-producing code extraction (Myreen & Owens, 2012), and classical foundations positions them well for our end-to-end pipeline. Standalone programs (Beyer & Huisman, 2018) have also been verified in HOL and Coq (Becker et al., 2018) which certify error-bounds on the outputs of arithmetic expressions. Static analysis programs have been written (Martinez, Majumdar, Saha, & Tabuada, 2010; Majumdar, Saha, & Zamani, 2012; Bouissou, Goubault, Putot, Tekkal, & Védrine, 2009) to detect arithmetic errors in the context of hybrid systems, but none have foundational soundness proofs. Other limitations of existing analyses include supporting linear ODEs (Majumdar et al., 2012), (non-linear) switched systems only (Bouissou et al., 2009), or stability properties only (Martinez et al., 2010). While stability is a fundamental property of a control system, safety is certainly fundamental as well.

Simulation. Simulation is an essential part of evaluating models and designs for any robotic system. Multiple simulation platforms are available, of which AirSim (Shah, Dey, Lovett, & Kapoor, 2018) is a recent platform for UAVs and autonomous cars. Other simulators would likely have worked as well, but we chose AirSim because it is configured with high-fidelity physical and visual models out of the box, which saves us from developing our own simulation model and assessing its accuracy. A pre-existing physical model is especially valuable because we wish to assess how well our own model conforms to other accepted, reasonably realistic models. If we had developed our own simulation, our results would risk an "overfitting" error wherein our safety monitors succeeded only because our simulation and verification models were similar. By using an independently-developed simulation, we reach more meaningful empirical results.

Despite the risk of over-fitting simulations to models, automated simulation of hybrid systems remains a topic of interest because it would provide lightweight validation with minimal time investment. A crucial stepping stone for verified simulation of hybrid systems is the verified integration of ODEs, which has been done in Isabelle/HOL (Immler, 2015).

3.2 1D Robot Example

We introduce an abstract dL model of a ground robot in a 1D corridor. We use this as a running example to describe the VeriPhy pipeline, culminating in a verified implementation

running on commodity robot hardware. A waypoint-following model for curved 2D driving is then considered in Section 3.7 as a larger-scale case study.

The robot can drive freely as long as it avoids hitting an obstacle. We model the robot with instantaneous control of velocity v. This abstraction is reasonable as shown in Section 3.6 because our robot drives slowly relative to its braking power. The controller is time-triggered, i.e., the system delay between controller runs is bounded by some T.

Formula (3.1) expresses model safety as a dL formula $P \rightarrow [\{\mathsf{ctrl};\mathsf{plant}\}^*]Q$. It says all states satisfying assumptions P lead to safe states (Q) no matter how many times the system loop $\{\mathsf{ctrl};\mathsf{plant}\}^*$ repeats. The program ctrl is a discrete time-triggered controller, while the program plant describes physical environment assumptions as an ODE.

$$\overbrace{d \geq 0 \land V \geq 0 \land \mathsf{T} \geq 0}^{P} \rightarrow [\{\mathsf{ctrl};\mathsf{plant}\}^*] \overbrace{d \geq 0}^{Q} \tag{3.1}$$

$$\mathsf{ctrl} \equiv \{\mathsf{drive} \cup \mathsf{stop}\};\ t := 0 \tag{3.2}$$

$$\mathsf{drive} \equiv ?d \geq \mathsf{T}V;\ v := *;\ ?0 \leq v \leq V \tag{3.3}$$

$$\mathrm{stop} \equiv v := 0 \tag{3.4}$$

$$\mathsf{plant} \equiv \{d' = -v,\ t' = 1 \,\&\, t \leq \mathsf{T}\} \tag{3.5}$$

Initially, the robot is driving at a safe distance $d \geq 0$ from the obstacle. We also know the system delay $\mathsf{T} \geq 0$ and maximum driving speed $V \geq 0$. Our safety condition $d \geq 0$ says the robot does not drive through the obstacle. Its controller (3.2) can either drive or stop ($\mathsf{drive} \cup \mathsf{stop}$), followed by setting a timer $t := 0$ which, by (3.5), wakes the robot controller again after at most time T. When the test in (3.3) passes, it is safe to keep driving for T time, and the robot can choose any velocity $v := *$ up to at most the maximum velocity ($?0 \leq v \leq V$). In each case, the controller is allowed to stop the robot (3.4) by setting velocity v to 0. Finally, the plant (3.5) changes the distance according to the chosen velocity v via the differential equation $d' = -v$. Time advances at the rate $t' = 1$, for any duration $t \leq \mathsf{T}$. The program $\mathsf{ctrl};\mathsf{plant}$ can then repeat and the controller can make its next decision. The dL proof of formula (3.1) is elaborated next.

Proving Safety. The VeriPhy pipeline starts with a *safety proof* in dL of the partial correctness assertion

$$P \rightarrow [\{\mathsf{ctrl};\mathsf{plant}\}^*]Q \tag{3.6}$$

in KeYmaera X (Fulton et al., 2015).

The proof of formula (3.6) is input as a proof script for dL in the Bellerophon language (Fulton et al., 2017). Bellerophon scripts combine high-level automated search procedures from a standard library with manual uses of dL axioms (Platzer, 2008a, 2012c, 2017a). Typical scripts focus on key system insights, such as invariants for loops and differential equations, and manually assisting automation with challenging sub-problems, like proving statements about real arithmetic. For simpler models like $\{\mathsf{ctrl};\mathsf{plant}\}^*$, the proof is typically automatic once invariants are provided (Platzer, 2008a; Platzer & Tan, 2020). Interactive proofs from the web-based UI can also be exported to the Bellerophon (Mitsch & Platzer, 2016a) proof script language, then passed to VeriPhy. The VeriPhy pipeline

begins (leftmost column of Fig. 3.1) by checking the Bellerophon script to establish that the source model has been verified:

Definition 3.1 (Verified input)**.** The hybrid program $\alpha \equiv \{\mathsf{ctrl}; \mathsf{plant}\}^*$ for time-triggered ctrl is *verified* (with invariant J, precondition P, and postcondition Q) if a dL formula $P \to [\alpha]Q$ has been proven valid via a loop invariant J, i.e., $P \to J$, $J \to Q$ and $J \to [\alpha]J$ have been proven valid.

3.3 ModelPlex Sandbox Synthesis

To enable abstraction in controller models, dL provides features which make it ill-suited for direct execution, such as nondeterminism. Nondeterministic controller models are a natural fit, however, for *sandboxing* the results of an external unverified controller by monitoring it for compliance with the dL model and executing a safe *deterministic* fallback upon compliance violation. The second step of the VeriPhy pipeline (second column of Fig. 3.1) synthesizes from the system safety proof and loop invariant J such a *sandbox controller* enforcing runtime safety by sandboxing untrusted controllers. Correct-by-construction monitors detect controller bugs and environment model violations (Mitsch & Platzer, 2016b, Thms. 1+2), invoking verified fallback control or signaling an error, respectively.

The shape of the synthesized sandbox controller is shown in Fig. 3.3. The fixed sandbox controller shape reflects the fact that VeriPhy expects a fixed input model format: the input must be a single loop containing a controller and a plant, where the controller is optionally a choice ($\cup$) containing several control branches. For clarity, we denote by $\vec{x}$ the vector of all variables in the current program state before executing $\mathsf{ctrl}; \mathsf{plant}$, and denote by $\vec{x}^+$ the tentative next state. In all, the sandbox controller performs the following tasks: It *i)* nondeterministically assigns ($\vec{x} := *$) arbitrary values to configuration parameters and initial system state from external sensors in (3.7), checking that they satisfy the precondition P; *ii)* checks that the untrusted controller decision $\vec{x}^+$ (3.8) satisfies the monitor formula $\mathsf{ctrlMon}(\vec{x}, \vec{x}^+)$ in (3.9); *iii)* else enforces a safe fallback action (3.10); *iv)* actuates the decision $\vec{x}^+$ by assigning it to state $\vec{x}$ (3.11); *v)* models sensing with nondeterministic assignments $\vec{x}^+ := *$ and monitors whether the sensor values comply with the environment in (3.13), then stores them for the next iteration with $\vec{x} := \vec{x}^+$ (3.14).

Lines (3.9)–(3.10) correspond to a nondeterministic if-then-else statement where the else branch in (3.10) is always allowed. This flexibility becomes important in Section 3.4 when machine arithmetic introduces uncertainty in the test of (3.9).

We first discuss the key ingredients of sandboxing: the control monitor $\mathsf{ctrlMon}$ (3.9) for detecting errors in untrusted controllers and the plant monitor $\mathsf{plantMon}$ (3.13) for detecting unexpected environment behavior. We then discuss their incorporation into the verified sandbox controller (Fig. 3.3) with safe fallback control (3.10).

3.3.1 Controller Monitor Formula

We use nondeterminism in dL controller models to abstract away control algorithm details that are not safety-relevant (e.g., optimizations to save power or ensure smooth travel).

$$
\begin{array}{lll}
\mathsf{sandbox} \equiv & & \\
\quad \vec{x} := *;\ ?P; & \textit{read initial state} & (3.7) \\
\quad \{ & & \\
\quad\quad \vec{x}^+ := \mathsf{extCtrl}(\vec{x}); & \textit{run external control} & (3.8) \\
\quad\quad \big\{\ ?\mathsf{ctrlMon}(\vec{x}, \vec{x}^+) & \textit{check if safe action} & (3.9) \\
\quad\quad\quad \cup\ \vec{x}^+ := \mathsf{fallback}(\vec{x})\big\}; & \textit{or fallback control} & (3.10) \\
\quad\quad \vec{x} := \vec{x}^+; & \textit{actuate action} & (3.11) \\
\quad\quad \vec{x}^+ := *; & \textit{sense next state} & (3.12) \\
\quad\quad ?\mathsf{plantMon}(\vec{x}, \vec{x}^+); & \textit{check safe environment} & (3.13) \\
\quad\quad \vec{x} := \vec{x}^+ & \textit{store sensors} & (3.14) \\
\quad \}^* & \textit{repeat} & (3.15)
\end{array}
$$

Figure 3.3: Sandbox controller overview.

Any such details are supplied by the untrusted controller, which can be implemented freely, even in languages that were not designed for verification. The untrusted controller is only known to be safe, however, if it behaves consistently with the verified controller model. ModelPlex (Mitsch & Platzer, 2016b) synthesizes a real arithmetic formula $\mathsf{ctrlMon}(\vec{x}, \vec{x}^+)$ over the model's state variables to check control decisions for compliance with the model. The condition $\mathsf{ctrlMon}(\vec{x}, \vec{x}^+)$ is efficiently checked at runtime for concrete values $\vec{x}$ of a start state (e.g., distance sensed before the controller runs) and $\vec{x}^+$ of an end state (e.g., new speed, chosen by the controller).

We give a brief overview of monitor synthesis here, and refer the reader to the literature (Mitsch & Platzer, 2016b) for full details on how monitor formulas can be automatically synthesized from an input model and verified. ModelPlex composes the safety theorem $P \rightarrow [\{\mathsf{ctrl}; \mathsf{plant}\}^*]Q$ with offline transformation proofs (Mitsch & Platzer, 2016b, Lem 4–8), reducing system safety to online monitor compliance. ModelPlex monitors the precondition P when the system starts in state ω_0 (check $\omega_0 \models P$ in equation (3.7) of the sandbox) and the controller monitor condition $\mathsf{ctrlMon}(\vec{x}, \vec{x}^+)$ at every observed transition (ω, ν) (check $\mathsf{ctrlMon}(\vec{x}, \vec{x}^+)$ in equation (3.9)). As a result, we get online safety ($\nu \models J$ and thus $\nu \models Q$) up through the current state ν by (Mitsch & Platzer, 2016b, Thm 2).

Definition 3.2 (Compliance)**.** The transition (ω, ν) *complies* with formula $\mathsf{ctrlMon}(\vec{x}, \vec{x}^+)$, written $(\omega, \nu) \models \mathsf{ctrlMon}(\vec{x}, \vec{x}^+)$, iff $\mathsf{ctrlMon}(\vec{x}, \vec{x}^+)$ holds using the values of state ω for plain variables x and the values of ν for variables x^+ in $\vec{x}^+$. Formally, let μ be the unique state such that $\mu(x) = \omega(x)$ for all $\omega(x)$ and $\mu(x^+) = \nu(x)$ for all $\nu(x)$. Then $(\omega, \nu) \models \mathsf{ctrlMon}(\vec{x}, \vec{x}^+)$ iff $\mu \models \mathsf{ctrlMon}(\vec{x}, \vec{x}^+)$.

The equations in Fig. 3.4 illustrate the offline transformation proof[2] for synthesizing controller monitor conditions $\mathsf{ctrlMon}$ to check controller implementation correctness. The

[2]The offline transformation proofs are originally from ModelPlex (Mitsch & Platzer, 2016b).

proof starts at the semantic statement $(\omega, \nu) \in [\![\{\mathsf{ctrl}; \mathsf{plant}\}^*]\!]$ and obtains an arithmetic monitor condition $\mathsf{ctrlMon}(\vec{x}, \vec{x}^+)$. Let Dom stand for the evolution domain constraint of the plant, then condition $\mathsf{ctrlMon}(\vec{x}, \vec{x}^+)$ also checks that Dom holds so that the controller does not itself cause a plant violation upon actuating the output $\vec{x}^+$.

$(\omega, \nu) \in [\![\{\mathsf{ctrl}; \mathsf{plant}\}^*]\!]$	Semantical condition
$\Updownarrow$	by (Mitsch & Platzer, 2016b, Lem 4)
$(\omega, \nu) \models \langle\{\mathsf{ctrl}; \mathsf{plant}\}^*\rangle(\vec{x} = \vec{x}^+)$	Logical criterion
$\Uparrow$	by (Mitsch & Platzer, 2016b, Lem 5)
$(\omega, \nu) \models \langle\mathsf{ctrl}; \mathsf{plant}\rangle(\vec{x} = \vec{x}^+)$	thus $\nu \models Q$
$\Uparrow$	by (Mitsch & Platzer, 2016b, Lem 6)
$(\omega, \nu) \models \langle\mathsf{ctrl}\rangle(\vec{x} = \vec{x}^+ \wedge \mathsf{Dom})$	by ModelPlex-generated dL proof, Lemma 3.1
$\Uparrow$	
$(\omega, \nu) \models \mathsf{ctrlMon}(\vec{x}, \vec{x}^+)$	by online monitoring

Figure 3.4: ModelPlex controller monitor synthesis.

Monitor Correctness Proof. ModelPlex's synthesized controller monitor conditions are correct by construction (Mitsch & Platzer, 2016b) from the process in Fig. 3.4, which guarantees Lemma 3.1. The controller monitor synthesis process of Fig. 3.4 starts by obtaining logical criterion $\langle\{\mathsf{ctrl}; \mathsf{plant}\}^*\rangle(\vec{x} = \vec{x}^+)$ from the proved property $P \to [\{\mathsf{ctrl}; \mathsf{plant}\}^*]Q$. We denote by $\vec{x} = \vec{x}^+$ component-wise equality between vectors $\vec{x}$ and $\vec{x}^+$.

Lemma 3.1 (Controller monitor correctness)**.** *The controller monitor* $\mathsf{ctrlMon}(\vec{x}, \vec{x}^+)$ *relating control input* $\vec{x}$ *to control output* $\vec{x}^+$ *guarantees that control output* $\vec{x}^+$ *is permitted by the verified control model* ctrl *on input* $\vec{x}$ *and respects the* plant *evolution domain constraint* Dom*, that is:*

$$\models \mathsf{ctrlMon}(\vec{x}, \vec{x}^+) \to \langle \mathsf{ctrl} \rangle (\vec{x} = \vec{x}^+ \wedge \mathsf{Dom})$$

Lemma 3.1 is a crucial lemma in the sandbox safety proof.

Example. In our running example, the monitor checks the bound variables d, v, and t:

$$\begin{aligned}&\langle\mathsf{ctrl}\rangle(\vec{x} = \vec{x}^+ \wedge \mathsf{Dom}) \equiv \\ &\quad \Big\langle \{?d \geq \mathsf{T}V;\ v := *;\ ?0 \leq v \leq V \quad \cup \quad v := 0\}; \\ &\quad\ t := 0 \Big\rangle (d^+ = d \wedge v^+ = v \wedge t^+ = t \wedge t \leq \mathsf{T})\end{aligned}$$

The offline monitor transformation proofs are implemented as automation in KeY-

maera X, outside the trusted core. On the above formula, this monitor formula is output:

$$\begin{aligned}\mathsf{ctrlMon} \equiv \big(\big(d \geq \mathsf{T}V \wedge 0 \leq v^+ \leq V\big) \quad \vee \quad v^+ = 0\big)\\ \wedge 0 \leq \mathsf{T} \wedge V \geq 0 \wedge t^+ = 0 \wedge d^+ = d\end{aligned}$$

The monitor checks both possible paths through the controller: the first disjunct captures the test conditions for driving with a new velocity v^+ (nondeterministic assignment $v := *$ followed by test $?0 \leq v \leq V$), whereas the second disjunct captures the emergency stop ($v := 0$), so $v^+ = 0$. The conditions further state that the constants are chosen according to the model assumptions ($0 \leq \mathsf{T} \wedge V \geq 0$), that both paths reset their clocks $t^+ = 0$ to correctly measure the duration until the next controller run, and that neither controller path alters the distance measurement, so $d^+ = d$.

3.3.2 Plant Monitor Formula

ModelPlex also synthesizes a formula $\mathsf{plantMon}(\vec{x}, \vec{x}^+)$ which holds only if the values $\vec{x}$ and $\vec{x}^+$ sensed in successive states comply with the plant model. For example, the plant monitor for our ground robot checks that sensed motion is consistent with the maximum speed V.

Plant monitoring is a key reason why synthesis must exploit proof insights. A real implementation always has some uncertainty in timing, sensing, and actuation, so a monitor would be doomed to fail (i.e., raise an alarm) if it required sensed values to exactly match the solution of a differential equation. Proofs save us from over-restrictive monitors because proofs need not employ the exact trajectory, but rather often employ *invariant* arguments which specify a broader safety region. In our example, safety eschews the exact trajectory $d^+ = d - vt^+$ in favor of the looser invariant $d^+ \geq v(\mathsf{T} - t^+)$. It suffices for safety to construct plantMon from the plant model's evolution domain Dom (e.g., $t \leq \mathsf{T}$) and the ODE invariants in the safety proof of Step 1 (e.g., $d \geq v(\mathsf{T} - t)$). In the sandbox controller from Fig. 3.3, the condition $\mathsf{plantMon}(\vec{x}, \vec{x}^+)$ checks that the observed evolution from the sensed values $\vec{x}$ of the previous iteration to the new values $\vec{x}^+$ is within this relaxed safety region. If a plant monitor fails, a violation raises an alarm, upon which best-effort fallback control is typically done. Unlike in the ctrl monitor case, however, fallback controller safety cannot be guaranteed when all of the physical assumptions are violated.

Lemma 3.2 (Plant monitor correctness)**.** *Let* $\{\mathit{ctrl}; \mathit{plant}\}^*$ *be verified with invariant* J*, and let* $\mathsf{ctrlMon}(\vec{x}, \vec{x}^+)$ *be a correct controller monitor according to Lemma 3.1. Then, loop invariant* J *is preserved when the* plant *monitor* $\mathsf{plantMon}(\vec{x}, \vec{x}^+)$ *is satisfied.*

$$\vDash \mathsf{ctrlMon}(\vec{x}, \vec{x}^+) \rightarrow [?\mathsf{plantMon}(\vec{x}, \vec{x}^+)]J$$

To conclude the plant monitor formula discussion, we review what the plant monitor does and does not check. The plant monitor checks whether observed physical behavior at runtime complies with the model. Monitoring compliance is crucial to our goal of rigorously transferring safety guarantees from a verified model to an implementation.

If our goals had been different, one could also imagine other conditions whose monitoring could be of practical use in tasks such as debugging sensor code. Sanity conditions such as nonnegativity of distance ($d^+ \geq 0$) and bounded change in distance ($|d - d^+| \leq \delta$

for some bound δ) might help in practice to catch bugs in sensor drivers, for example. Because dL proofs about ODEs often happen to prove basic sanity conditions as lemmas, those conditions may often happen to appear as components of a plant monitor. Thus, plant monitors may incidentally detect sensor bugs when those bugs result in sensor values that could not possibly fit the model. In a similar spirit, plant monitors would alert us to the case where actuator disturbances are so great as to cause sensed data to diverge from the model (Section 3.6). Regardless, the core purpose of the plant monitor is to assess compliance between the implementation and the model.

3.3.3 Fallback Control

Unsafe control choices are detected by the controller monitor and replaced with provably safe fallback control choices. *Any* controller that satisfies the controller monitor can be used for safe fallback according to Lemma 3.3. Concretely, we take the verified fallback from the controller ctrl, e.g., $v := 0; t := 0$ for our ground robot in this example. In general, VeriPhy allows the fallback controller to be specified explicitly as an input to the pipeline.

Lemma 3.3 (Fallback correctness)**.** *Let program* $\{\mathit{ctrl}; \mathit{plant}\}^*$ *be verified with loop invariant* J *and let* $\mathsf{ctrlMon}(\vec{x}, \vec{x}^+)$ *be a controller monitor per Lemma 3.1. A fallback controller is correct if* $\mathsf{ctrlMon}(\vec{x}, \vec{x}^+)$ *holds at the end whenever* J *holds initially.*

$$\vDash J \rightarrow [\vec{x}^+ := \mathsf{fallback}(\vec{x})]\mathsf{ctrlMon}(\vec{x}, \vec{x}^+)$$

3.3.4 Provably Safe Sandboxing

Theorem 3.4 says safety results transfer: from the theorem $P \rightarrow [\{\mathsf{ctrl}; \mathsf{plant}\}^*]Q$ for program $\{\mathsf{ctrl}; \mathsf{plant}\}^*$, we obtain safety of the synthesized sandbox with the same postcondition Q. Theorem 3.4 is proven anew for each model using proof automation implemented in KeYmaera X. The proof reuses invariants which are mined from the safety proof of the input model by executing that proof. In the best case, this approach allows sandbox safety proofs for systems where a fully automated proof would be difficult.

However, classical VeriPhy also reveals one of its fundamental limitations in the process of exploiting safety proofs of input models. In practice, classical VeriPhy often fails with cryptic error messages when input models fall outside rigid fixed formats or when the shape of the input model safety proof differs from the shape of the input model. Two root problems underlie this limitation. Firstly, the modeling and proof language were not designed to ensure, from first principles, that code can be synthesized from *all* provable systems. Secondly, models and proofs are separate artifacts, so it is not immediately obvious which proof steps apply to which model statements when consuming a model and proof in a tool like classical VeriPhy. These root problems are addressed in Part II and Part III respectively by ensuring proofs correspond to code and by developing an integrated language for models and proofs which serves as the basis for a reimplementation of VeriPhy.

Limitations aside, the transfer of formal safety guarantees to implementation-level using sandboxes is a crucial contribution of classical VeriPhy.

Theorem 3.4 (Sandbox safety)**.** *Let program* $\{\mathsf{ctrl};\mathsf{plant}\}^*$ *be verified (Def. 3.1) with some precondition* P *and postcondition* Q*. Assume a correct controller monitor, correct* plant *monitor, and correct fallback. Then all runs of the* sandbox *program (from Fig. 3.3) starting in* P *(from Def. 3.1) are safe (*Q*):*

$$\vDash P \rightarrow [\mathsf{sandbox}]Q$$

Moreover, sandbox *is verified with the same invariant precondition, postcondition, and invariant as* $\{\mathsf{ctrl};\mathsf{plant}\}^*$*.*

Proof. By dL proof from Lemma 3.1, Lemma 3.2, and Lemma 3.3. The tactic for the dL proof is outlined in Section 3.8.3. □

Running Example. The provable dL formula in Fig. 3.5 illustrates the controller and plant monitor conditions of our running example embedded into their sandbox.

$$
\begin{array}{l}
\text{see (3.1): } d \geq 0 \land V \geq 0 \land \mathsf{T} \geq 0 \\
\boxed{P} \rightarrow \big[V := *;\ \mathsf{T} := *;\ d := *;\ t := *; \quad // \ \vec{x} := * \\
\qquad ?d \geq 0 \land V \geq 0 \land \mathsf{T} \geq 0; \quad // \ ?P \\
\qquad \big\{\ t^+ := *;\ v^+ := *;\ d^+ := d; \quad // \ \vec{x}^+ := \mathsf{extCtrl} \\
\qquad\quad \{\quad ?\mathsf{ctrlMon}(d,t,v,d^+,t^+,v^+\} \\
\qquad\qquad \cup\, t^+ := 0;\ v^+ := 0\ \big); \quad // \ \vec{x}^+ := \mathsf{fallback} \\
\qquad\quad t := t^+;\ v := v^+; \quad // \ \vec{x} := \vec{x}^+ \qquad (\mathit{actuate}) \\
\qquad\quad d^+ := *;\ t^+ := *; \quad // \ \vec{x}^+ := * \qquad (\mathit{sense}) \\
\qquad\quad ?\big(0 \leq t^+ \leq \mathsf{T} \land d^+ \geq v(\mathsf{T} - t^+)\big); \quad // \ ?\mathsf{plantMon}(\vec{x},\vec{x}^+) \\
\qquad\quad d := d^+;\ t := t^+ \quad // \ \vec{x} := \vec{x}^+ \qquad (\mathit{store}) \\
\qquad \big\}^*\big] \boxed{Q} \\
\qquad\quad \text{see (3.1) : } d \geq 0
\end{array}
$$

Figure 3.5: Sandbox of a velocity-controlled ground robot.

Truth of the monitor formula implies runtime safety of the CPS, but the monitor formulas and sandboxes are hard to execute until we concretely implement the arithmetic operations and nondeterministic approximations contained therein.

3.4 Interval Word Arithmetic Translation

Having shown safety of the sandbox controller in dL, we turn our attention toward correct compilation, the first step of which is to formally justify implementing real numbers with interval arithmetic over machine words. We formalize both real arithmetic and interval arithmetic semantics for dL in an extension of the soundness formalization from Section 2.3.

We show an arithmetic soundness theorem: any formula which holds in the interval semantics holds in the real-number semantics. The opposite direction does not hold because interval arithmetic is more conservative, meaning that many valid formulas of dL (e.g. $x \leq 0 \vee x > 0$) are not valid over the interval semantics.

The formalization presented here is done in Isabelle/HOL[3] because semantic arguments about dL are outside of KeYmaera X's purview. Because we wish to soundly combine results from multiple theorem provers, it is important that the provers are trustworthy and that shared definitions have the same meaning in each prover. For this reason, we remove the KeYmaera X prover core from the trusted base by using the verified proofchecker implemented in Section 2.8. See Section 2.8 for details and for the process of exporting proof terms from KeYmaera X for cross-checking. The checker supports all sandbox safety proofs in this chapter, on the scale of $\approx 10^5$ proof steps.

We have soundly transitioned from proofs of system safety in KeYmaera X to the truth of system safety according to the semantics of dL in Isabelle/HOL. The standard semantics of dL feature arithmetic on real numbers, which are crucial for physics but ill-suited to efficient execution. Next, we soundly approximate the real semantics with a computable 32-bit integer interval arithmetic semantics, enabling efficient sandbox execution. Here, we present our translation to interval arithmetic and prove it sound in Isabelle/HOL.

The work presented here only formalizes interval arithmetic for a fragment of dL: nondeterministic assignment, ODEs, loops, division, modalities, and quantifiers are omitted. Modalities, quantifiers, and ODEs are omitted because we expect they have been eliminated before interval arithmetic is applied. Division is omitted because interval bounds for division are often so conservative that rewriting models to use multiplication (without changing their meaning) often results in tighter bounds. Loops and nondeterministic assignment are omitted for a more subtle reason. In the final compilation step, the CakeML sandbox program (Section 3.5) uses hard-coded looping and nondeterministic assignment which are handled as a special case in its correctness proof, rather than employing the hybrid program connectives for loops and nondeterministic assignments.

The major design choice for the present stage of the VeriPhy pipeline is arithmetic representation. We wish to keep compilation simple, support a wide variety of hardware, and keep the arithmetic soundness proof simple. We chose fixed-precision integer (interval) arithmetic because it is widely used in embedded software for its predictability and is universally supported by hardware and compilers. Fixed-precision integer arithmetic is conservative because of the limited precision available. Our evaluations show that our limited precision is sufficient in the 1D example (Section 3.6) and can be made sufficient in the 2D example (Section 3.7) through careful manual choice of physical units. One lesson learned (Section 3.8) is that fixed-precision arithmetic becomes a more significant limitation as model complexity increases.

[3]The Isabelle/HOL formalization was ultimately ported to HOL4 in order to streamline the correctness argument. The Isabelle/HOL version is presented here because the HOL4 formalization is a port of the author's work in Isabelle/HOL.

Semantics. The transition to interval arithmetic does not require transforming the program source; we merely assign a new semantics to the existing constructs of dL. Helper functions for the semantics are listed in Fig. 3.6, the term semantics are listed in Fig. 3.7, formulas semantics are in Fig. 3.8, and program semantics are in Fig. 3.9. We write ω_I, ν_I for *interval states* assigning to each variable x an interval $[\ell, u]$ of 32-bit machine words for lower and upper bounds on the (real number) value of x, respectively. Machine words are interpreted as signed integers in standard two's-complement format, excepting sentinel values for negative (∞_w^-) and positive (∞_w^+) infinity. Truth is written $\top$, falsehood is written $\bot$, and uncertainty is written U.

We write $\omega_I[\![f]\!] : [\overline{\mathbb{R}}, \overline{\mathbb{R}}]$ for the value of term f in the interval state ω_I, which is an interval in the extended reals where any finite endpoints are closed endpoints. Likewise, we write $(\omega_I, \nu_I) \in [\![\alpha]\!]$ when interval state ω_I can reach ν_I upon running hybrid program α. Because interval arithmetic is conservative, the resulting formula semantics is three-valued: we write $\omega_I[\![P]\!] = \top$ when P is definitely true in interval state ω_I, $\bot$ when it is definitely false, or U when it is unknown. The author's free-logic treatment of dL with definite description (Bohrer, Fernández, & Platzer, 2019) is related to our interval semantics in the sense that both are three-valued. In the former, the third truth value is used to capture formulas containing terms that are not defined in every (e.g., real-valued) state, while we, the latter, use the third truth value to represent formulas whose values are unknown because some terms are *under*-defined, i.e., the value of the term is an interval which is too broad to ensure a definite truth value for a given formula.

$$
\begin{aligned}
\mathrm{trunc}(w) = {} & \max(\infty_w^-, \min(\infty_w^+, w)) \\
(-_w)w_1 = {} & \textbf{if } (w_1 = \infty_w^-) \textbf{ then } \infty_w^+ \textbf{ else if } (w_1 = \infty_w^+) \textbf{ then } \infty_w^- \textbf{ else } -w_1 \\
w_1 \hat{+}_w w_2 = {} & \textbf{if } \max(w_1, w_2) \in \{\infty_w^+, \infty_w^-\} \textbf{ then } \max(w_1, w_2) \textbf{ else } \mathrm{trunc}(w_1 + w_2) \\
w_1 \check{+}_w w_2 = {} & \textbf{if } \min(w_1, w_2) \in \{\infty_w^+, \infty_w^-\} \textbf{ then } \min(w_1, w_2) \textbf{ else } \mathrm{trunc}(w_1 + w_2) \\
w_1 *_w w_2 = {} & \textbf{if } (w_1 = 0 \vee w_2 = 0) \textbf{ then } 0 \\
& \textbf{else if } \mathtt{is}\infty^+{}_*(w_1, w_2) \textbf{ then } \infty_w^+ \\
& \textbf{else if } \mathtt{is}\infty^-{}_*(w_1, w_2) \textbf{ then } \infty_w^- \\
& \textbf{else } \mathrm{trunc}(w_1 * w_2) \\
[\ell_1, u_1] \check{*}_w [\ell_2, u_2] = {} & \mathtt{min}_w(*_w(\ell_1, \ell_2), *_w(u_1, \ell_2), *_w(\ell_1, u_2), *_w(u_1, u_2)) \\
[\ell_1, u_1] \hat{*}_w [\ell_2, u_2] = {} & \mathtt{max}_w(*_w(\ell_1, \ell_2), *_w(u_1, \ell_2), *_w(\ell_1, u_2), *_w(u_1, u_2))
\end{aligned}
$$

Figure 3.6: Interval arithmetic for executable dL, helper functions.

In Fig. 3.6, operation $\mathrm{trunc}(w)$ returns the argument w if it is in range, else positive or negative infinity when w is out of range. We implement bounds checking by sign-extending to 64-bit words, where our operations on 32-bit values are guaranteed not to overflow, and then checking the result. This is done, e.g., in $\hat{+}_w$ and $\check{+}_w$, (casts between 32- and 64-bit words are omitted for brevity). Rounding modes differ in handling of infinite inputs, e.g., $\infty_w^- + \infty_w^+$ is indeterminate, bounded below only by ∞_w^-, and bounded above

only by ∞^+_w. Arithmetic operations augmented with overflow checks are written with a subscript $_w$, such as $-_w$. The helper functions $\hat{+}_w$ and $\check{+}_w$ apply addition with upward and downward rounding, respectively. Likewise, $\hat{*}_w$ and $\check{*}_w$ perform multiplication with upward and downward rounding, but they accept two intervals as arguments rather than two words because both the upper and lower bounds of multiplication depend on the upper and lower bounds of each operand as inputs. The upper and lower multiplication bounds are written in terms of $*_w$, which performs exact multiplication with bounds checking on both the inputs and outputs. The functions $\texttt{is}\infty^+{}_*$ and $\texttt{is}\infty^-{}_*$ stand in for the input bounds checks: $\texttt{is}\infty^+{}_*(w_1, w_2)$ holds if the product of w_1 and w_2 is positive infinity, while $\texttt{is}\infty^-{}_*(w_1, w_2)$ holds when the product is negative infinity. We do not list their full definitions because each one consists of a large number of uninformative cases.

In Fig. 3.7, we give the definitions for the fragment of dL programs whose interval semantics have been formalized in Isabelle/HOL. The value of literal q is the singleton interval $[q, q]$ and the value of variable x is the interval $\omega_I(x)$. The semantics of each arithmetic operator is reduced to the helper functions from Fig. 3.6. Note that division is not formalized in the present release, one reason being that the most common uses of multiplication in dL can be rewritten using multiplication and a second reason being that the interval bounds for division are often far more conservative than the corresponding bounds for multiplication.

$$
\begin{aligned}
\omega_I[\![q]\!] &= [q, q] \\
\omega_I[\![x]\!] &= \omega_I(x) \\
\omega_I[\![f_1 + f_2]\!] &= [\ell_1 \check{+}_w \ell_2, u_1 \hat{+}_w u_2] \text{ where } \omega_I[\![f_i]\!] = [\ell_i, u_i] \\
\omega_I[\![f_1 * f_2]\!] &= [([\ell_1, u_1] \check{*}_w [\ell_2, u_2]), ([\ell_1, u_1] \hat{*}_w [\ell_2, u_2])] \text{ where } \omega_I[\![f_i]\!] = [\ell_i, u_i] \\
\omega_I[\![\texttt{max}(f_1, f_2)]\!] &= [\texttt{max}_w(\ell_1, \ell_2), \texttt{max}_w(u_1, u_2)] \text{ where } \omega_I[\![f_i]\!] = [\ell_i, u_i] \\
\omega_I[\![\texttt{min}(f_1, f_2)]\!] &= [\texttt{min}_w(\ell_1, \ell_2), \texttt{min}_w(u_1, u_2)] \text{ where } \omega_I[\![f_i]\!] = [\ell_i, u_i] \\
\omega_I[\![-(f)]\!] &= [-_w u, -_w \ell] \text{ where } \omega_I[\![f]\!] = [\ell, u] \\
\omega_I[\![\texttt{abs}(f)]\!] &= [\texttt{max}_w(\ell, -_w u), \texttt{max}_w(u, -_w \ell)] \text{ where } \omega_I[\![f]\!] = [\ell, u]
\end{aligned}
$$

Figure 3.7: Interval arithmetic for executable dL, terms.

The formula semantics are given in Fig. 3.8 on the next page. The semantics for comparison formulas say that the comparison formula is true ($\top$) if all pairs of elements from respective intervals satisfy the comparison, false ($\bot$) if no pairs satisfy it, or unknown (U) if some satisfy it and others do not. For example, the strict inequality $x <_w x \check{+}_w y$ could be either true or false in the state $\nu_I = \{x \mapsto [1, 2], y \mapsto [0, 1]\}$, so the conservative truth value is U. In contrast, the truth value of the nonstrict inequality $x \leq_w x \check{+}_w y$ is $\top$. Exact equalities are true ($\top$) only when both intervals are the same singleton interval. The propositional connective semantics are optimistic in the sense that a formula can be true or false even when one subformula is unknown. For example, $\phi \wedge \psi$ is $\bot$ when ϕ is U and ψ is $\bot$, because a false formula remains false when conjoined with any other formula.

$$\omega_I[\![f_1{>}f_2]\!] = \begin{cases} \top & \text{if } \omega_I[\![f_i]\!] = (\ell_i, u_i) \text{ and } \ell_1 > u_2 \\ \bot & \text{if } \omega_I[\![f_i]\!] = (\ell_i, u_i) \text{ and } u_1 \leq \ell_2 \\ U & \text{otherwise} \end{cases}$$

$$\omega_I[\![f_1{\geq}f_2]\!] = \begin{cases} \top & \text{if } \omega_I[\![f_i]\!] = (\ell_i, u_i) \text{ and } \ell_1 \geq u_2 \\ \bot & \text{if } \omega_I[\![f_i]\!] = (\ell_i, u_i) \text{ and } u_1 < \ell_2 \\ U & \text{otherwise} \end{cases}$$

$$\omega_I[\![f_1{=}f_2]\!] = \begin{cases} \top & \text{if } \omega_I[\![f_1]\!] = \omega_I[\![f_2]\!] = (\ell, u) \text{ and } \ell = u \\ \bot & \text{if } \omega_I[\![f_i]\!] = (\ell_i, u_i) \text{ and } (\ell_1 > u_2 \text{ or } \ell_2 > u_1) \\ U & \text{otherwise} \end{cases}$$

$\wedge$	$\top$	U	$\bot$
$\top$	$\top$	U	$\bot$
U	U	U	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

$\vee$	$\top$	U	$\bot$
$\top$	$\top$	$\top$	$\top$
U	$\top$	U	U
$\bot$	$\top$	U	$\bot$

ϕ	$\top$	U	$\bot$
$\neg\phi$	$\bot$	U	$\top$

Figure 3.8: Interval arithmetic for executable dL, formulas.

Likewise, $\phi \vee \psi$ is $\top$ when ϕ is U and ψ is $\top$, because a true formula remains true when disjoined with any other formula. Semantics for first-order quantifiers are not given because we wish to present an executable fragment and quantifiers over uncountable sets are difficult to execute. Instead, we expect that real quantifier elimination (G. E. Collins & Hong, 1991) has been applied to first-order real arithmetic formulas beforehand. Assuming that implication is defined as $\phi \to \psi \equiv \psi \vee \neg\phi$, then our interpretation of the propositional connectives agrees with Kleene's logic K_3 (Kleene, 1938).

$$\begin{array}{ll} (\omega_I, \nu_I) \in [\![x := f]\!] & \text{iff } \nu_I = \omega_I \text{ except } \nu_I(x) = \omega_I[\![f]\!] \\ (\omega_I, \nu_I) \in [\![?\phi]\!] & \text{iff } \omega_I = \nu_I \text{ and } \omega_I[\![\phi]\!] = \top \\ (\omega_I, \nu_I) \in [\![\alpha \cup \beta]\!] & \text{iff } (\omega_I, \nu_I) \in [\![\alpha]\!] \text{ or } (\omega_I, \nu_I) \in [\![\beta]\!] \\ (\omega_I, \nu_I) \in [\![\alpha; \beta]\!] & \text{iff } (\omega_I, \mu_I) \in [\![\alpha]\!] \text{ and } (\mu_I, \nu_I) \in [\![\beta]\!] \end{array}$$

Figure 3.9: Interval arithmetic for executable dL, programs.

The program semantics are in Fig. 3.9. Tests ($?\phi$) are conservative in the sense that they only pass when test conditions (ϕ) are definitely true ($\top$), i.e., they treat unknown (U) formulas the same as definitely false ($\bot$) ones. Tests collapse 3-valued truth into 2-valued truth because they must: control flow must decided conclusively in order to decide which statements are executed, unlike the formula semantics, which admit unknown values (U). The semantics of assignment, choice, and sequential composition are written similarly to their standard definitions in the real-valued semantics of dL. The most notable change

regarding these constructs is the use of interval states. For example, in the semantics of $x := f$, the value of f is an interval, which is assigned as the value of x in the final state ν_I.

Relating Real and Interval Semantics. We now formalize our notion of correctness: the interval semantics is sound with respect to real-number semantics if all word intervals contain their corresponding real numbers. Formally, we define a notation $\nu \in (\!|\nu_I|\!)$ meaning the values of all variables in interval state ν_I contain their correspondents in real-number state ν. We likewise define the notation $r \in (\!|w_\ell, w_u|\!)$ to mean the real number r is in the interval $[w_\ell, w_u]$. We first reduce the two-sided bounds to one-sided bounds ($w_\ell \overset{\text{wr}}{\leq} r$ and $w_u \overset{\text{wr}}{\geq} r$) before defining the one-sided bounds:

$$\begin{aligned} r \in (\!|w_\ell, w_u|\!) \text{ iff } w_\ell \overset{\text{wr}}{\leq} r \text{ and } w_u \overset{\text{wr}}{\geq} r \\ \omega \in (\!|\omega_I|\!) \text{ iff } \omega(x) \in (\!|\omega_I(x)|\!) \text{ for all } x \in \mathcal{V} \end{aligned}$$

We now give the one-sided safe bounds $w \overset{\text{wr}}{\leq} r$ and $w \overset{\text{wr}}{\geq} r$ meaning word w is a lower or upper bound for real r, respectively:

$$\begin{aligned} w \overset{\text{wr}}{\leq} r \text{ iff } w \overset{\text{wr}}{=} r' \text{ for some } r' \leq r \\ w \overset{\text{wr}}{\geq} r \text{ iff } w \overset{\text{wr}}{=} r' \text{ for some } r' \geq r \end{aligned}$$

The decomposition into one-sided bounds will be useful in the Isabelle/HOL proofs because it allows us to build lemmas about two-sided bounds from simpler lemmas about one-sided bounds. Inexact one-sided bounds are defined in terms of exact bounds $w \overset{\text{wr}}{=} r$ meaning word w exactly represents real r. Here, `w2r` is the standard injection of two's-complement words into reals:

$$\begin{aligned} \infty_w^+ \overset{\text{wr}}{=} r \text{ iff } r \geq \texttt{w2r}(\infty_w^+) \\ \infty_w^- \overset{\text{wr}}{=} r \text{ iff } r \leq \texttt{w2r}(\infty_w^-) \\ w \overset{\text{wr}}{=} \texttt{w2r}(w) \text{ otherwise} \end{aligned}$$

Soundness. We use the above definitions to state and prove soundness theorems that conservatively relate the interval semantics to the real semantics.

Theorem 3.5 (Soundness for terms)**.** *Interval valuations of terms contain their real valuations. That is, if $\omega[\![f]\!] = r$ and $\omega \in (\!|\omega_I|\!)$ then $r \in \omega_I(\!|f|\!)$.*

Proof. In this proof and throughout many proofs in this thesis, we give facts names in parentheses so that we may refer to them by name later on.

By induction on f. We give the representative case $f = f_1 + f_2$. In this case, assume $\omega \in (\!|\omega_I|\!)$ and $\omega[\![\theta_1 + \theta_2]\!] = r$ for some $r \in \mathbb{R}$. Expand the semantics of $+$ to get some r_1 and r_2 such that $r = r_1 + r_2$ and each $\omega[\![\theta_i]\!] = r_i$. By the IHs, we have (InI) each $r_i \in \omega_I(\!|\theta_i|\!)$. Let $[\ell_1, u_1]$ and $[\ell_2, u_2]$ be the intervals returned by $\omega_I(\!|\theta_1|\!)$ and $\omega_I(\!|\theta_2|\!)$, respectively. Then (InI) expands to (LI) $\ell_i \overset{\text{wr}}{\leq} r_i$ and (RI) $u_i \overset{\text{wr}}{\geq} r_i$.

In order to show $r \in \omega_I[\![\theta_1 + \theta_2]\!]$, it suffices by definition of $\omega_I[\![\theta_1 + \theta_2]\!]$ to show $r \in [\ell_1 \check{+}_w \ell_2, u_1 \hat{+} u_2]$, so by expanding definitions it suffices to show (LR) $\ell_1 \check{+}_w \ell_2 \overset{\text{wr}}{=} r_1 + r_2$ and (RR) $u_1 \hat{+}_w u_2 \overset{\text{wr}}{=} r_1 + r_2$.

Facts (LR) and (RR) both follow by correctness of rounding modes, the former by assumptions (LI) and the latter by assumptions (RI). Specifically, correctness of rounding means upward-rounding operations preserve upper bounds and downward-rounding operations preserve lower bounds. Our Isabelle/HOL formalization proves rounding correctness properties as lemmas because, to our knowledge, they are not provided by existing Isabelle/HOL arithmetic libraries (L. Yu, 2013).

For example, the upper bound lemma for addition says that if $w_1 \overset{\text{wr}}{\geq} r_1$ and $w_2 \overset{\text{wr}}{\geq} r_2$ then $w_1 \hat{+}_w w_2 \overset{\text{wr}}{\geq} r_1 + r_2$. The proof is by cases, following the structure of the definition of $w_1 \hat{+}_w w_2$. When bound checks detect overflow, they soundly return infinities. When w_1 and w_2 are both finite and bounds checks pass, it suffices to show that the casts are sound, which they are. □

Theorem 3.6 (Soundness for formulas)**.** *If the interval semantics of a formula is true or false, the real semantics agree.*

- *If* $\omega_I[\![P]\!] = \top$ *and* $\omega \in [\![\omega_I]\!]$ *then* $\omega \models P$.
- *If* $\omega_I[\![P]\!] = \bot$ *and* $\omega \in [\![\omega_I]\!]$ *then* $\omega \not\models P$.
- *If* $\omega_I[\![P]\!] = U$ *we make no claim.*

Proof. By induction on the proof of $\omega_I \in [\![P]\!]$, and by Theorem 3.5. We show a representative case $P \equiv (f_1 < f_2)$. Comparisons $f_1 < f_2$ bridge terms to formulas. For soundness, we conservatively compare the *upper* bound of f_1 with the *lower* bound of f_2, and consider comparisons of overlapping interval undefined (U). Formally: If $w_1 \overset{\text{wr}}{\geq} r_1$ and $w_2 \overset{\text{wr}}{\leq} r_2$ and $w_1 <_w w_2$ then $r_1 < r_2$. The proof is direct, by the definitions of $\overset{\text{wr}}{\leq}, \overset{\text{wr}}{\geq}, \overset{\text{wr}}{=}$, and $<_w$. □

Theorem 3.7 (Soundness for programs)**.** *If* $(\omega_I, \nu_I) \in [\![\alpha]\!]$ *and* $\omega \in [\![\omega_I]\!]$*, then there exists* $\nu \in [\![\nu_I]\!]$ *where* $(\omega, \nu) \in [\![\alpha]\!]$.

Proof. By induction on programs α, using Theorem 3.6. □

Together, these theorems show that all program behaviors accepted by the sandbox program in interval semantics correspond to behavior in the real semantics, which is safe by Theorem 3.4:

Corollary 3.8 (Sandbox soundness)**.** *Let program* $\{\mathsf{ctrl}; \mathsf{plant}\}^*$ *be verified (Def. 3.1) with some precondition* P *and postcondition* Q*. Assume sensing is sound (*$\omega \in [\![\omega_I]\!]$*) and assume the sandbox controller program has at least one program transition starting from* ω_I *(*$\omega_I[\![P]\!] = \top$ *and* $(\omega_I, \nu_I) \in [\![\mathsf{sandbox}]\!]$*). Then there is a real state* ν *underlying the final interval state (*$\nu \in [\![\nu_I]\!]$*) which is safe, i.e.,* $\nu \in [\![Q]\!]$.

Proof. By assumption, $\{\mathsf{ctrl}; \mathsf{plant}\}^*$ is verified with precondition P, postcondition Q, and invariant J. By Theorem 3.4, we have that sandbox is verified with the same formulas, so the implications (A) $P \to J$, (B) $J \to [\mathsf{sandbox}]J$, and (C) $J \to Q$ are valid in the real-valued dL semantics by soundness of dL.

Sensing soundness means the initial interval state contains the initial real state ($\omega \in [\![(\omega_I)]\!]$). By Theorem 3.6 on assumption $\omega_I[\![P]\!] = \top$ we have $\omega \in [\![P]\!]$. By (A) we have (Inv) $\omega \in [\![J]\!]$. By Theorem 3.7 there exists (Contained) a $\nu \in [\![(\nu_I)]\!]$ where $(\omega, \nu) \in [\![\mathsf{sandbox}]\!]$. Then by (Inv) and (B) we have $\nu \in [\![J]\!]$, which together with (Contained) is what we wanted to prove. □

Now the sandbox is executable at a high level and still safe. Next, we will soundly implement the executable interval semantics at the machine level.

3.5 Sandbox Implementation in CakeML

The sandbox program (Fig. 3.3) can now be understood with interval semantics, by Corollary 3.8. Intervals help implement the monitoring checks (3.9) and (3.13) using machine arithmetic. However, the sandbox still contains high-level abstract constructs. Nondeterministic assignments model sensing in (3.7) and (3.12) and external control in (3.8). Control branching (between (3.9) and fallback (3.10)) and looping are nondeterministic.

In this section, we explain how the sandbox is implemented as a CakeML program (Section 3.5.1). We resolve the aforementioned sources of nondeterminism, external controller calls, and actuators, all with CakeML's support for *foreign function interfaces* (*FFIs*). The resulting program is then compiled down to machine code (ARMv6, x64, etc.) using the verified CakeML compiler (Tan et al., 2016).

By employing the verified CakeML compiler, we know that the compiled machine code soundly implements CakeML source programs. It remains to show (Section 3.5.3) that our CakeML program soundly implements the sandbox. This verification step is made easier because CakeML is itself a high-level programming language with an accompanying suite of verification tools (Myreen & Owens, 2012; Guéneau et al., 2017). The CakeML program, however, senses and actuates in the real world, so its soundness relies on assumptions about the correctness of sensor and actuator FFIs (Section 3.5.2).

3.5.1 CakeML Sandbox

We first explain how we implement nondeterminism and external interaction. The following pseudocode snippet illustrates the sandbox loop implementation. Lines are numbered on the left with corresponding equation numbers from Fig. 3.3. The pseudocode uses . notation for fields of the state, where fields ending in ⁺ model variables of form x^+.

```
fun cmlSandboxBody state =
   if not (stop ()) then
     state.ctrl⁺ := extCtrl state;
     state.ctrl := if intervalSem ctrlMon state = ⊤
                      then state.ctrl⁺
                      else fallback state;
     actuate state.ctrl;
     state.sensors⁺ := sense ();
     if intervalSem plantMon state = ⊤ then
```

```
      Runtime.fullGC ();
      state.sensors := state.sensors⁺;
      cmlSandboxBody state
    else violation "Plant Violation"
```

The tail-recursive function `cmlSandboxBody` keeps track of a CakeML representation of the current state (`state`). We use field-assignment notation for `state`, closely following the assignments in Fig. 3.3. Loop termination is decided by the `stop` FFI wrapper. The `stop` wrapper itself makes an FFI call to external code (`ffiStop`) which decides whether to stop the loop, e.g., upon user request or battery depletion.

Nondeterministic assignments for external control and actuation are implemented by `extCtrl` and `sense`, which are FFI wrappers around external drivers. From the current state variable vector $\vec{x}$, we single out the sensor variables (`state.sensors`), actuated variables (`state.ctrl`), and constants (`state.consts`); $\vec{x}^+$ is treated likewise.

The `actuate` FFI wrapper executes the control decision `state.ctrl`, taken from `extCtrl` when the controller monitor `ctrlMon` is satisfied or the `fallback` otherwise. Both `extCtrl` and `fallback` take `state` as their argument because the control decisions which they output are allowed to depend on the entire state.

The above nondeterminism came from the environment and was thus resolved externally with FFIs. The nondeterministic choice between (3.9) and (3.10), in contrast, simply provides us freedom in controller implementation. We exploit this freedom when the ternary truth-value of `ctrlMon` in the interval semantics (`intervalSem`) is unknown (U): we are free to use the fallback (3.10) even when `ctrlMon` is not definitely false ($\bot$), so we conservatively use it in the unknown (U) case as well.

If the *plant* monitor fails, however, sandboxing can no longer guarantee safety. Here, the function `cmlSandboxBody` exits the control loop by calling the `violation` function with an error message. The function returns control to user code, which may initiate (unverified) best-effort recovery measures in the case of plant violations. For time-triggered controllers, the plant monitor only holds when the system delay is within our specified limit T. We minimize the risk of a delay violation by garbage-collecting (`Runtime.fullGC`) after each cycle, making runtimes predictable. The cost is negligible in practice because each control cycle does minimal heap allocation.

The sandbox entry point `cmlSandbox`, elided here, simply invokes `cmlSandboxBody` on the initial state after checking initial conditions P. We have thus reduced implementation of the sandbox to implementation of the FFIs.

3.5.2 CakeML FFIs

We now specify and implement the FFIs. FFIs bridge CakeML to the external world: physical sensing/actuation and untrusted control. Thus, the crucial specification step is to model external behavior in HOL4: We write `es:ext` for an *external state*, a record capturing all external state and effects, current and future. The external state is taken as the ground truth of the world, which all (e.g. sensing/actuation) FFIs must read or change. This makes the notion of sensing/actuation correctness precise. The 6 FFIs assumed by

Table 3.1: External functions and their intended meaning.

External func.	Intended Meaning
`ffiConst`	Get the values of system constants
`ffiSense`	Get the current sensor readings
`ffiExtCtrl`	Get the next (untrusted) control decision
`ffiActuate`	Actuate a control decision
`ffiStop`	Check whether to run more control cycles
`ffiViolation`	Exit control loop due to a fatal violation

VeriPhy are summarized in Table 3.1, along with their informal meanings. Of these, we take `ffiSense` and `ffiStop` as our examples.

FFI Model. Each FFI specification consults the external state to determine the ground truth of the current state and effect of external code. They each return a result `r` and a new external state `es` when invoked safely (i.e. `SOME(r,es')`) or `NONE` if calling conventions were violated. For example, the `ffiSense` calling convention expects one word per sensor in the array *bytes*, then we specify that the values sensed by `ffiSense` match the ground truth `es.sensor_vals`. In `ffiSense`, `NSENSORS` is the number of sensor variables (i.e., elements of `state.sensors`) and `word_to_bytes` converts the sensor values from machine words to a byte array as required by the CakeML FFI interface.

```
ffiSense bytes (es:ext) =
  if LEN bytes = NSENSORS*WORD ∧
     LEN es.sensor_vals = LEN bytes
  then SOME (word_to_bytes es.sensor_vals, es)
  else NONE
```

Unlike `ffiSense`, which must always return the current sensor values, `ffiStop` has complete freedom to decide when the loop should stop. We assume that it eventually stops, but we make no assumptions regarding when exactly it stops. In HOL4, this scenario is modeled by querying an oracle (`es.stop_oracle`). The HOL4 oracle feature allows us to model values that are entirely arbitrary (i.e., unknown to us), except that the oracle eventually tells us to stop:

```
ffiStop bytes (es: ext) =
  if LEN bytes = 1
  then SOME (query es.stop_oracle, es)
  else NONE
```

When queried, the oracle returns a bit, with `1` telling the sandbox loop to stop.

Neither `ffiSense` nor `ffiStop` modifies the state of the external world, so the external state `es` is unchanged. The external state is modified, e.g., when `ffiActuate` sends control values to actuators.

The full specification is ≈150 lines of HOL4 and formally captures the assumed behavior of each FFI from Table 3.1.

FFI Stub Implementation. The end-user must provide external FFI implementations. Here, we implement `ffiSense` by filling a VeriPhy-generated C stub with calls to application-specific drivers. Per the specification, `ffiSense` populates `sensor_vals` with the actual sensor values:

```
void ffiSense(int32_t *sensor_vals, long nSensors) {
  sensor_vals[0] = distanceDriver();  // return d
  sensor_vals[1] = currentTime();     // return t
}
```

We discuss the assumption that `ffiSense` complies with its HOL specification. The specification says that the length of `sensor_vals` in machine words should be the number `NSENSORS` of sensed variables in the system, e.g., 2 in our example. The length requirement is a basic and unsurprising well-formedness requirement without which `ffiSense` could not return all the sensor values without violating memory safety. The second requirement of the specification is that the sensor values returned by `ffiSense` must agree with the ground truth of the world. The second requirement is nontrivial because accurate sensing is a topic of active research in its own right. However, the requirement is a natural one at the present layer of abstraction: while proving sensor accuracy would be interesting future work, it is highly orthogonal to questions of correct FFI integration in CakeML.

CakeML FFI Wrapper. The CakeML sandbox program accesses the FFIs through CakeML wrapper functions. The wrapper functions use arrays to communicate the values of variables. A mapping between array indices and variable names is computed automatically by VeriPhy when it generates stub implementations of the wrapper functions in C. Specifically, VeriPhy's generated stub code contains comments that indicate which variable name is associated with each array index.

As an example of an FFI wrapper, the `sense` function wraps the `ffiSense` FFI:

```
fun sense () =
let val sensorArr = Word8Array.array (NSENSORS*WORD) 0
    val () = #(ffiSense) sensorArr
in arr_to_list sensorArr end
```

The `sense` function first allocates a byte array `sensorArr` with one word per sensor value. It then invokes `ffiSense` using CakeML's FFI call syntax `#(ffiSense)`. Once `sensorArr` contains actual sensor values, `sense` returns them reformatted as a list.

The specification of `ffiSense` is helpful for understanding the correctness of `sense`. Function `sense` allocates an array `sensorArr` of size `NSENSORS*WORD` to receive sensor values because that is the size expected by `ffiSense` according to its specification.

The specification of `ffiSense` also promises that the final values in `sensorArr` contain the ground truth, which is an essential assumption in order to prove that system safety is satisfied in the physical world (i.e., in the ground truth state) rather than only a

hypothetical world. An attentive reader will note that because real sensors are imperfect, it is overoptimistic to assume that a scalar value provided by a sensor exactly matches ground truth. In principle, the use of interval arithmetic can help relax this assumption to a more realistic one: we could allow sensors to return any proper interval containing the ground truth, so that a sensor can soundly and conservatively return a wider interval to communicate its uncertainty. While the version of VeriPhy in this chapter happens to use point intervals (equivalently, scalars) for simplicity, Chapter 8 will permit the use of proper intervals, which could be used by a clever Engineer to soundly capture sensor uncertainty.

The remaining FFIs are modeled and implemented similarly to the representative examples shown above, with `ffiCtrl` and `ffiActuate` also having their own oracles to model external control and actuation, respectively.

3.5.3 Verifying the CakeML Sandbox

Next, we verify the CakeML program `cmlSandbox`, assuming that the FFIs behave according to our FFI model. The main verification work is carried out with CakeML's Characteristic Formulae (CF) framework (Guéneau et al., 2017), which allows reasoning about the FFIs with assertions à la separation logic. As with interval arithmetic soundness, our results are generic across all sandbox instances.

We write $[\![\omega]\!]$ for a CakeML state containing an external state `es:ext` and a runtime store. We write $\big([\![\omega]\!], [\![\nu]\!]\big) \in [\![\texttt{cmlSandbox}]\!]$ to mean that executing the CakeML sandbox (`cmlSandbox`) from the initial CakeML state $[\![\omega]\!]$ terminates with the CakeML state $[\![\nu]\!]$, see (Guéneau et al., 2017) for formal details. The states implicitly agree with `cmlSandbox` as to which variables are sensors/actuators, etc. For any CakeML state $[\![\omega]\!]$, the underlying interval state $(\!|\omega|\!)$ represents each value $[\![\omega]\!](x) = w$ exactly by a point-interval $[w, w]$, which implies that sensing is exact. Sensor uncertainty could in principle be encoded with non-point intervals.

Theorem 3.9 (CakeML sandbox correctness)**.** *For any initial CakeML state $[\![\omega]\!]$, assuming that its `stop` oracle eventually stops the loop (by returning the bit `1` when queried), then we have a CakeML state $[\![\nu]\!]$ such that $\big([\![\omega]\!], [\![\nu]\!]\big) \in [\![\texttt{cmlSandbox}]\!]$. In addition,*

1. *If $[\![\omega]\!]$ violates initial condition P, then `cmlSandbox` leaves the initial CakeML state unchanged: $[\![\omega]\!] = [\![\nu]\!]$.*
2. *Else, either the `stop` oracle of $[\![\nu]\!]$ stopped the loop and $\big((\!|\omega|\!), (\!|\nu|\!)\big) \in (\!|\mathsf{sandbox}|\!)$ holds for the corresponding interval states, or*
3. *There exists $[\![\mu]\!]$ where $\big((\!|\omega|\!), (\!|\mu|\!)\big) \in (\!|\mathsf{sandbox}|\!)$ and $[\![\nu]\!]$ was obtained by actuating* (3.11) *in $[\![\mu]\!]$ where (after sensing) `intervalSem` raises a violation of the plant monitor `plantMon`* (3.13)*.*

We assume that the `stop` oracle eventually stops the loop because real systems do not run forever. Under this assumption, soundness is verified by induction on known-finite execution traces. The violation in case 3 of Theorem 3.9 is raised conservatively when `plantMon` has an unknown truth value (U), analogously to the control monitor `ctrlMon`. The violation is guaranteed to be raised when `plantMon` *first* fails, ensuring early detection of any model deviations.

Using CakeML's compiler correctness theorem (Tan et al., 2016), we extend Theorem 3.9 to the machine code, written `CML(cmlSandbox)`, which CakeML outputs from input `cmlSandbox`. We write $\{\!|\texttt{CML(cmlSandbox)}|\!\}$ for the output's (machine code) semantics, and accordingly $\{\!|\omega|\!\}$ for a machine-level program state.

Theorem 3.10 (Sandbox machine code correctness)**.** *Under the standard CakeML compiler correctness assumptions (Tan et al., 2016), let $\{\!|\omega|\!\}$ be an initial machine state whose `stop` oracle eventually stops the loop. Then we have a machine state $\{\!|\nu|\!\}$ such that $\big(\{\!|\omega|\!\}, \{\!|\nu|\!\}\big) \in \{\!|\texttt{CML(cmlSandbox)}|\!\}$, and $\{\!|\omega|\!\}, \{\!|\nu|\!\}$ satisfy one of the three cases listed in the conclusion of Theorem 3.9. The machine code may also exit with an out-of-memory error if the CakeML runtime exhausts its heap or stack.*

VeriPhy's end-to-end chain of correctness guarantees is a corollary.

Corollary 3.11 (End-to-end implementation guarantees)**.** *In addition to the assumptions of Theorem 3.10, assume further that Case 2 of the theorem occurs and the CakeML runtime does not run out of memory. Let $\{\mathsf{ctrl}; \mathsf{plant}\}^*$ be verified with postcondition Q, external interaction be sound, then there is a real state ν underlying state $\{\!|\nu|\!\}$ which is safe, i.e., some $\nu \in [\![Q]\!]$.*

Proof. Let P be the initial condition, Q the postcondition, and J the loop invariant.

By Case 2 of Theorem 3.10 we have some $\{\!|\omega|\!\}$ and $\{\!|\nu|\!\}$ s.t. $\omega\{\!|P|\!\}$ and $\big(\{\!|\omega|\!\}, \{\!|\nu|\!\}\big) \in \{\!|\texttt{CML(cmlSandbox)}|\!\}$. By soundness of CakeML compilation (Tan et al., 2016), the machine states $\{\!|\omega|\!\}$ and $\{\!|\nu|\!\}$ are the image under compilation of some $[\![\omega]\!]$ and $[\![\nu]\!]$ such that $\omega[\![P]\!]$ and $\big([\![\omega]\!], [\![\nu]\!]\big) \in [\![\texttt{cmlSandbox}]\!]$. By Theorem 3.9 we have $(\langle\!|\omega|\!\rangle, \langle\!|\nu|\!\rangle) \in \langle\!|\mathsf{sandbox}|\!\rangle$ where $\langle\!|\omega|\!\rangle$ and $\langle\!|\nu|\!\rangle$ are the interval states underlying $[\![\omega]\!]$ and $[\![\nu]\!]$. Because $\langle\!|\omega|\!\rangle$ is the underlying state of $[\![\omega]\!]$, we also have $\omega\langle\!|P|\!\rangle$. By Theorem 3.6 and Theorem 3.7 we have $\omega \in \langle\!|\omega|\!\rangle$ and $\nu \in \langle\!|\nu|\!\rangle$ such that $\omega \in [\![P]\!]$ and $(\omega, \nu) \in [\![\mathsf{sandbox}]\!]$. By composing Corollary 3.8 we have that sandbox is verified with precondition P, postcondition Q, and invariant J, yielding $\omega \in [\![J]\!]$ by the base case, then $\nu \in [\![J]\!]$ by the inductive step and $(\omega, \nu) \in [\![\mathsf{sandbox}]\!]$, then finally $\nu \in [\![Q]\!]$ by the postcondition step. To complete the proof, observe that $\nu \in \langle\!|\nu|\!\rangle$ and that $\langle\!|\nu|\!\rangle$ is the underlying state of $[\![\nu]\!]$ which is the preimage of $\{\!|\nu|\!\}$ under CakeML compilation. By transitivity, ν is an underlying real-valued state of machine code state $\{\!|\nu|\!\}$, which completes the proof.

When applying the CakeML correctness theorem, that theorem's assumptions include sensing soundness per Corollary 3.8, correctness of FFI with respect to the external state model, the availability of sufficient memory, and any other CakeML compiler correctness assumptions (Tan et al., 2016). □

3.6 Hardware Evaluation

The pipeline has successfully synthesized sandboxes for our velocity-controlled robot example, a train safety controller (Platzer & Quesel, 2009), and acceleration-controlled motion, both with (Section 3.7) and without (Quesel, Mitsch, Loos, Aréchiga, & Platzer, 2016) waypoint-following. This section evaluates the velocity-controlled robot on hardware and in simulation, then Section 3.7.3 evaluates a waypoint-following model in simulation. First,

the velocity-controlled robot is evaluated because it is a simpler, illustrative example that demonstrates core concepts. Then, the waypoint-following system (Section 3.7.3) demonstrates that the approach scales to non-trivial models.

The goal of the evaluations is to validate VeriPhy by observing its behavior when used with software and hardware implementations. A successful evaluation of VeriPhy should show that it detects control and plant monitor violations when they occur and that it then correctly applies the sandbox logic. We should certainly also show safe behavior when plant monitors hold, because we formally proved the safety of that case.

For VeriPhy to truly succeed, however, we should also explore non-safety properties such as *operational suitability*. For VeriPhy to be useful in practice, we wish to minimize any unnecessary violations of the monitors and minimize any unnecessary use of the fallback. For example, if the fallback were used so heavily that the vehicle stopped in place rather than eventually driving to its goal, that would constitute failure of the operational suitability objective. Thankfully, the vehicle did drive to its goal. One contributing factor to operational suitability was the fact that our control code and machine arithmetic were sufficiently fast. While no special effort was required to ensure the speed of control and arithmetic code, it was important to check their performance because slow code would have caused plant monitor violations by exceeding the time budget allowed by the model. Those plant monitor violations would have interfered with operational suitability by increasing the use of the fallback and would have potentially even compromised safety; it is well-known that slow code could lead to unsafety in a CPS if slow performance forces the controller to run so rarely or so late that it cannot enforce safety. The operational suitability aspect of the evaluation is crucial because the Engineer would not be willing to use programs which cannot reach their operational goals or whose code is so slow that they become unsafe.

Our evaluations also contribute to a subjective assessment of VeriPhy by serving as a proof-of-concept that the sandbox controllers generated by VeriPhy can be integrated with real code and executed. For the velocity-controlled car, it may seem self-evident that the generated sandbox code could be integrated with an implementation, but issues could have hypothetically arisen if our model and robot were mismatched. For example, the motors on our robot expect velocities as inputs, so it was important that we used a velocity-controller model rather than an acceleration-controlled model, which would have been more difficult to integrate with a velocity-based motor interface. The importance of integration increases with system complexity, e.g., in Section 3.7.3. In short, integration should not pose a major programming hurdle if models are designed with implementation in mind, but one contribution of Section 3.7 will be that it presents a model which is well-suited for integration with the software implementation of the 2D vehicle. By demonstrating well-suited models, we show that it is possible in practice for the Logic-User to build and prove models that can serve as inputs to VeriPhy and result in useful code for the Engineer.

Before presenting the evaluation, we summarize the goals of the evaluation with respect to each character. Since the Logician is concerned with formal proofs rather than experiments, the experimental evaluation is orthogonal to his own priorities. The experiments do however show that the Logician has fulfilled his safety promise to the Engineer. The Engineer also needed operational suitability and needed code which she could integrate with

her hardware and software implementations of CPSs, which are respectively demonstrated by our plant monitor compliance results and by the fact that it was possible to implement the experiments. In this evaluation, the Logic-User shows she was able to build and prove models that are suitable for use with VeriPhy. In Section 3.8.3, we will reflect on challenges that the Engineer and especially the Logic-User encounter while respectively developing the experiments and verified models, which will motivate the following parts of the thesis.

This section presents a robot evaluation and a simulated evaluation; the robot evaluation is the primary evaluation of the two. We perform the robot evaluation on commodity robot hardware with several controllers. This common platform minimizes hardware cost while our multiple controls allow testing the approach's generality. The controllers are tested in several scenarios, of which some comply with the model and others do not. Thus, we can both assess that the expected safe behavior occurred for compliant scenarios and assess that our monitors detect non-compliant environments. We augment the robot evaluation with simulations showing how the sandbox controller responds to various environments with various untrusted controllers. Simulations are a useful supplement because they offer a higher degree of reproducibility and fine-grained experimental control than hardware-based experiments.

Hardware Platform and Calibration. We give details of our robot platform that are relevant to the experiments. Our experiments use a GoPiGo3 Raspberry Pi-based robot. It is equipped with two separately controlled motors and a laser distance sensor with 25° field-of-view and typical indoor measurement range on white background of 200 cm with ≈94% obstacle detection rate. Depending on operating temperature and voltage, the distance measurements are off by at most ±3 cm. The motors take speed commands in the range $-25\,\frac{\text{cm}}{\text{s}}$ to $25\,\frac{\text{cm}}{\text{s}}$, controlled internally with a proportional-integral-derivative (PID) controller. It has a stopping margin of about 2.5 cm from engaging "brakes" by setting $v = 0$ until full stop from maximum speed. The sensed distance incorporates this margin, closely mimicking instantaneous stopping per our model.

Drivers. We give performance details for the drivers for sensing and actuation on the GoPiGo3, whose execution dominated the running time of the experiments. GoPiGo3 provides C drivers for the motors, but only Python drivers for the distance sensor. The vast majority of execution time is spent calling the Python driver, though the reported running time of the driver has varied between works. In the experiments performed during the preparation of this thesis, the running of time of the sensor driver was ≈370–400 ms, while running times of ≈180–220 ms were reported in the conference version of the work (Bohrer et al., 2018). Thus, the system completes ≈2.5–5 control cycles per second (Hz).

The high running time for the sensor can be attributed to the fact that our experiment code creates a new Python process every time it calls the Python sensor driver. We expect that it is possible to greatly reduce the sensing time and thus greatly increase control frequency, for example by creating a single long-running Python program and calling it repeatedly from C using inter-process communication or using the standard Python API for C integration. Actuation and control each took less than 1 ms, so reducing Python

integration overhead would suffice to significantly increase control frequency. The default[4] sensing time of the laser sensor is 30 ms, so we predict a control frequency of ≈33 Hz could be obtained if driver overhead were eliminated.

Experiment Setup. The robot is initially stationary, 75 cm from an obstacle, then drives straight toward the obstacle with user-defined constant speeds $10, 15, 20, 25\,\frac{\text{cm}}{\text{s}}$ (maximum speed of the robot). Since the robot measures time in ms, we measure speed in $\frac{\text{mm}}{\text{s}}$ and distance in μm. Thus, the greatest distance in the system, 75 cm, can be represented in 20 bits, well within our 32-bit limit. We performed the experiment with both stationary and moving obstacles. The robot stops close to a stationary obstacle, with ≈3 cm safety margin to account for sensor and actuator uncertainty. If the obstacle moves away, the robot follows, stopping close to the obstacle's final position. If the obstacle moves closer, the robot stops before reaching the obstacle.

We tested two implementations of the untrusted external controller. Controller A follows a user-defined speed when safe to do so and otherwise stops. This is safe and thus does not violate the monitor. Controller B first sets a user-defined speed then spikes to maximum speed near the obstacle. This is unsafe and violates the monitor, invoking fallback control. Our experiments on both the real robot and a simulated plant record distance, speed, and monitor violations vs. time.

Experimental Results. Fig. 3.10 plots distance over time in simulation for maximum speed $V = 25\frac{\text{cm}}{\text{s}}$, system delay $\mathsf{T} = 220$ ms, and initial distance 75 cm, with varying sensor disturbance and both static and malicious obstacles. Solid lines indicate safe sandbox controller executions, including those where fallback is engaged when a control violation occurs. Dashed lines indicate plant violations where the environment caused a collision. Symbols **C†**,**C↯**,**P↯**, and **C✓** indicate speed spike, control/plant violations, and restoration of normal control, respectively.

Fig. 3.11 plots results on the real robot. We plot a correct controller approaching a stationary obstacle (blue line with circle markers), faulty controller approaching a stationary obstacle (orange-red line with × markers), obstacle approaching the robot then receding (yellow line with + markers), and robot following an obstacle which is stationary at first before receding (black line with no markers). Symbols **Ob+**, **Ob0**, and **Ob-** indicate proceeding, stopped, and receding obstacles, respectively. The figures for both the simulation and robot show that monitors correctly detect plant violations. Fallback is then engaged as a safety best-effort, which ensured safety in simulation.

The robot engaged the fallback at ≈4.6 cm from the obstacle, stopping at ≈2.6 cm (due to the safety margin). Under small disturbances, the plant monitor holds and safety is assured. Malicious obstacles or dangerously high disturbances are detected as plant violations, triggering fallback. Plant violations represent scenarios where physical behavior deviated from the model, thus where the system's continued safety throughout the future cannot be formally proved, thus fallback control is merely a best-effort for safety in this scenario. When one encounters plant violations in practice, the appropriate approaches to

[4]The sensor is equipped with several presets ranging from 20–200 ms which vary in accuracy.

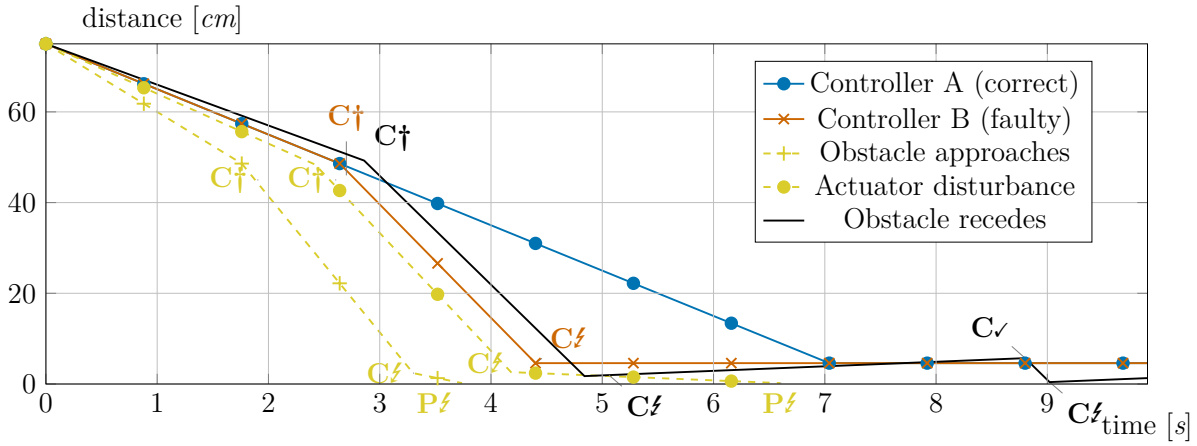


Figure 3.10: Controller sandbox, simulated plant.

reduce or even eliminate them are to change the model (e.g., to account for malicious obstacles) or change the implementation (e.g., to keep disturbances within a safe range). We intentionally simulated controller faults at $d \approx 50\,\text{cm}$ by issuing $v = V = 25\frac{\text{cm}}{\text{s}}$ continually. The fallback engages right before the faulty controller would become unsafe ($d < \mathsf{T}V$). The fallback engaging is the desired behavior. In contrast to plant violations, provable safety can be maintained when control violations occur because the violating control decision is automatically replaced with the proven-safe fallback action.

Slightly different scenarios are tested in the simulated experiments (Fig. 3.10) vs. the robot experiments (Fig. 3.11). The simulated results contain two similar-looking yellow plots, but the plot arise from tests of different phenomena. In the first case (the plot with + markers), forward motion of the obstacle is simulated explicitly; in the latter (circle markers), the simulation models actuation error wherein the robot moves at a velocity different from the one which is commanded. In contrast, the robot experiments only contain a single yellow plot because there is no separate test for actuator disturbance; any disturbance present in the physical system would be present in all experiments. The yellow plot of the robot experiments also differs incidentally from the simulated forward obstacle motion because it tests forward and backward obstacle motion in the same run. Likewise, in the receding case of the robot experiment, the obstacle waits before receding, whereas it recedes from the beginning of the simulated case. These movements differ because the simulations were implemented with simplicity in mind, whereas the robot experiments could easily incorporate more complex motion on the part of the experimenter. In any case, it was not our goal to make the simulated and robotic motion agree. Rather, the robot experiments serve as the primary evaluation of the real-world robotic behavior. The simulated evaluation supplements the robot experiments by providing greater reproducibility and fine-grained experimental control.

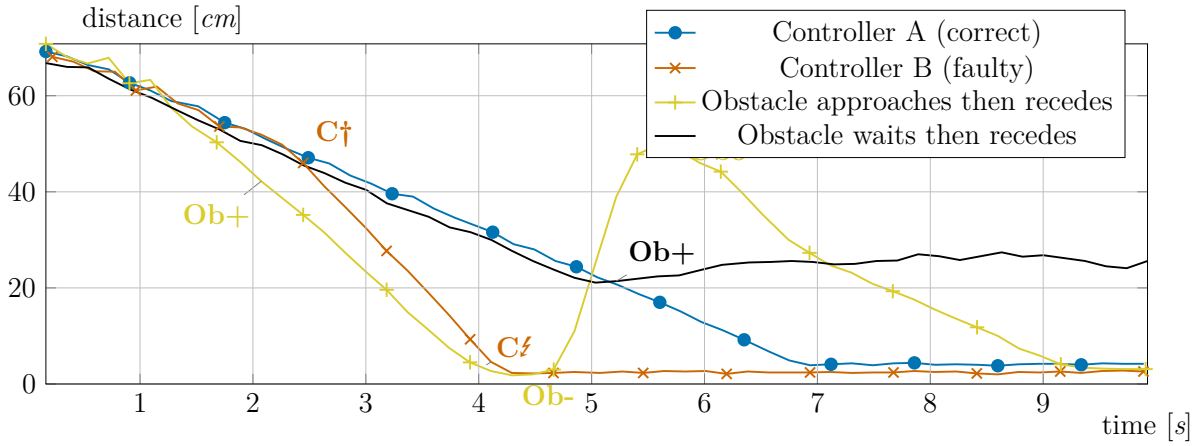


Figure 3.11: Controller sandbox, real robot.

3.7 2D Robot Case Study

The 1D robot model of Section 3.2 allowed us to validate the basic principles of VeriPhy, but that simple model did not confront certain modeling and verification challenges that are common to real robotic systems. This section studies a 2D wheeled robot model in order to expose and resolve common 2D modeling, verification, and implementation challenges, thus exposing the strengths and limitations of VeriPhy when applied to practical models. The vehicle model developed in this section follows piecewise curved (Dubins, 1957) paths with speed limits and acceleration control. This is significantly more representative of realistic driving scenarios than the 1D model, because speed limits, acceleration, and steering are fundamental to driving. The resulting VeriPhy sandbox is tested in integration with controllers and test environments that we developed for AirSim (Shah et al., 2018), a simulator which is widely used for both ground and air robotics. In Section 3.8, we discuss the challenges faced in realistic scenarios like 2D driving and how we will overcome them.

3.7.1 2D Robot Model

This section introduces our 2D robot model in dL. The circular Dubins dynamics are bloated by a tolerance ε, which accounts for imperfect actuation and for discrepancies between the Dubins dynamics and real dynamics of the implementation. That is, because realistic robots never follow a path perfectly, we will bloat each arc to an annulus section which is more easily followed, shown in Fig. 3.13. In our *relative* coordinate system, the robot's position is always the origin, from which perspective the waypoint "moves toward" the vehicle (Fig. 3.13). The current position of the current waypoint is specified by coordinates (x, y). The curvature of the path to the waypoint is k and the speed limits form an interval $[\mathsf{vl}, \mathsf{vh}]$ which the robot's velocity v must satisfy by the time the waypoint is approximately reached, i.e., when distance to the waypoint is at most ε. Curvature parameter $\mathsf{k} \neq 0$ yields curved arc sections while $\mathsf{k} = 0$ yields line segments. Together, these are the primitives of Dubins paths. The model is presented here without units for the sake

of readability. The version used in simulation (Section 3.7.3) includes unit conversions to account for the units of the simulator, and has been verified as well. While our relative coordinate system is a natural fit for vehicle-centric sensing and eliminates the need to model a non-zero vehicle position, it does make enforcement of absolute paths more difficult. Because the comparison of relative and absolute coordinates requires familiarity with the model, the comparison is deferred (Fig. 3.17) until after the model has been introduced.

Our hybrid program α is a time-triggered *control-plant* loop: $\alpha \equiv \{\mathsf{ctrl}; \mathsf{plant}\}^*$. Relative coordinates simplify proofs (fewer variables) and implementation (real sensors and actuators are vehicle-centric). The main variables of our model are depicted in Fig. 3.12.

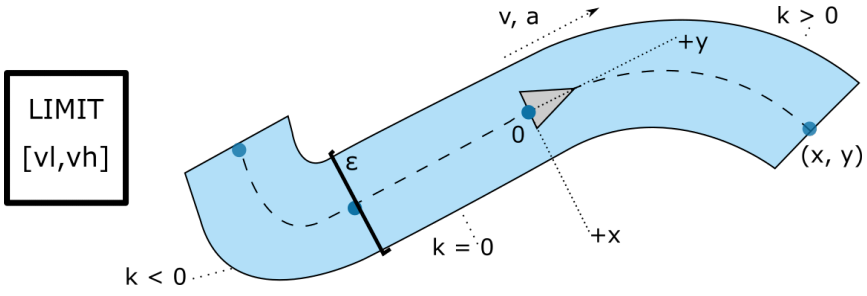


Figure 3.12: 2D driving model: system variables.

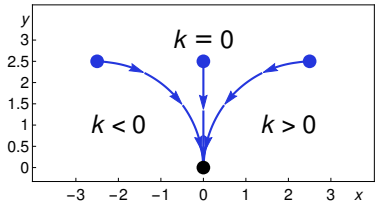


Figure 3.13: Trajectories of dynamics for different choices of k.

The plant is an ODE describing the robot kinematics:

$$\mathsf{plant} \equiv \{\mathsf{x}' = -\mathsf{v}\,\mathsf{k}\,\mathsf{y},\ \mathsf{y}' = \mathsf{v}\,(\mathsf{k}\,\mathsf{x} - 1),\ \mathsf{v}' = \mathsf{acc},\ \mathsf{t}' = 1 \tag{3.16}$$

$$\&\ \mathsf{t} \leq \mathsf{T}\ \wedge\ \mathsf{v} \geq 0\} \tag{3.17}$$

The positive x axis points to the right of the vehicle and the positive y axis points forward[5]. Here, acc is an input from the controller describing the acceleration with which the robot is to follow the arc of curvature k to waypoint (x, y). In the equations for x', y': *i)* The v factor models (x, y) moving at linear velocity v, *ii)* The $\mathsf{k}, \mathsf{x}, \mathsf{y}$ factors model circular motion with curvature k, with $\mathsf{k} > 0$ corresponding to counter-clockwise rotation of the waypoint, $\mathsf{k} < 0$ to clockwise rotation, and $\mathsf{k} = 0$ to straight-line motion, *iii)* The additional -1 term in the y' equation indicates that the center of rotation is $\left(\frac{1}{\mathsf{k}}, 0\right)$. To explain why, we

[5]This is a different coordinate system from the journal version (Bohrer, Tan, et al., 2019). Our coordinate system was chosen to be accessible to readers unfamiliar with the driving literature, while the one in the journal version was chosen to be accessible to readers familiar with the driving literature.

first introduce a standard idiom for representation of rotational motion in ODEs, an idiom which is clearest in the simpler ODE $\{\mathsf{x}' = -\mathsf{y}, \mathsf{y}' = \mathsf{x}\}$, whose solutions (call them $x(t)$ and $y(t)$) will by definition satisfy $x'(t) = -y(t)$ and $y'(t) = x(t)$ which are the derivative rules for cosine and sine, respectively. From initial conditions where $x^2 + y^2 = 1$, the functions $x(t) = \cos(t), y(t) = \sin(t)$ satisfy the equations $x'(t) = -y(t)$ and $y'(t) = x(t)$ and are thus *the* solution, by uniqueness of solutions, meaning that the behavior of the ODE is rotation along the unit circle centered at the origin.

In the more general setting of motion at speed v with the circle centered at $(\mathsf{off}, 0)$, we define $\mathsf{x}' = -\mathsf{v}\ \mathsf{k}\ \mathsf{y}$ and $\mathsf{y}' = \mathsf{vk}(\mathsf{x} - \mathsf{off})$. Factor v describes the linear speed of the waypoint (or vehicle). To ensure proper speed, multiplication by normalization factor k effectively divides out the magnitude of position vector x, y. Factor $\mathsf{x} - \mathsf{off}$ captures the fact that y' should be proportional to the current *relative* x-coordinate, which is $\mathsf{x} - \mathsf{off}$. We confirm our intuition by considering the behavior of $\mathsf{x} - \mathsf{off}$ at several extremal points. For example, $\mathsf{x} - \mathsf{off}$ and thus y' go to zero at the point where (x, y) is $(\mathsf{off}, \frac{1}{|\mathsf{k}|})$, the topmost point of the circle and thus a point where the motion of the waypoint is tangent to the x axis. Likewise, the circle intersects the x-axis when (x, y) is point $(\frac{1}{k} - \mathsf{off}, 0)$, which maximizes $|\mathsf{x} - \mathsf{off}|$ and yields $|\mathsf{y}'| = \mathsf{v}$, which is desired because motion is perpendicular to the y-axis.

The equations $\mathsf{v}' = \mathsf{acc}$ and $\mathsf{t}' = 1$ respectively say that acceleration is the derivative of velocity and t is the current time. The domain constraint $\mathsf{t} \leq \mathsf{T} \wedge \mathsf{v} \geq 0$ says that the duration of one control cycle shall never exceed a timestep parameter $\mathsf{T} > 0$ representing the maximum delay between control cycles and that the robot never drives in reverse. Fig. 3.14 illustrates a goal of size $\varepsilon = 1$ around the origin (equivalent to a goal around the waypoint) and several trajectories which pass through the goal. Note that our assumption on T is $\mathsf{T} > 0$ as opposed to $\mathsf{T} \geq 0$ in Section 3.2 because we are interested liveness, which requires $\mathsf{T} > 0$ whereas safety does not.

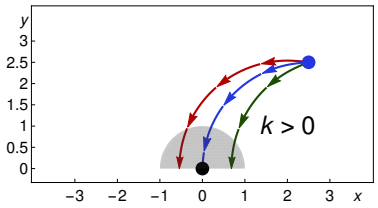


Figure 3.14: Trajectories of plant for choices of $\mathsf{k} > 0$ when $\varepsilon = 1$.

The controller's task is to compute an acceleration acc which slows down (or speeds up) soon enough that the speed limit $\mathsf{v} \in [\mathsf{vl}, \mathsf{vh}]$ is ensured *by the time* the robot reaches the goal. The controller model is given in Fig. 3.15.

Plan assignment $(\mathsf{x}, \mathsf{y}) := *$ chooses the next 2D waypoint, assignment $[\mathsf{vl}, \mathsf{vh}] := *$ chooses the speed limit interval, and $\mathsf{k} := *$ chooses any curvature. The notations $(\mathsf{x}, \mathsf{y}) := *$ and $[\mathsf{vl}, \mathsf{vh}] := *$ are suggestive notations for assignments which define 2D points and intervals, but each is implemented as the sequential composition of two scalar nondeterministic assignments. The *feasibility* test $?\mathsf{Feas}$ determines whether or not the chosen waypoint, speed limit, and curvature are *physically* attainable in the current state under the plant dynamics (e.g., it checks that there is enough remaining distance to get within the speed

$$
\begin{aligned}
\mathsf{Ann} &\equiv\ |\mathsf{k}|\varepsilon \le 1 \wedge \left|\frac{\mathsf{k}\ (\mathsf{x}^2+\mathsf{y}^2-\varepsilon^2)}{2} - \mathsf{x}\right| < \varepsilon \\
\mathsf{Feas} &\equiv \mathsf{Ann} \wedge \mathsf{y}{>}0 \wedge 0{\le}\mathsf{vl}{<}\mathsf{vh} \wedge \mathsf{AT}{\le}\mathsf{vh}-\mathsf{vl} \wedge \mathsf{BT}{\le}\mathsf{vh}-\mathsf{vl} \\
\mathsf{ctrl} &\equiv\ (\mathsf{x},\mathsf{y}) := *;\ [\mathsf{vl},\mathsf{vh}] := *;\ \mathsf{k} := *;\ ?\mathsf{Feas};\ \underbrace{\mathsf{acc} := *;\ ?\mathsf{Go}}_{\mathsf{ctrl}_\mathrm{a}} \\
\mathsf{Go} &\equiv -\mathsf{B} \le \mathsf{acc} \le \mathsf{A} \wedge \mathsf{v}+\mathsf{accT} \ge 0 \\
&\quad \wedge \Big(\mathsf{v} \le \mathsf{vh} \wedge \mathsf{v}+\mathsf{accT} \le \mathsf{vh} \vee \\
&\qquad (1+|\mathsf{k}|\varepsilon)^2\Big(\mathsf{vT}+\frac{\mathsf{acc}}{2}\mathsf{T}^2+\frac{(\mathsf{v}+\mathsf{accT})^2-\mathsf{vh}^2}{2\mathsf{B}}\Big)+\varepsilon \le \|(\mathsf{x},\mathsf{y})\|_\infty\Big) \\
&\quad \wedge \Big(\mathsf{vl} \le \mathsf{v} \wedge \mathsf{vl} \le \mathsf{v}+\mathsf{accT} \vee \\
&\qquad (1+|\mathsf{k}|\varepsilon)^2\Big(\mathsf{vT}+\frac{\mathsf{acc}}{2}\mathsf{T}^2+\frac{\mathsf{vl}^2-(\mathsf{v}+\mathsf{accT})^2}{2\mathsf{A}}\Big)+\varepsilon \le \|(\mathsf{x},\mathsf{y})\|_\infty\Big)
\end{aligned}
$$

Figure 3.15: Controller model for 2D circular driving.

limit interval). We also simplify plans so all waypoints satisfy $\mathsf{y} > 0$, by subdividing any violating path segments automatically. Assuming the arc is not a full circle, bisecting the arc once always suffices the initial arc has angle strictly less than 2π, so the bisected arc's angle is less than magnitude π and thus satisfies $\mathsf{y} > 0$. It is worth this effort to gain the assumption $\mathsf{y} > 0$ because it simplifies the external controller and proofs. The abbreviation ctrl_a names just the control code responsible for deciding *how* the waypoint is followed rather than *which* waypoint is followed.

In Feas, formula Ann says we are within the *annulus section* (Fig. 3.16) ending at the waypoint (x,y) with the specified curvature k and width ε. In Fig. 3.16, trajectories from the displaced green and red waypoints with slightly different curvatures remain within the annulus. A larger choice of ε yields more error tolerance in the sensed position and followed curvature at the cost of an enlarged goal region. Formula Ann also makes a simplifying assumption that the radius of the annulus is at least ε, which is a modest assumption: it is like assuming the radius of a turn is at least the width of a road. Feas also says the speed limits are assumed distinct and large enough to not be crossed in one control cycle.

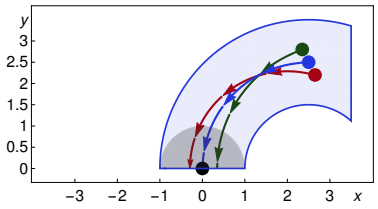


Figure 3.16: Annular section through the (blue) waypoint $(2.5, 2.5)$.

The *admissibility* test ?Go checks that the chosen acc will take the robot to its goal with a safe speed limit, by *predicting future motion* of the robot. We illustrate this with the

upper bound conditions. The bound will be satisfiable after one cycle if either the chosen acceleration acc already maintains speed limit bounds ($\mathsf{v} \leq \mathsf{vh} \land \mathsf{v}+\mathsf{acc}\mathsf{T} \leq \mathsf{vh}$) or when there is enough distance left to restore the limits before reaching the goal. For straight line motion ($\mathsf{k} = 0$), the required distance is found by integrating acceleration and speed:

$$\underbrace{\mathsf{v}\mathsf{T} + \frac{\mathsf{acc}}{2}\mathsf{T}^2}_{\text{distance in time } \mathsf{T}} + \frac{\overbrace{(\mathsf{v}+\mathsf{acc}\mathsf{T})}^{\text{speed in time } \mathsf{T}})^2 - \mathsf{vh}^2}{2\mathsf{B}} + \varepsilon \leq \|(\mathsf{x},\mathsf{y})\|_\infty$$

where $\mathsf{acc} \in [-\mathsf{B}, \mathsf{A}]$. The extra factor of $(1+|k|\varepsilon)^2$ in formula Go accounts for the fact that an arc along the inner side of the annulus is shorter than one along the median (Fig. 3.16).

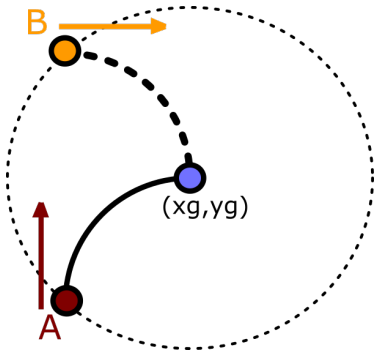


Figure 3.17: Relative model specifies non-unique motion.

Having discussed each section of the model, we now discuss the subtleties and limitations that come with our relative coordinate system approach. Our relative coordinate system is a natural choice from the perspective of implementation platforms where sensors typically use relative coordinates and also reduces the number of variables compared to an absolute coordinate system, which is associated with easier automated verification of arithmetic proof goals. However, it is important to recognize that the use of relative coordinates introduces a conceptual gap compared to the intuitive notion of an absolute path as depicted in Fig. 3.12 and that the conceptual gap has implications for the strength of the results in our experimental evaluation (Section 3.7.3). We represent the path from the vehicle to the waypoint using a curvature and endpoint, which notably *do not* uniquely represent the current position and orientation of the robot, but rather represent a circle centered on the waypoint. This phenomenon is depicted in Fig. 3.17, where the waypoint is indicated (x,y). Consider the two states A (red) and B (orange) depicted in Fig. 3.17, where the origin of the vehicle in each state is the labeled point and its orientation is the same-colored vector drawn next to the labeled point. The points A and B are starkly different in an absolute coordinate system but indistinguishable in our model because the relative position of the waypoint is the same from each point: the path from A to the waypoint is the solid arc segment while the path from B to the waypoint of the same curvature is the thick dashed arc segment. By rotating the solid arc segment and orientation vector around the waypoint, we observe that *every* point on the thin dashed circle appears

identical to points A and B in relative coordinates. We refer to the points on the thin dotted circle as ambiguous points and motion along the circle as ambiguous motion.

It is unclear whether ambiguous motion occurs much in practice. The direction of motion in practice is typically similar to orientation, but ambiguous motion occurs in the direction of the dashed circle, which is not tangent to the orientation. Ambiguous motion would require not only an extreme difference between orientation and direction of movement, but a matching change in orientation as well. In future work, one might try to detect ambiguous motion by monitoring *lower bounds* on elapsed distance toward the waypoint. A subtle feature of ambiguous motion is that it preserves distance to the waypoint, while motion along the arc reduces remaining distance, thus a mismatch between the vehicle's sensed velocity and waypoint distance could indicate ambiguous motion.

Nonetheless, it is important to recognize that the possibility of ambiguous motion is a limitation of our model. The plant monitor is responsible for assessing model compliance as a means of assessing safety. The limitation of our model is that because the model's definition of safety allows ambiguous motion, the plant monitor will allow it too, regardless of whether that motion is safe in an informal sense.

While it is important to categorize limitations of our model, the experimental evaluation (Section 3.7.3) will ultimately embrace the idea of monitoring whether there exist feasible paths rather than checking whether a fixed path is satisfied. After all, the VeriPhy controller will apply emergency braking whenever monitors fail, so it is important for liveness that monitors eventually pass. Monitoring compliance to a fixed path would allow the system to get stuck if a path is ever violated, while monitoring existence of a path provides an opportunity for the system to recover by identifying a new path.

3.7.2 Proofs

Our proofs for the 2D model include both safety and liveness properties. Safety says the robot stays within its annulus section and maintains its speed limits, while liveness says the robot eventually reaches the end of the section assuming it runs a particular control algorithm and has perfect actuation. While classical VeriPhy only uses a safety proof to synthesize its sandbox, the liveness proof serves several useful purposes. The liveness theorem is invaluable because it validates the dL model: if it were not even possible at the level of the dL model to achieve liveness, then it would be absolutely impossible at the level of implementation to achieve liveness. Because it is undesirable to change the model drastically once implementation has started (i.e., delays in modeling and verification will trickle down to greater delays in implementation and testing), it is valuable to have this confidence in the model before implementation even begins. Additionally, the safety and liveness proofs for our 2D model serve as an important point of comparison for the proofs of Part II, where game proofs combine safety and liveness reasoning.

Safety means speed limits are obeyed whenever the robot reaches a waypoint:

Theorem 3.12 (Safety)**.** *The following* dL *formula is valid when* ctrl *and* plant *are defined according to Fig. 3.15 and* (3.17) *and when* J *is defined as in Section 3.7.3:*

$$\mathsf{A}{>}0 \land \mathsf{B}{>}0 \land T{>}0 \land \varepsilon{>}0 \land J \to [\{\mathit{ctrl}; \mathit{plant}\}^*]\big(\|(x,y)\| \leq \varepsilon \to \mathsf{v} \in [\mathit{vl}, \mathit{vh}]\big)$$

Because the safety *proof* will show that some loop invariant J is preserved, we could trivially strengthen the postcondition to J if we wish. The invariant will include facts such as the robot always remaining within some annulus of width ε around an arc leading to the waypoint, which is useful for safety, the limitations described in Fig. 3.17 notwithstanding. The first four assumptions ($\mathsf{A}{>}0 \wedge \cdots \wedge \varepsilon{>}0$) are basic sign conditions on the symbolic constants used in the model. The final assumption, J, is the loop invariant. The full definition of J is in Section 3.7.3. We write $\|(\mathsf{x},\mathsf{y})\|$ for the Euclidean norm $\sqrt{\mathsf{x}^2+\mathsf{y}^2}$ and consider the robot "close enough" to the waypoint when $\|(\mathsf{x},\mathsf{y})\| \leq \varepsilon$ for our chosen goal size ε. While speed limits and loop invariants capture the desired notion of safety, they do not prove that the robot ever reaches the goal, which is a *liveness* property:

Theorem 3.13 (Liveness)**.** *The following* dL *formula is valid when* ***ctrl*** *and* ***plant*** *are defined according to Fig. 3.15 and* (3.17) *and when J is defined as in Section 3.7.3:*

$$\begin{aligned}&\mathsf{A}{>}0 \wedge \mathsf{B}{>}0 \wedge \mathsf{T}{>}0 \wedge \varepsilon{>}0 \wedge J \rightarrow\\ &[\{\mathsf{ctrl};\mathsf{plant}\}^*]\Big(\mathsf{v}{>}0 \wedge \mathsf{y}{>}0 \rightarrow\\ &\quad\langle\{\mathsf{ctrl}_a;\mathsf{plant}\}^*\rangle\big(\|(\mathsf{x},\mathsf{y})\| \leq \varepsilon \ \wedge\ \mathsf{v} \in [\mathsf{vl},\mathsf{vh}]\big)\Big)\end{aligned}$$

This theorem has the same assumptions as Theorem 3.12. It says that no matter how long the robot has been running ($[\{\mathsf{ctrl};\mathsf{plant}\}^*]$), then if some simplifying assumptions[6] still hold ($\mathsf{v} > 0 \wedge \mathsf{y} > 0$) the controller can be continually run ($\langle\{\mathsf{ctrl}_\mathrm{a};\mathsf{plant}\}^*\rangle$) with admissible acceleration choices (ctrl_a) to reach the present goal ($\|(\mathsf{x},\mathsf{y})\| \leq \varepsilon$) within the desired speed limits ($\mathsf{v} \in [\mathsf{vl},\mathsf{vh}]$).

3.7.3 Simulations

One crucial aspect of the notion of end-to-end verification pursued in this thesis is that correctness guarantees must hold at implementation-level. In order to validate our claim of implementation-level correctness, it is crucial to evaluate VeriPhy by implementing systems with it, as we have done in Section 3.6. That evaluation does not tell the entire story, however, because a crucial aspect of our practicality goal is ensuring that VeriPhy can scale to models and proofs of non-trivial complexity.

In the evaluation, we simulate a car driving in several different environments under several different controllers, during which we record the time taken to complete the course and the percentage of time during which control and plant monitors respectively report non-compliance. The details of the environments and controllers are incidental. Rather, we seek

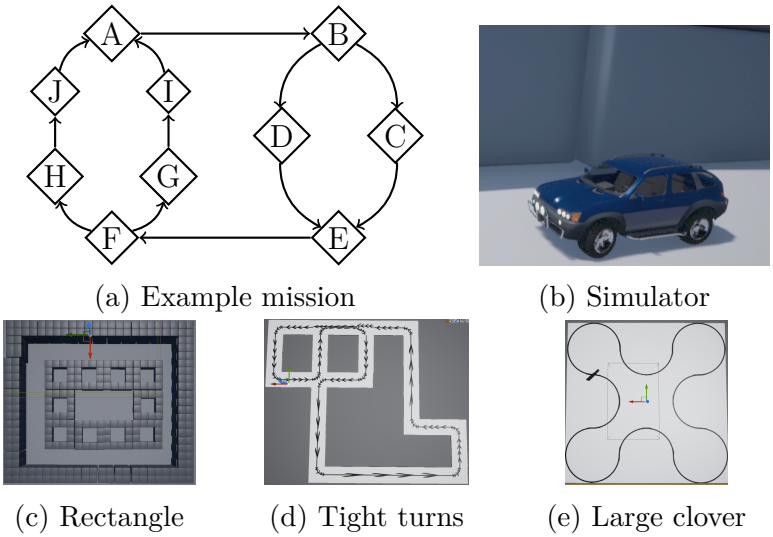
[6]The simplifying assumptions $\mathsf{v} > 0 \wedge \mathsf{y} > 0$ say the robot is still moving forward and the waypoint is still in the upper half-plane, i.e., we have not run *past* the waypoint. In the case that the waypoint has not been reached, the assumption $\mathsf{v} > 0$ can be proved by unrolling the loop once and accelerating. We assume $\mathsf{v} > 0$ for simplicity because it is intuitively obvious that acceleration is possible when the waypoint has not been reached and liveness properties hold trivially if the goal has already been reached. The assumption $\mathsf{y} > 0$ is easily met in practice, which is best seen by observing $\mathsf{y} > 0$ holds whenever the angular length of the arc to the waypoint is strictly between 0 and π radians. Any curve of length at least π radians is equivalent to the concatenation of shorter arcs, so no generality is lost. In the experiments, we satisfy the assumption by splitting long arc sections before passing them to the VeriPhy controller.

to minimize monitor noncompliance, because a high degree of compliance indicates that our verified model was accurate with respect to the simulated implementation and because plant monitor compliance, in particular, is required in order for formal implementation-level safety guarantees to be applicable. The Engineer would be unlikely to find VeriPhy useful if the Logician can provide guaranteed safety to her for nontrivial systems. Moreover, the evaluation's implementation serves as a proof-of-concept for the integration of VeriPhy with existing codebases in settings where the physics are nontrivial. Such a proof-of-concept is important to both the Engineer and the Logic-User because it both shows that the Engineer can successfully integrate VeriPhy-generated code with her implementation and shows, more subtly, that the Logic-User can write a realistic-enough model to generate that useful code when fed as the input to VeriPhy.

Given that this section's emphasis is on model and proof complexity, it is appropriate to use simulations rather than hardware for the present evaluation. Nonetheless, the simulation platform we chose, the autonomous driving mode of AirSim (Shah et al., 2018), is known to have reasonable physical fidelity in addition to its high-quality visuals and an open-source code base that is particularly friendly to modification. While absolute fidelity of the physical simulation is not the primary goal of AirSim's underlying Unreal Engine, it is possible to model nontrivial mechanical systems in Unreal. For example, the AirSim car independently models the physics of every tire and suspension, allowing customization of details such as tire friction. Because AirSim models its car in significant detail and because AirSim has been successfully applied in dozens of projects, we have good reason to believe its physics model is more faithful than any purpose-built one-off simulation would be.

In AirSim, the author implemented several basic control algorithms known as *bang-bang* and *proportional-derivative* control. The author developed several driving environments for testing, shown in Fig. 3.18. Fig. 3.18a illustrates the plan data structure used to drive the controller: a plan is represented as a graph. Our examples are cycle graphs because the car drives in a repeatable loop, but the data structure allows higher degrees in order to represent non-deterministic plans where the controller is allowed to choose their path. The simulations showed that while it is rare to achieve zero monitor failures, failure rates were kept low enough to complete all environments. The failure rate for the control monitor captures the percentage of control cycles in which the fallback action was applied to ensure safety. The failure rate for the plant monitor captures the percentage of control cycles in which we cannot guarantee that the safety theorem applies indefinitely in the future, i.e., we cannot guarantee that the vehicle will remain on its desired trajectory or maintain the required speed limits in the future. The fallback was not applied when only the plant monitor failed. Non-completion typically indicates that the fallback controller braked to a stop and the monitor still failed in that state, leaving the system stationary forever.

When interpreting completion results and safety guarantees, it is important to acknowledge how the planning code from our experiments computes the waypoint coordinates and curvatures passed to the monitor. We do *not* strictly enforce that the vehicle is always on the fixed, hard-coded path from each environment, rather when the vehicle begins to deviate, the planner chooses a new arc which leads to the same hard-coded waypoint, i.e., it replans the path dynamically. The consequences of dynamic replanning for safety are important: the safety property we monitor is compliance with a dynamically-chosen path,



(a) Example mission (b) Simulator

(c) Rectangle (d) Tight turns (e) Large clover

Figure 3.18: Implementation and environments built in AirSim.

which may not imply safety in the colloquial sense if the dynamic plan deviates significantly from the initial hard-coded plan. The use of sandbox control does however make the use of replanning important: sandbox controllers are only live if the use of the fallback option eventually results in recovery, i.e., it eventually leads to a state in which monitors succeed. If compliance to a fixed path is monitored, then deviation from the path could cause monitors to fail indefinitely, no matter how long the fallback braking controller is applied. Replanning results in a major improvement in liveness because it guarantees that when the vehicle deviates from the initial path, there still exists a path which leads back to the initial waypoint.

Failure rates and replanning aside, our results demonstrate that it was possible to develop a simulation which worked seamlessly with VeriPhy monitors that were synthesized from the model. The model and proof were iteratively developed using feedback from the resulting monitors. In favorable conditions, the failure rates were as low as 0.1% for the control monitor and 4.0% for the plant monitor, and in the worst case plant failures reached 66% while control failures remained under 5%.

Our initial hypothesis for the 66% failure rate was that high-speed drifting caused the physical assumptions of the model to be verified, but the lower failure rates achieved in the constructive reimplementation of VeriPhy (Chapter 8) suggest this is not the case, because the latter evaluation exhibits the same motion. While we do not have a conclusive explanation for the failure rate, arithmetic overflow is one possible explanation because classical VeriPhy uses fixed-precision integers and the Clover level happens to involve the largest numeric values. Secondly, an idiosyncrasy of our untrusted controller caused excessive steering at the handoff point between two path segments, but the algorithm was revised for smoother steering before the latter evaluation (Chapter 8) was performed. It is possible that the improvement in failure rates owes to improved steering behavior. Despite later changes in steering algorithm, we report the initial evaluation (Bohrer, Tan, et al., 2019) here for the sake of transparency.

Table 3.2: Average speed, monitor failure rates, plant violation rates, for AirSim and human driver in Rectangle, Turns, and Clover for Patrol missions.

	Avg. Speed (m/s)				
World	BB	**PD1**	PD2	PD3	Human
Rect	4.3	**6.32**	7.16	12.6	9.92
Turns	3.78	**3.95**	4.43	4.69	9.66
Clover	X	**29.5**	29.5	29.5	28.9

	Ctrl Fail.				
World	BB	**PD1**	PD2	PD3	Human
Rect	0.5%	**0.1%**	0.1%	0.19%	1.14%
Turns	1.0%	**1.0%**	1.1%	4.7%	3.61%
Clover	X	**0.2%**	0.2%	0.19%	0.29%

	Plant Fail.				
World	BB	**PD1**	PD2	PD3	Human
Rect	36.8%	**8.23%**	8.5%	14%	41.3%
Turns	18.6%	**3.95%**	6.8%	11%	21.1%
Clover	X	**66%**	66%	66%	48%

While failure rates should not be dismissed, a strong point of the experimental results is that every combination except for one completed the course, and our simple model was still general enough to account for these practical control decisions and physical dynamics in order to complete the course. Thus, our formal link from dL to CPS execution can be made to work in practice for nontrivial driving scenarios, despite its caveats. Table 3.2 gives the results of several controllers (PD1 is the slowest PD controller, PD3 the fastest) as well as a human pilot (the author) on all environments. Overall, the slow PD controller had the best monitor failure rates of any controller. Qualitatively, the results tell us that while perfect failure 0% rates are not always attainable for the present model, low failure rates are achievable in practice, where low control failure rates mean that the fallback controller will rarely need to engage and low plant failure rates mean that because physical assumptions are rarely violated, the formal guarantees provided by VeriPhy are applicable almost all the time. On the flip side, the non-zero failure rates also indicate the value of developing less conservative models in future work to reduce failure rates without compromising safety. The completion times for the automated controllers were competitive with the human pilot overall: the human performed better on the Turns map, which had high-angle low-speed turning, while the automated controllers slightly outperformed the human on the large Clover map where the human was less successful with low-precision high-speed turning.

The use of relative coordinates and replanning imply important limitations on our safety guarantees, because we ultimately monitor compliance with a dynamically chosen

path rather than a fixed one. During the test, the deviation between the fixed and dynamic paths was visually apparent during the cases that used the bang-bang tests, but not the others. In future work, there are several potential ways to provide stricter safety guarantees: absolute coordinates would more faithfully model compliance with an absolute path, while development of a fallback controller for steering (not just braking as in our current model) could make it possible for the fallback controller to recover reliably without the use of replanning. Additionally, the use of more nuanced dynamical models could improve compliance. For example, bicycle models, which provide a more detailed account of steering dynamics, have seen success in the robotics literature (Althoff & Dolan, 2014), yet their deductive verification would be a novel topic of study.

3.8 Discussion

We discuss limitations and challenges encountered in the case study, as well as how they are addressed in the remaining chapters of this thesis. We divide the discussion into two sections in order to discuss the lessons relevant to two different chapters. To guide the development of our Kaisar proof language (Chapter 7), we first discuss our experience using Bellerophon in the 2D system safety proof (Theorem 3.12) in order to assess general challenges in large-scale Bellerophon proofs, which are not specific to its usage in the context of VeriPhy. While no single proof is representative of all practical usage, the proof of Theorem 3.12 is a useful example because of its nontrivial length. The second section of the discussion (Section 3.8.3) covers lessons learned about the design and implementation of VeriPhy rather than the input proof language, in order to guide the eventual constructive reimplementation of VeriPhy in Chapter 8.

3.8.1 Bellerophon Language Primer

We briefly introduce Bellerophon (Fulton et al., 2017), the proof script language of KeYmaera X, because it is discussed in Section 3.8.2. A proof script is written in combinator style by composing atomic proof steps:

$$ps ::= \mathsf{tac}(\mathsf{arg}_1, \ldots, \mathsf{arg}_n) \mid ps; ps \mid <(ps, \ldots, ps) \mid (ps \mid ps) \mid ps^* \mid ps^k$$

Atomic built-in tactics tac can optionally take arguments of several types, the most common of which are formula positions and expressions, as well as strings. KeYmaera X supports a large, but fixed, vocabulary of built-in tactics including first-order sequent calculus rules, hybrid systems axioms of dL, general proof search, and automation for first-order arithmetic and differential equations. Expression arguments are written like strings, meaning they are delimited with double quotes[7], e.g., `"expression"`. Exact formula positions are nonzero integers (-1 for first antecedent formula, 1 for first succedent formula) and several other position *locators* are supported: `'L` and `'R` search the left or right-hand side of the sequent until an applicable position is found. A locator `loc` can be annotated with a formula using

[7]The following syntax is sometimes used in older proofs: `{`expression`}`

the notation `loc=="formula"`, meaning that the locator must find the given `formula` or an error will be raised. Likewise, the locator `loc~="formula"` must find a formula which unifies with the given `formula` or an error will be raised.

To apply tactics sequentially, we separate them with a semicolon. We write $(ps_1 \mid ps_2)$ to try ps_1 first, then try the alternative ps_2 instead if ps_1 fails. We write $<(ps_1, \ldots, ps_n)$ when there are n open subgoals to apply each ps_i to prove the ith subgoal. Most tactics expect a single subgoal, so it is standard to use $<$ immediately after branching. The repetition ps^* applies ps as many times as possible in sequence, until its next application would either fail or make no progress. The finite repetition ps^k applies ps k times.

A KeYmaera X proof file also lets the user declare or define functions, predicates, or hybrid programs before writing their proof. Function definitions have form:

```
Type name(Type1 arg1, ..., TypeN argN) = body;
```

Like in C, the return type is written first, and argument types, if any are written before names. Declarations of uninterpreted functions use the same syntax for type signatures but omit the body. Functions with no arguments are used to represent real-valued parameters that never change, like in the dL uniform substitution calculus. Predicate and hybrid program definitions and declarations are analogous, but respectively use `<->` or `::=` in place of the equals sign.

3.8.2 Bellerophon Proof Script Examples

The proofs of safety and liveness were written in Bellerophon, the proof script language of KeYmaera X. The proof scripts are of nontrivial length, with 279 manual steps for safety and 589 manual steps for liveness. Thus, we can look to the safety and liveness proof scripts as examples of Bellerophon proving at scale in order to assess the limitations of Bellerophon proofs in the large. The final model used here features units of measure which are not discussed in Section 3.7. We simply multiply by constant scaling factors `kf` and `df` when converting curvatures and distances.

The KeYmaera X modeling language does support some important features which aid scalability. Notably, the Logic-User can define named functions and predicates which are then used in the model. The following snippet defines a predicate `onCircle` which determines whether a point (x, y) is on a circle of curvature k through the origin, and does so using helper functions `sq` and `circle`:

```
Real sq(Real x) = x*x;
Real circle(Real x, Real y, Real k) = k*(sq(x)+sq(y))-2*x*kf()*df();
Bool onCircle(Real x, Real y, Real k) <-> circle(x,y,k) = 0;
```

Above, `kf` and `df` are unit-conversion scaling factors (both equal to 10), which are not needed in a high-level unitless model but become necessary as soon as we wish to interact with a simulator that has concrete units.

While definitions provide an important abstraction mechanism, the implementation of that abstraction in Bellerophon can at times be leaky in practice. The definition mechanism

in Bellerophon is implemented with uniform substitutions (Platzer, 2017a) that replace uninterpreted symbols with their definitions. A uniform substitution approach makes it easy to expand definitions (using a special tactic `expandAllDefs` in Bellerophon), but there is no provided tactic for reversing the expansion of definitions; if possible at all, unexpanding definitions would require inconvenient manual cuts at the least. This phenomenon is not problematic on its own, but can interact awkwardly with a major use case of Bellerophon: performing proof steps in the KeYmaera X UI and extracting proof scripts after-the-fact. The automatic extraction of proof scripts often results in fully-expanded, unreadable, unmaintainable expressions. As an intentionally-unreadable example[8], we show a single loop invariant step using our loop invariant `J(xg, yg, k, v, vl, vh)` applied at position `1`, which is highly verbose when expanded:

```
loop("v>=0&abs(k)*eps()<=100*1&((k*(eps()*eps())-2*100*eps())*(10*10)
  < k*(xg*xg+yg*yg)-2*xg*100*10
&k*(xg*xg+yg*yg)-2*xg*100*10 < (k*(eps()*eps())+2*100*eps())*(10*10))
&(0<=vl&vl < vh&A()*T()<=10*(vh-vl)&B()*T()<=10*(vh-vl))
&(v<=vh|(1*100*(1*100)+2*eps()*abs(k)*1*100+eps()*eps()*(k*k))
  *(v*10*(v*10)-vh*10*(vh*10))<=2*B()*(yg-10*eps())*(100*100)*(10*10)
  |(1*100*(1*100)+2*eps()*abs(k)*1*100+eps()*eps()*(k*k))*(v*10*
    (v*10)-vh*10*(vh*10))<=2*B()*(abs(xg)-10*eps())*(100*100)*(10*10))
&(vl<=v|(1*100*(1*100)+2*eps()*abs(k)*1*100+eps()*eps()*(k*k))
  *(vl*10*(vl*10)-v*10*(v*10))<=2*A()*(yg-10*eps())*(100*100)*(10*10)
  |(1*100*(1*100)+2*eps()*abs(k)*1*100+eps()*eps()*(k*k))*(vl*10*
    (vl*10)-v*10*(v*10))<=2*A()*(abs(xg)-10*eps())*(100*100)*(10*10))
", 1);
```

An additional class of maintenance challenges regards formula position. That is, it can be challenging to give a clean and maintainable specification of which formula a rule should be applied to. At their simplest, Bellerophon scripts use numbers to indicate sequent positions (positive for succedent, negative for antecedent), which suffices for simple cases such as rule applications that operate on a single succedent formula at position 1. For example, the following step says to prove a conjunction on the right by proving each conjunct. In the example, `tacL` and `tacR` stand for Bellerophon tactics that prove the respective conjuncts.

```
andR(1); <(tacL, tacR)
```

However, positional arguments become fragile when used to select assumptions from large contexts, whose contents frequently change between versions of a model and even versions of KeYmaera X. The following step case-analyzes a disjunction, which happens to be the 11th assumption.

```
orL(-11)
```

Positional proof steps will break whenever the position of an assumption changes, and moreover cannot be read by humans except side-by-side with the proof state of a running

[8]Where `eps` is plaintext notation for ε and `abs` is absolute value

KeYmaera X prover. To alleviate this issue, Bellerophon allows proof steps to specify a locator (formula, formula pattern, or `'L` or `'R`) with a formula: if the locator finds no matching formula, the prover reports an error. The following weakening step `hideR` expects a particular formula at succedent position 1:

```
hideR(1=="[{xg'=-v*k*yg/(100*10),yg'=v*(k*xg/(100*10)-1),v'=a,t'=10
&((v>=0&t<=T())&t>=0&(k*(eps()*eps())-2*100*eps())*(10*10)
< k*(xg*xg+yg*yg)-2*xg*100*10&k*(xg*xg+yg*yg)-2*xg*100*10
 < (k*(eps()*eps())+2*100*eps())*(10*10))&10*v+a*(T()-t)>=0}]
 (1*100*(1*100)+2*eps()*abs(k)*1*100+eps()*eps()*(k*k))*
 (v*(T()-t)*10+a/2*((T()-t)*(T()-t))+(vl*10*(vl*10)-(v*10+a*(T()-t))
 *(v*10+a*(T()-t)))/(2*A()))<=(yg-10*eps())*(100*100)*(10*10)")
```

Several issues are evident in the step above. The formula is quite long because it is fully expanded, and because weakening steps are usually auto-generated by clicking on the UI on a proof state where definitions have been expanded, it would be difficult to recover a concise, abbreviated formula, thus this verbosity is typical of auto-generated steps. Unless the Logic-User manually rewrites the step to use definitions, the step will have to be changed whenever changes to the model or proof cause a different formula to appear at position 1. Even when the Logic-User does choose to manually rewrite the step, it may not always be trivial to do so because the content of the context can vary greatly from the text of the model. In our `hideR` example, the domain constraint of the ODE contains several conjuncts that were introduced by differential cuts and do not appear explicitly in the source model. Before the Logic-User can decide how to write the proof step, they must first *trace* the relationship between a model and its proof. Without knowing how to trace that relationship, a novice user might read a line of Bellerophon and have no idea which model statement is currently being proved.

Thus far we have identified expansion of definitions and applications of the cut rule as sources of traceability challenges. Just as definition expansion works by substitution, reasoning about ODE solutions, deterministic assignments, and nondeterministic assignments is often done using substitution and/or renaming, so that assignment-like reasoning is another source of traceability challenges in a similar vein to definition expansion. Because our 2D driving case study does not use solution-based reasoning for ODEs and the only assignments which appear are simple, assignments were not the main challenge in our case study, but are worth mentioning because they can present a challenge in examples that rely heavily on assignment and ODE solutions. We discuss the modest impact of assignments on our case study in order to explain the more significant impact they can have in other cases. Consider the following snippet from our proof:

```
    hideL(-21=="abs(k_0)*eps()<=100*1");
... MR("a>=0&10*v+a*(T()-t)<=10*vh", 1);
```

The former line hides a fact which mentions k_0, a variable which does not appear in the model but is rather automatically generated to refer to the value of k "in the old state." The later line (after several omitted intermediate steps) uses monotonicity reasoning to prove the stated formula as a lemma "in the new state." The Demonic nondeterministic

assignment $k := *$ featured in our model has a modest impact on the proof script because reasoning about Demonic nondeterministic assignment only requires renaming and it is relatively easy for a reader to understand that k_0 means an old value of k. Traceability is more difficult if $k := *$ is assigned many times, leading to many subscripted variables k_i, or if deterministic assignments or ODE solutions are used. To reduce variable count, KeYmaera X prefers to reason about deterministic assignments and ODEs by substitution whenever possible. If our controller model explicitly computed acceleration `a` using some complicated function `f(v, vl, vh,xg, yg, k)`, then variable `a` would be eliminated to the far more complicated term `f(v, vl, vh,xg, yg, k)`, which would then need to be written in the formula of the `MR` step. The alternative approach (close to the approach under the hood by Kaisar in Chapter 7) is to introduce a ghost variables a_0 analogous to the use of k_0 and remember its relationship to the current value of a, if any. Without the labeling syntax that will be provided by Kaisar, however, renaming and substitution both deviate from the text of the model: the former causes the `hide` step to deviate while the latter would cause the `MR` step to deviate.

Lastly, typical Bellerophon proof approaches have particularly high maintenance burdens for arithmetic proofs. It is a common idiom to introduce simplified arithmetic assumptions using `cut` and then weaken all unnecessary assumptions with `hideL` before applying an arithmetic solver. Because performance of arithmetic solvers depends greatly on the number of assumptions used, the addition or modification of a single assumption could lead many seemingly-unrelated arithmetic steps to slow down or hang because the set of assumptions supplied to the solver has increased. Such slowdowns are typically resolved by inserting `hideL` calls which weaken the new assumption for proof branches in which it is irrelevant. The problem is that many such changes could be necessary and those changes could be nonlocal. In practice, the number of `hide`s can be significant: 154 in our example, as many as 11 per arithmetic call. To alleviate this issue, the latest release of Bellerophon provides a tactic (named `using`, but distinct from the keyword of the same name introduced in Chapter 7) which searches the context for a list of formulas and weakens all others. However, Bellerophon's `using` tactic shares the limitations of Bellerophon's locators because it is search-based: during maintenance, many expressions appearing in a model will be modified, which will result in the Bellerophon proof failing to check unless all corresponding expressions in the proof script are maintained in lockstep.

In Chapter 7, we will develop a new, structured proof design to resolve the above Bellerophon issues, using features such as named assumptions, annotation-based proof, and positive-style (as opposed to negative or weakening-style) arithmetic proofs as a default. In Bellerophon's defense, the comparison between our language and Bellerophon will not be one-to-one because we do not pursue integration with the KeYmaera X UI. As we have discussed, the limitations of Bellerophon are most salient when using it in combination with the UI. However, those limitations are founded in more basic phenomena: changing state causing changes in the context, the irreversibility of definition substitutions, and an unstructured representation of contexts. That context representation makes hard-to-maintain proof idioms (such as positional references and negative, `hideL`-based arithmetic proofs) easier than their more maintainable counterparts such as search-based. Proofs with named facts are particularly challenging for Bellerophon's context representation,

which is why they are unsupported in Bellerophon. The basic phenomena listed above will inspire our language, resulting in a design that remains fundamentally distinct from Bellerophon even as Bellerophon's definition mechanism and formula locators improve over time. Moreover, while it is possible for a Bellerophon expert to write proofs in a style which minimizes the problems discussed here, it is a design goal of Kaisar to make such problems outright difficult or even impossible to encounter, regardless of expertise. Though Bellerophon features such as locators and the `using` tactic can support expert users in improving proof maintainability, Kaisar targets the crucial goal of supporting *all* proof authors, including novices, in writing proofs that are easy to read and maintain.

3.8.3 Limitations of Classical VeriPhy

In testing the VeriPhy sandbox, some of the challenges we encountered were incidental, while some can only be addressed with deeper changes. By simple virtue of testing a larger model and proof, we identified and fixed a number of bugs where VeriPhy made overly strict assumptions on the input model or used excessively brute-force, slow proof approaches. This same debugging process also exposed limitations which still persist in classical VeriPhy. Before discussing limitations at greater length, we discuss the KeYmaera X tactic for proving safety of VeriPhy sandbox controllers in order to observe the challenges in proving sandbox safety. We give a very high-level outline in Fig. 3.19. The outline mixes Bellerophon code with pseudocode and is meant for explanatory use only.

The construction of the `sandboxTactic` is a function of the Bellerophon tactic that proves safety of the input system (`safeTac`), the input system safety theorem statement (`safeFml`) and `fallback` program. The first line pattern-matches on `safeFml` to extract the `init`ial conditions, `controller`, plant model `ode&odeDom;`, and `safety` postcondition. Because double quotes are used to indicate dL expressions in the Bellerophon syntax, the first line uses double quotes on the left-hand side as pseudo-code for the pattern-match on the dL expression datatype. The key step of sandbox proof generation and a key source of fragility in the classical VeriPhy implementation is the following line. Function `proveWithListener` executes `safeTac` to prove `safeFml`, but listens in on the execution of the tactic interpreter to collect additional `listenerInfo`, specifically loop and differential invariants as well as proofs of important subgoals (`lemmas`). The safety theorem statement and invariants provide enough information to determine the `monitors`[9]. Those lemmas will be reused in the proof of sandbox safety. The `base` case shows that `loopInv` holds at the start of the loop, the `post`condition step shows that `safe` follows from `loopInv`, `ctrlToInv` shows that `loopInv` holds after the controller, and `plantToInv` shows that `loopInv` holds after the plant, roughly meaning `plantMon -> ctrlInv`. Note that `loopInv` must be proved to hold both after the controller and after the loop body; this is a requirement of the sandbox tactic generator, albeit a natural assumption because a loop invariant must be true after all executions of the plant, including those of duration 0. The author of `safeTac` is required to use a

[9]We included `plantMon` here to emphasize that it can be computed from the theorem statement and invariants. It is not actually mentioned explicitly in the following lines because the key lemma `plantToInv` about the plant monitor has already been proved in the source model.

variant of the `loop` tactic that additionally proves that the loop invariant holds between the control and plant. An additional lemma `fallbackCheck` is proved (without listeners, using automation) so that the control monitor provably holds after the fallback controller. Once the necessary lemmas have been extracted or proved, the actual tactic expression for the sandbox safety proof begins.

```
def sandboxTactic(safeTac, safeFml, fallback) = {
  let "init -> [{ctrl;ode&odeDom;}*]safe" = safeFml
  let listenerInfo = proveWithListener(safeTac, safeFml)
  let (loopInv, diffInvs) = listenerInfo.invariants
  let (plantMon, ctrlMon) = monitors(safeFml, inv, diffInvs)
  let (base, post, ctrlToInv, plantToInv) =  listenerInfo.lemmas
  let fallbackCheck = prove(auto,"loopInv->[fallback]ctrlMon")
  chase(1); ...; /* start of tactic */
  loop("loopInv", 1); <(
    use(base) // base case
  , use(post) // postcondition
  , /* loop body */ ...;
    generalize("loopInv"); <(
      // untrusted controller and fallback
      choiceb(1); andR(1); <( ...;
        // untrusted controller
        cut("[ctrl]loopInv",1);
          <(...;chase(1); ...; prop, use(ctrlToInv))
      , /* fallback */ ... ;
        cut("[fallback]ctrlMon");
        <(...; chase(1); ...; prop, use(fallbackCheck)))
    , /* plant */
      use(plantToInv)))
}
```

Figure 3.19: Outline: VeriPhy sandbox safety tactic.

We elide large, nontrivial sections of the proof using dots (`...`) in order to focus on proof highlights which are essential to the following discussion. We use a variety of built-in tactics such as `chase` (hybrid program simplification), `prop` (propositional reasoning), the `cut` rule, `andR` (conjunction on the right), `choiceb` (box choice formula), `loop` (invariant reasoning), and `generalize` (monotonicity).

The main proof is a `loop` invariant argument using the same invariant `loopInv` from the system safety model. To prove the base case and postcondition step, we `use` the lemmas which were already proved in the system safety proof. The rest of the tactic proves that the loop body preserves the invariant. The proof of the body begins with the `generalize` tactic which reasons by monotonicity, showing that the invariant holds after the controller and that its truth at the beginning of the plant implies the invariant at the end of the plant. The first branch following `generalize` considers the controller model

of the sandbox, which consists of one branch for untrusted control and one branch for the fallback controller. The crucial step for the untrusted controller is to show the invariant (`cut("[ctrl]loopInv",1)`) using the lemma `use(ctrlToInv)` and a number of transformation steps which we omit for clarity. Likewise, the crucial step for the fallback controller is to show the invariant using the lemma `fallbackCheck`. Once the proof of the control model is complete, `plantToInv` is used to show that the invariant is preserved by the (discrete) plant model.

We discuss the implications of this sandbox tactic outline regarding issues of robustness in classical VeriPhy:

- The model is in a fixed format: control-plant loops with a single occurrence of the ODE are required.
- The proof is in a restricted format: the loop invariant must be proved to hold between the control and plant.
- Because the lemma `plantToInv` is extracted by locating a single application of the differential weakening tactic (`dW`), the single ODE proof must end in `dW`.
- All `diffInvariants` from the proof are flattened into one collection. Even for a single ODE proof, flattening the invariants can be inaccurate when nested differential cuts are used. When showing a differential cut ϕ, Bellerophon permits performing a case split and proving distinct cuts ψ_1 or ψ_2 as lemmas to ϕ in different cases. Because of flattening, the sandbox tactic would generate a monitor which expects both of ψ_1 and ψ_2 to hold unconditionally, thus forgetting the branching structure.
- The fallback controller lemma `fallbackCheck` is proved using full automation. If the controller is too complex or its safety argument too subtle, the user has no way to provide a manual proof. This poses a scalability issue for complex fallbacks.
- The monitor correctness lemma (Lemma 3.1 from Section 3.3) is also proven automatically, though we omitted it from the outline to simplify the presentation. Large controller models could pose a challenge in principle for an automated proof of Lemma 3.1, which is an automated proof of a theorem about the user-provided *non-fallback* controller model.
- The proof relies on several built-in tactics which can be unpredictable because of their general-purpose, heuristic nature. The `chase` tactic, in particular, is applied to complex programs in the sandbox safety proof. It has historically proved challenging to debug the use of `chase` in VeriPhy and determine whether a given error message constitutes a mistake in the system model or a bug in VeriPhy. Usability for the Logic-User suffers if it is hard to distinguish flawed inputs from those which are merely unsupported or merely expose bugs in the sandbox tactic. As a historical example, early development versions of VeriPhy unexpectedly failed on models with multiple controller branches, but the error messages simply indicated a failure in the sandbox tactic without giving Logic-User insight into the fact that the VeriPhy implementation was at fault.

We can ascribe a common theme to the limitations above: classical VeriPhy makes surprising assumptions on model and proof structure, often silent assumptions which lead

to tactic failures when not satisfied. It is unsurprising that classical VeriPhy makes these awkward and implicit assumptions because the Bellerophon tactic language admits a wide variety of proof styles and the structure of a tactic can in general vary wildly from the structure of the input system. It would be convenient for the Logic-User if arbitrarily nested loops and arbitrarily many ODEs were supported, but there is no straightforward way for classical VeriPhy to inspect a Bellerophon tactic and match up different loops and ODEs from a model with corresponding sections of a complex Bellerophon program. Instead, the sandbox tactic opts to assume a single loop and ODE in the model, then hope that loop and ODE rules are only used in the tactic in well-behaved ways. In the case of the fallback, no manual proof is provided as input, let alone a well-structured proof.

The theme of structuring is not only useful for explaining limitations we discovered during the development and use of classical VeriPhy, it is also useful for explaining a new feature we desire for the constructive successor (Chapter 8): synthesis of whitebox controllers from liveness proofs. For controllers where direct safety and liveness proofs are possible, we would rather have a controller which is both safe *and* live rather than rely on a sandboxing approach which sacrifices liveness for safety. However, before pursuing synthesis of whitebox controllers from liveness proofs, we would like our input format to make it easy to state a model and its liveness theorem. However, our experience stating and proving a liveness theorem (Theorem 3.13) reveals that it is surprisingly difficult to state a theorem that captures both safety and liveness in dL, especially if we wish for the theorem statement to be both correct and suitable for automated consumption. Our theorem statement (Theorem 3.13) suggests that a formula of shape $[\alpha](\mathsf{safe} \wedge \langle\alpha\rangle\mathsf{goal})$ might correctly combine safety and liveness, where safe holds when the current state is in some safe region and goal holds when some liveness objective has been met. However, such theorems do not faithfully capture the dynamics of a system featuring a safe, live controller. Such a statement divides system dynamics into two phases: the first phase shows safety only while giving up all system control, the second shows liveness when we have total control. We desire a controller that satisfies a stronger notion of correctness: a controller should remain correct as system execution alternates repeatedly between that which we control (the controller) and that which we do not (the plant).

When stating liveness theorems of shape $[\alpha](\mathsf{safe} \wedge \langle\alpha\rangle\mathsf{goal})$, an additional hiccup is that total control in the liveness proof is often far too strong an assumption. It is for that very reason that our theorem statement (Theorem 3.13) uses distinct programs α and β for the modalities $[\alpha]$ and $\langle\beta\rangle$: the program β appearing in the liveness modality should express the fact that we do *not* get to choose our waypoint, even when we *do* choose the acceleration. Theorem statements in the style of Theorem 3.13 are poorly suited for automated consumption in the sense that the consumer would need to reconcile the two programs α and β that are syntactically distinct but fulfill a morally equivalent purpose.

In short, an overarching theme of the following chapters is to provide the foundations and structuring principles that make a more robust and general version of VeriPhy possible. In Part II in particular, great emphasis will be placed on foundations in order to provide a thorough, systematic basis for synthesis in the reimplementation of VeriPhy, thus ensuring from the start that it does not encounter the same robustness issues and format restrictions that arose in this chapter's implementation. The new foundations and principles of

Part II serve a dual purpose because they will provide a robust basis not only for sandbox controller synthesis but for new features including whitebox controller synthesis and, in the input language, structured proof principles. Once the new foundations are complete, Part III exploits those foundations for the reimplementation of VeriPhy along with a new, structured proof language Kaisar that serves as its input.

The key structuring principles for a better synthesis tool include tight correspondences between proofs and code, between liveness and control, and between proof and model. In Part II, we develop the constructive logic of hybrid games to resolve two of these correspondences. Game logics, in contrast to normal dynamic logics, provide a common proof artifact for safety and liveness. A *constructive* game logic ensures that every safety proof corresponds cleanly to monitoring code and every liveness proof[10] corresponds cleanly to control code. The natural-deduction calculi developed in Part II will also foreshadow a clear connection between proof text and model text, but that connection will be provided conclusively in Chapter 7, which presents the structured Kaisar proof language, which is motivated by the limitations of Bellerophon that the Logic-User encountered.

Arithmetic will be another major theme in both our development of constructive game logics (Part II) and our reimplementation of VeriPhy (Chapter 8). Rather than interval arithmetic, our constructive logic for hybrid games (Chapter 5) will use a more powerful foundation: constructive real numbers that support arbitrarily precise approximations. Constructive VeriPhy (Chapter 8) will take the middle ground between classical VeriPhy and Chapter 5 by using arbitrary-precision rational intervals, which are less powerful than constructive reals, but easier to implement than constructive reals and certainly more powerful than the integer intervals used here. Rational intervals will, for practical purposes, eliminate the overflow errors experienced in classical VeriPhy. Increasing arithmetic precision is important because our arithmetic constraints both required the Engineer to choose units of measurement with extreme care and made debugging more difficult if the Engineer encountered an overflow that they did not expect.

In all, we will remove limitations on model and proof structure, limitations on arithmetic, and the limitation to safety of sandbox controllers. Removing model and proof limitations will reduce the need for maintenance and rework by the Logic-User. The Engineer will benefit from relaxed arithmetic guarantees especially, but also from the availability of whitebox control and perhaps indirectly from unrestricted modeling. In exchange for the new benefits on part of the Logic-User and Engineer, the constructive VeriPhy implementation presented in Chapter 8 will argue correctness informally by appeal to the major theorems of Part II rather than provide an extensive chain of formal artifacts for a rigorous safety theorem. The Kaisar (Chapter 7) language's complexity makes even a paper proof significantly more involved than it was in the case of this chapter, while the entirely new foundations used in Chapter 8 make full formal artifacts even more difficult to attain.

Ultimately however, there is no reason to despair. While the implementations of VeriPhy from this chapter and Chapter 8 both aim to resolve the interests of the Logician, Engineer, and Logic-User, they ultimately settle on different tradeoffs. Each is useful,

[10]In particular, dL allows case analysis on undecidable real arithmetic formulas, yet controller synthesis must restrict controllers to only use decidable case analysis principles.

with the present chapter being most useful when the Logician's concerns are given top priority and the implementation from Chapter 8 being most useful when the Engineer and Logic-User are given higher priority.

Part II

Constructive Game Logics

Chapter 4

Constructive Discrete Game Logic

4.1 Introduction

Part I concluded (Section 3.8) with a discussion of the limitations of classical VeriPhy, which we now review in order to motivate constructive game logic (CGL). A major practical limitation of classical VeriPhy was its reliance on fixed formats for models and proofs; without a systematic interpretation of dL proofs as programs, synthesis often fails with cryptic error messages when the input lies outside the supported fragment, a fragment which does not even have a precise definition. A second major limitation, albeit one that exists by design, is that classical VeriPhy uses blackbox monitor-based controllers as its *only* controller paradigm. Sandbox controllers are limiting in that they must sacrifice liveness to ensure safety whenever their untrusted controller is noncompliant with their controller monitor. An additional limitation is that they still require the Engineer to write her own untrusted controller which is then sandboxed. To overcome the limitations of sandboxes, additional support for synthesis of concrete whitebox controllers would be a desirable feature both because synthesis of whitebox controllers can more readily preserve *liveness* in addition to safety and because they relieve the Engineer from the obligation of writing her own controller.

The above limitations motivate the development of a new logical foundation which both serves as the basis for a reimplementation of the VeriPhy approach and stands as a first-class theoretical contribution in its own right: constructive game logic (CGL). The foundations of CGL are developed in Part II of this thesis before serving as the basis for a reimplementation of VeriPhy in Part III. Because Part II focuses on developing new foundations, it is primarily written from the Logician's perspective, but we will occasionally mention the Engineer and Logic-User in order to point out how the new foundations support them. To ease the introduction of CGL, Chapter 4 will first introduce *discrete* CGL, which is then extended to hybrid games in Chapter 5. As the name suggests, hybrid games are the extension of hybrid systems to games, or equivalently the extension of games to hybrid systems.

We begin our discussion of constructive games by explaining why the limitations of classical VeriPhy are sufficient motivation for new foundations and why CGL is the right foundation to develop. Classical VeriPhy's implementation fragility and reliance on fixed

input formats stem from the fact that its processing of models and proofs is insufficiently systematic. Though classical VeriPhy's high-level architecture is built on generic, systematic concepts such as sandboxing and verified compilation, the input proof language is so open-ended and allows proof and model structure to differ from each other so greatly as to make a systematic treatment of the entire proof language untenable. Even the underlying classical logic dL, though systematic in the development of its semantics and proof calculus, *does not* feature a systematic treatment of the crucial challenge that underlies synthesis of code from verified models: all proofs must correspond to programs. That is, our logic must be constructive. The reason we pursue a foundational approach is that a robust implementation of VeriPhy demands logical foundations that provide a systematic correspondence between proofs and code and which admit a systematic, synthesis-friendly proof language design. By developing new foundations, we provide the necessary systematic treatment.

Constructivity is crucial to a systematic treatment of synthesis; games are crucial to systematic synthesis of concrete whitebox controllers specifically. The fundamental advantage of games over systems is that they allow free alternation between existential and universal choices: a game model can easily specify which aspects of the model are ours to control and which are outside our control, then a game proof resolves our existential choices. A typical game model provides an existential specification for control, i.e., it specifies that it is up to our proof to resolve our control decisions (which must provably achieve their stated goals for *all* actions of the opponent). A game proof then systematically specifies the control algorithm, which can be translated to executable code by a synthesis algorithm. Moreover, *theorem statements* in game logic (GL) can also freely combine existential and universal quantification, meaning they excel at proving safety and liveness together in a single proof artifact. Basing synthesis on foundations which readily combine safety and liveness proofs greatly simplifies the task of synthesizing code that is simultaneously safe and live.

Constructivity becomes particularly crucial in combination with games, particularly games which feature existential specifications of control. Constructive game proofs contain concrete existential witnesses, thus they contain concrete, executable control code. In contrast, classical game proofs (e.g., using the original GL (Parikh, 1983)) would show the existence of correct control decisions which in principle may depend on undecidable properties and even in practice may not be readily synthesized from the proof artifact. When Chapter 5 develops CdGL, a CGL for hybrid games, the use of real numbers will make constructivity even more crucial to synthesis because, as that chapter will explain, constructive proofs about reals, when compared to classical proofs about reals, greatly simplify the challenge of simultaneously preserving safety and liveness in synthesized code which performs arithmetic case analyses. We highlight these crucial benefits of constructivity in order to emphasize that whenever CGL accepts fewer proofs than GL does, and thus requires additional care on the part of the Logic-User, the added restrictions and added efforts have the concrete benefit of enabling synthesis of code from *every* proof. For CdGL specifically, the concrete benefits will also include improving the safety and liveness of synthesized arithmetic code.

In addition to the fundamental advantages of constructivity and games in general, our eventual tool developments Part III will benefit from our choice to develop a natural-

deduction calculus specifically, which is a widely-used among constructive logics. A canonical natural-deduction[1] proof closely follows the structure of the game being verified, simplifying both synthesis tool implementation and high-level proof language design.

Beyond its motivations within the context of CPS, CGL is a first-class contribution of the thesis and is of broader theoretical interest. The broader theoretical motivation for CGL stems from the observation that program logics (such as GLs) and constructivity are both fundamental tools in the theory of programming languages, yet constructive program logics have received surprisingly little study (Section 4.3). Game logics (Parikh, 1983; Platzer, 2015a) are a particularly expressive class of program logics that extend regular *dynamic logics* (DLs) (Pratt, 1976) with games; just as DLs include Hoare calculi (Hoare, 1969) as fragments, so do game logics. Thus, CGL includes constructive first-order DL and Hoare calculus as fragments so that our results for CGL apply to those fragments as well. Those fragments, let alone games, are also under-studied (Section 4.3).

Part II studies constructive program logics in depth and consists of three chapters. Chapter 4 develops a CGL for discrete games; when studying game logic, our results apply to dynamic logic and Hoare logic, which can be expressed as fragments, while focusing on generic CGL yields a presentation which might easily be extended to domains beyond CPS. Chapter 5 specializes CGL to hybrid games, yielding a variant called CdGL which is suitable for verification of CPSs. Chapter 6 develops a refinement calculus for CdGL and provides a connection from CdGL back to dL, paving the way for implementation work in Part III. The practical perspective will be addressed in Part III, not Part II.

While constructive program logics have received relatively little study, constructive logics are widely used for verified *functional* programming. Programming in constructive logic relies on the *Curry-Howard correspondence* (Curry & Feys, 1958; Howard, 1980), wherein propositions correspond to types, proofs to functional programs, and proof term normalization to program evaluation. Higher-order constructive logics (Coquand & Huet, 1988) obey the Curry-Howard correspondence and are used to develop verified functional programs. Programming languages can also have their semantics formalized in constructive proof assistants such as Coq (*Coq Proof Assistant*, 1989), after which constructive proof rules from the metalogic can be used to reason about programs. Both are excellent ways to develop verified software, but we study something else.

We study the computational content of a program logic *itself*. Every fundamental concept of computation is expected to manifest in all three of logic, type systems, and category theory (Harper, 2011). Because dynamics logics (DLs) such as GL have shown that program execution is a first-class construct in modal logic, the Logician has an imperative to explore the underlying notion of computation by developing a constructive GL with a Curry-Howard interpretation.

The computational content of a proof is especially clear in GL, which generalizes DL to programmatic models of zero-sum, perfect-information games between two players, traditionally named Angel and Demon. Both normal-play and misère-play games can be

[1] As opposed to the Hilbert axioms and classical sequent calculus rules used in KeYmaera X, which have different strengths. Hilbert systems emphasize simplicity of axioms, sequent calculi have advantages for automated proof search, and natural-deduction systems achieve a close relationship between proofs and functional programs.

modeled in GL. In both GL and CGL, the diamond modality $\langle\alpha\rangle\phi$ and box modality $[\alpha]\phi$ say that some player has a strategy to ensure ϕ is true at the end of α, which is a model of a game. The difference between classical GL and CGL is that classical GL allows proofs that exclude the middle, which correspond to strategies which branch on undecidable conditions. CGL proofs do not exclude the middle, thus they correspond to strategies which are *effective* and can be executed by computer. Effective strategies are crucial because they enable the synthesis of code that implements a strategy. Strategy synthesis is crucial because even simple games can have complicated strategies, and synthesis provides assurance that the implementation correctly solves the game. A CGL strategy resolves the choices inherent in a game: a diamond strategy specifies every move made by the current player, while a box strategy specifies the moves, if any, made by the non-current player. Note that our own terminology for players differs from standard terminology in the literature; readers familiar with standard terminology are particularly encouraged to read Section 4.2 for a discussion of how and why our terminology differs.

In developing *Constructive Game Logic* (CGL), adding constructivity is at the same time a huge change and a small change. On the one hand, the development of CGL includes the development of entirely new semantics and a new natural deduction calculus; on the other hand, our example proofs suggest that the difference between classical and constructive (discrete) game proofs is almost entirely transparent to the proof author in practice because non-contrived discrete game proofs rarely branch on undecidable properties. Thus, CGL gives us the theoretical advantages of a constructive foundation while providing a proof-writing experience which is expected to agree closely with its classical predecessor.

We outline the numerous ways in which constructivity requires new theoretical developments for CGL. We provide a natural deduction calculus for CGL (Section 4.7) equipped with proof terms and an operational semantics on the proofs (Section 4.9), demonstrating the meaning of strategies as functional programs and of winning strategies as functional programs that are guaranteed to achieve their objective no matter what counter-strategy the opponent follows. While the proof calculus of a constructive logic is often taken as ground truth, we go a step further and develop a realizability semantics for CGL as programs performing winning strategies for game proofs, then prove the calculus sound against it (Section 4.8). We adopt realizability semantics in contrast to the winning-region semantics of classical GL because it enables us to prove that CGL satisfies novel properties (Section 4.10). Compared to the type-theoretic semantics which we will develop in Chapter 5, realizability semantics are a natural choice for a first, most general presentation of CGL because realizers easily admit an open-ended description: our semantics capture the computational constructs needed in every CGL without excluding features which may be desired by domain-specific variants such as CdGL. The proof of our Strategy Property (Theorem 4.20) constitutes an (on-paper) algorithm that computes a player's (effective) strategy from a proof that they can win a game. This is the key test of constructivity for CGL, which would not be possible in classical GL, because GL proves classical existence of winning strategies, which need not be effective. We show that CGL proofs have *two* computational interpretations: the operational semantics interpret an arbitrary proof (strategy) as a functional program which reduces to a normal-form proof (strategy), while realizability semantics interpret Angel strategies as programs which defeat arbitrary De-

monic opponents. Normal-forms help characterize the fragment of proof terms which must be handled in any static computation which consumes proofs, while the realizability semantics characterize how games are executed dynamically.

This chapter emphasizes the theoretical motivations for CGL and a generic presentation of them. The example models of this chapter (Section 4.5) are introductory discrete games which can be understood without a background in CPS. In Chapter 5, hybrid game verification in CdGL will be applied to an example CPS, before the practical advantages of CdGL are exploited in Part III, whose synthesis algorithm is inspired by Theorem 4.20 from this chapter.

4.2 Player Terminology and Player Constructivity

We compare the terminology used for players in this thesis to various terminologies which are used in the literature. The comparisons are summarized in Table 4.1. The terminologies vary between different logics which are discussed in greater detail in Section 4.3.

A major difference between the logics is the choice of which players are classical vs. constructive. For that reason, we elaborate on what it means for CGL's Demon to be classical before describing the competing terminologies. As will be discussed in Section 4.6.2, even a classical Demon must present constructive proofs when playing a test, because CdGL is *proof-relevant*: Angel's strategy may branch on the proof which Demon gave to her. Rather, Demon's classicality manifests in nondeterministic assignments, choices, and loops. When Demon resolves a nondeterministic assignment, he is permitted to classically choose any real number (which Angel, being constructive, is then only permitted to inspect using inexact comparisons), as opposed to constructively choosing a number by presenting a term which computes it. When Demon chooses which branch of a Demonic choice to take, he makes his choice classically, i.e., his branches need not be decidable. His choice of when to terminate a Demonic loop is classical in the same sense. When Demonic ODEs are added in Chapter 5, the choice of ODE duration will be classical in the same sense as nondeterministic assignments, but the proof of the domain constraint will be constructive in the same sense as tests, so that Angel can inspect its proof.

The names used in each logic can be divided into three categories. Discrete GL and dGL have two classical players named Angel and Demon, respectively. By making both players classical, GL and dGL admit classical semantics, which may be desired for simplicity, yet they unfortunately permit Angel to use strategies which have no interpretation as programs. The logics CGL and CdGL introduced in this thesis (the latter in Chapter 5) name the players Angel and Demon, but make Angel play constructively while Demon has more freedom because he plays classically. It is crucial to make Angel constructive so that her strategies will only ever use constructs for which code can be synthesized; as examples, any condition on which she branches must be decidable and whenever she chooses the value assigned by a nondeterministic assignment, she must give an explicit computation for it. Conversely, classical Demons are an aesthetic choice which, in our experience thus far, had no major implications on our semantics and proof calculus but helpfully emphasized that Demon need not be a computer, particularly in the setting of CPS. Of course, Angel could be

Logic	Player Names	Classical Players?
GL, dGL	Angel, Demon	Both
CGL, CdGL	Angel, Demon	Demon
dDG$\mathcal{L}$	Verifier, Falsifier	Both

Logic	$\langle\alpha\rangle\phi$ Meaning	$[\alpha]\phi$ Meaning
GL, dGL	Angel moves first, Angel wins	Angel moves first, Demon wins
CGL, CdGL	Angel moves first, Angel wins	Demon moves first, Angel wins
dDG$\mathcal{L}$	Verifier moves first, Verifier wins	Falsifier moves first, Verifier wins

Table 4.1: Terminologies for players.

made classical and Demon constructive, which would result in the same logic CdGL, but with the names of the players reversed.

We also note the logic dDG$\mathcal{L}$ (which, as does dGL, targets some version of games) for its unique player naming convention. The technical differences between dGL and dDG$\mathcal{L}$ are discussed in Section 4.3; the present section is solely interested in their terminology choices. The logic dDG$\mathcal{L}$ calls its players Verifier and Falsifier because the first seeks to make a postcondition true while the second seeks to make the postcondition false. Both players in dDG$\mathcal{L}$ are classical, following the convention set by GL. The choice of the names Verifier and Falsifier in dDG$\mathcal{L}$ is notable because the names directly describe each player's goals rather than relying on the reader's intuitions about the competing goals of an Angel and Demon.

Not only do the logics vary in the names and constructivity of their players, they also differ in how their modalities are read or explained in terms of those players. While CGL and CdGL use the same names Angel and Demon that are used in GL and dGL, the modalities are explained differently. In GL and dGL, Angel is always in control at the beginning of the game (first-to-move), so that Demon (second-to-move) only controls choices which occur under a[2] duality symbol $\cdot^d$, where $\cdot^d$ is a game connective which switches turns as explained in Section 4.4. Because Angel is always first-to-move in GL and dGL, the modalities $\langle\alpha\rangle\phi$ and $[\alpha]\phi$ are respectively read to mean that Angel or Demon respectively has a strategy to win game α where their goal is to make ϕ true at the end of gameplay.

In CGL and CdGL however, Angel is not always the player first-to-move, rather Angel is always the constructively-controlled player, who might move first or second. Thus, CGL and CdGL read the modalities differently from GL and dGL: modal formula $\langle\alpha\rangle\phi$ means that Angel has a strategy to win α with goal ϕ when she *moves first*, while modal formula $[\alpha]\phi$ means the same player Angel has a strategy to win α with goal ϕ when she *moves second*, i.e., if Demon controls top-level choices and Angel controls decisions that appear under a duality. Our difference in reading owes to the fact that one player is classical but the other is constructive. As the players take turns throughout gameplay, we must

[2]The syntax of games allows dualities to be nested. The player first-to-move controls choices that appear under an even number of dualities, including zero, while the player second-to-move controls those which appear under an odd number.

remember whether the current player is classical vs. constructive, i.e., whether that player is Angel vs. Demon. In the classical setting of GL and dGL, the players are both classical, which enables those logics to uniformly call the first player Angel.

The logic dDG$\mathcal{L}$ (Quesel, 2013, Ch. 4) is unique in its use of the names Verifier and Falsifier rather than Angel and Demon. For readers familiar with dDG$\mathcal{L}$, it serves as a valuable point of comparison because the dDG$\mathcal{L}$ reading of the modalities is closer to ours than GL or dGL: in the dDG$\mathcal{L}$ reading, all modalities ask whether Verifier can make a goal condition true, but $\langle\alpha\rangle\phi$ means that Verifier starts in control and $[\alpha]\phi$ means that Falsifier starts in control.

If we had chosen the names Verifier and Falsifier, the readings used in dDG$\mathcal{L}$ would work for us as well. Rather, we chose not to use the names Verifier and Falsifier for two reasons. Constructive logic has no notion of a formula ϕ being false, nor does the absence of a constructive proof of ϕ mean the negation $\neg\phi$ is provable, thus the name "Falsifier" could be misleading in the constructive context. Secondly, this thesis follows the players from modeling and proof all the way to extraction and execution of strategies. The names Verifier and Falsifier are primarily meaningful when discussing proofs, while the names Angel and Demon have a consistent meaning from proof to code: Angel is the player controlled by code while Demon is the classical adversary.

4.3 Related Work

This work is at the intersection of game logic and constructive modal logics. Individually, they each have a rich literature, but little work has been done at their intersection. Compared to the following works, we are the first for GL and the first with a proofs-as-programs interpretation for a full first-order program logic.

Games in Logic. The propositional GL (Parikh, 1983) developed by Parikh was followed by coalitional GL (Pauly, 2002). A first-order generalization of GL is the basis of differential game logic dGL (Platzer, 2015a, 2017b) for hybrid games. Separate to dGL, the logic dDG$\mathcal{L}$ (Quesel, 2013, Ch. 4) was developed which also addresses hybrid games but uses a weaker advance-notice semantics for loops. GLs are unique in their clear delegation of strategy to the *proof* language rather than the *model* language, crucially allowing succinct game specifications with sophisticated winning strategies. Succinct specifications are important: specifications are *trusted* because proving the *wrong theorem* would not ensure correctness. Relatives without this separation include Strategy Logic (Chatterjee, Henzinger, & Piterman, 2007), Alternating-Time Temporal Logic (ATL) (Alur, Henzinger, & Kupferman, 2002), CATL (van der Hoek, Jamroga, & Wooldridge, n.d.), Ghosh's SDGL (Ghosh, 2008), Ramanujam's structured strategies (Ramanujam & Simon, 2008), dynamic-epistemic logics (van Benthem, 2015; van Benthem, Pacuit, & Roy, 2011; Van Benthem, 2001), evidence logics (van Benthem & Pacuit, 2011). Strategies are not delegated to the proof language in process calculi (which do not support games) such as Hybrid CSP (Zhou et al., 1995; Liu et al., 2010), HyPA (Cuijpers & Reniers, 2005), and Hybrid χ (Schiffelers et al., 2003) either, but process calculi do share some modeling capabilities with GL: specifically, two-way

communication channels in process algebra can be compared to program variables whose values can be modified by both players.

Angelic Hoare logic (Mamouras, 2016) features several proof systems, some of which provide a GL-like separation between specification and proof, some of which do not. The synthesis of strategies is only considered for the version of their calculus which combines specification and proof by annotating programs with formulas. Moreover, in all versions of Angelic Hoare logic, only a class of games called *safety games* are considered. Safety games are a strict subset of those supported in GLs; notably, liveness and reach-avoid properties are not supported.

Constructive Modal Logics. CGL introduces constructive semantics for games, not to be confused with game semantics (Abramsky, Jagadeesan, & Malacaria, 2000), which are used to give programs semantics *in terms of* games. We draw on work in semantics for constructive modal logics, of which two main approaches are intuitionistic Kripke semantics and realizability semantics.

An overview of Intuitionistic Kripke semantics is given by Wijesekera (Wijesekera, 1990). Intuitionistic Kripke semantics are parameterized over worlds, but in contrast to classical Kripke semantics, possible worlds represent what is currently *known* of the state. Worlds are preordered by $w_1 \geq w_2$ when w_1 contains at least the knowledge in w_2. Kripke semantics were used in Constructive Concurrent DL (Wijesekera & Nerode, 2005), where both the world and knowledge of it change during execution. A key advantage of realizability semantics (van Oosten, 2002; Lipton, 1992) is their explicit interpretation of constructivity as computability by giving a *realizer*, a program which witnesses a fact. Our semantics combine elements of both realizability semantics and traditional Kripke semantics: strategies are represented by realizers, but the game state is a Kripke world. Constructive set theory (Aczel & Gambino, 2006) helps understand which set operations are allowed in constructive semantics.

Modal semantics have also exploited mathematical structures such as: *i)* neighborhood models (van Benthem, Bezhanishvili, & Enqvist, 2017), topological models for spatial logics (van Benthem & Bezhanishvili, 2007), and for temporal logics of dynamical systems (Fernández-Duque, 2018), *ii)* categorical (Alechina, Mendler, de Paiva, & Ritter, 2001), sheaf (Hilken & Rydeheard, 1999), and pre-sheaf (Ghilardi, 1989) models, and *iii)* coalgebraic semantics for classical Propositional Dynamic Logic (PDL) (Doberkat, 2011). While games are known to exhibit algebraic structure (Goranko, 2003), such laws are not essential to this chapter. Our semantics are also notable for the seamless interaction between a constructive Angel and a classical Demon.

CGL is first-order, so we must address the constructivity of operations that inspect game state. We use rational numbers here, so equality is decidable, while Chapter 5 develops the CdGL variant of CGL with constructive reals (Bishop, 1967; Bridges & Vita, 2007).

Intuitionistic modalities also appear in dynamic-epistemic logic (DEL) (Frittella, Greco, Kurz, Palmigiano, & Sikimic, 2016), but DEL focuses on proof-theoretic semantics while we use realizability semantics to focus on computation. Intuitionistic Kripke semantics also appear in multimodal System K with iteration (Celani, 2001), a weak PDL fragment.

Constructivity and Dynamic Logic. While constructive program logics have been studied significantly less than other program logics or other constructive logics, there have been several attempts to develop a constructive program logic. With CGL, we bring past efforts to fruition. Prior work on PDL (Degen & Werner, 2006) sought an Existence Property (also called Existential Property) for Propositional Dynamic Logic (PDL), but they questioned the practicality of their own implication introduction rule, whose side condition is non-syntactic. Degen cited a first-order Existence Property as an open problem beyond the methods of their day (Degen & Werner, 2006), and a (weaker semantic counterpart to) an Existence Property is one of our results. To our knowledge, only one approach (Kamide, 2010) considers Curry-Howard or functional proof terms for a program logic. While their work is a notable precursor to ours, their logic is a weak fragment of PDL without tests, monotonicity, or unbounded iteration. In contrast, we support not only PDL but the much more powerful first-order GL. Lastly, we are preceded by Constructive Concurrent Dynamic Logic, (Wijesekera & Nerode, 2005) which gives an intuitionistic Kripke semantics for Concurrent Dynamic Logic (Peleg, 1987), a proper fragment of CGL. Their work focuses on an epistemic interpretation of constructivity, algebraic laws, and a tableaux calculus. We differ in our use of realizability semantics and natural deduction, which were essential to developing a Curry-Howard interpretation for CGL. In summary, we are justified in claiming to have the first Curry-Howard interpretation with proof terms and a (weak) Existence Property for an *expressive* program logic, the first constructive game logic, and the only one with first-order proof terms.

While constructive natural deduction calculi map most directly to functional programs, proof terms can be generated for any proof calculus, including a well-known interpretation of classical logic as continuation-passing style (Griffin, 1990). Proof terms have been developed (Fulton & Platzer, 2016) for a Hilbert calculus for dL, but that work focuses on a provably correct interchange format for classical dL proofs, not constructive logics.

Compared to previous constructive dynamic logics, CGL also supports first-order reasoning, assignment, iteration, and duality. The combination of first-order reasoning with game reasoning is synergistic: for example, repetition games are known to be more expressive than repetition systems (Platzer, 2015a). Our proof calculus (Section 4.7) also includes a new natural-deduction formulation of monotonicity. Additionally, first-order games are rife with changing state, which is an important consideration in the design of any (sound) dynamic logic. Our calculus in particular is designed to provide persistent contexts through the (automatic) use of ghost variables: when some variable x is modified, formulas of the context which mention x are soundly renamed to use a fresh variable y standing for the *old* value of x. Our use of ghost variables to remember old values of variables foreshadows the design of the Kaisar proof language in Chapter 7, where the use of labels for references to past states is reducible to the use of ghost variables. In the appendix (Appendix A.3), we use our calculus to prove the example formulas.

4.4 Syntax

We define the language of CGL, consisting of terms, games, and formulas. The syntax of CGL overlaps greatly with the syntax of dL, but we present the syntax in full for the sake of clarity. Furthermore, the CGL calculus, in contrast to the dL calculus of Chapter 2, is not based on uniform substitution. This choice was made to ensure that we do not distract from developing a constructive GL in the effort to constructively interpret uniform substitution symbols.

The simplest terms are *program variables* $x, y \in \mathcal{V}$ where $\mathcal{V}$ is the (at most countable) set of variable identifiers. Globally-scoped mutable program variables contain the state of the game, also called the *position* in game-theoretic terminology. All variables and terms are rational-valued ($\mathbb{Q}$); we also write $\mathbb{B}$ for the set of Boolean values $\{0, 1\}$ meaning false and true respectively. For the sake of precision, we inductively define a (closed) language of terms. However, the term language of discrete CGL should be easily extensible with any term operators that support basic properties such as coincidence (Lemma 4.11) and substitution (Lemma 4.13). In CdGL (Chapter 5, Chapter 6), we will switch to an open-ended term language in order to better exploit the term constructs available in type theory.

Definition 4.1 (Terms)**.** A *rational term* f, g is inductively defined by the syntax

$$f, g \ ::= \ q \mid x \mid f + g \mid f \cdot g \mid f \mathsf{\ div\ } g \mid f \mathsf{\ mod\ } g$$

where $q \in \mathbb{Q}$ is a rational literal, x a program variable, $f + g$ a sum, $f \cdot g$ a product.

Division-with-remainder and modulus require special attention because they are typically associated with integers, but all terms in CGL have rational type. In CGL, these operations are intended for use with values that happen to be integers, but we generalize the standard notion of division-with-remainder so that our definition is well-defined on all rational arguments but agrees with the standard definition on integer arguments. We define $f \mathsf{\ div\ } g$ as the integer k which maximizes $k \cdot g$ while satisfying the constraint $k \cdot g \leq f$, thus $f \mathsf{\ div\ } g$ is *not* the standard notion of rational division. For example, $2 \mathsf{\ div\ } \frac{5}{7} = 2$ because $2 \cdot \frac{5}{7} \leq 2 < 3 \cdot \frac{5}{7}$. Our definition also works for negative divisors, which often require special attention in division-with-remainder. For example, $2 \mathsf{\ div\ } \frac{-5}{7} = -2$ because $-2 \cdot \frac{-5}{7} \leq 2 < -3 \cdot \frac{-5}{7}$. We define the remainder uniquely as the number such that $f = f \mathsf{\ mod\ } g + g \cdot (f \mathsf{\ div\ } g)$. For example, $2 \mathsf{\ mod\ } \frac{5}{7} = \frac{4}{7}$ because $2 = \frac{4}{7} + \frac{5}{7} \cdot 2$. Likewise, $2 \mathsf{\ mod\ } \frac{-5}{7} = \frac{4}{7}$ because $2 = \frac{4}{7} + \frac{-5}{7} \cdot -2$. Our division-with-remainder operator is not a focus of this work, but is presented because it is an important component of classic example games such as Nim, and because we find it simpler to use a single rational type rather than introduce multiple arithmetic types. Divisors g are assumed to be nonzero. Negation $-f$ is defined as $-1 \cdot f$ and subtraction $f - g$ is defined as $f + (-g)$.

While only rational-valued terms are used in CGL models and proofs, Boolean-valued terms are also used to define the realizer semantics (Section 4.6.1). A Boolean term is simply a propositional combination of comparisons of rational terms.

A game in CGL is played between a constructive player named Angel and a classical player named Demon. In *classical* GL, Angel always refers to the current player and Demon always refers to the opposite player. In *constructive* GL and thus in this thesis, the

meaning of the names Angel and Demon is subtly different: Angel always refers to the player we control, whose strategy is determined by a proof, while Demon always refers to the player we do not control. Demon is not required to resolve strategy choices constructively[3]. Because Demon is an adversary, Demon plays an adversarial strategy where Angel's worst-case scenario always comes true. In CPSs, for example (Chapter 5), Demon will represent an environment consisting of physics and any adversarial agents. While the classical terminology is well-established, we find our terminology helpful when discussing the relationship between proofs and code: classical GL has Angel and Demon switch identities when they switch turns, but we find it simpler to say that a proof provides a strategy for a single player Angel who is not always in control, rather than a player who is sometimes called Angel and sometimes called Demon. However, the classical meaning of Angel and Demon provides a strong intuitive reading that Angelic choices are resolved existentially while Demonic choices are resolved universally. We will continue to use the adjectives Angel*ic* and Demon*ic* to refer to existential and universal choices, respectively. The CGL definitions of Angel and Demon are also well-suited to the fact that the players in CGL are asymmetric: Angel's strategy must be computable while Demon's strategy is not assumed to be computable because Demon will often not be a computer, especially in the CPS context. The author is unaware of any practical example where Angel's strategy would benefit from the assumption that Demon's strategy is computable, rather the difference between computable and noncomputable Demon strategies is seen as an aesthetic distinction: because Demon is not meant to be a computer, it is preferable that the semantics of a Demonic strategy do not assume computability.

When we speak of how "a" player plays a given game, we mean the player who is next-to-move, and say "opponent" for the other player. When discussing concrete example games, we say "the first player" for the player that makes the first move, and "the second player" for the other.

Definition 4.2 (Games)**.** The set of *games* α, β is defined recursively as such:

$$\alpha, \beta \ ::= \ ?\phi \mid x := f \mid x := * \mid \alpha \cup \beta \mid \alpha; \beta \mid \alpha^* \mid \alpha^d$$

Compared to dL, the games of CGL add a duality operator α^d and leave out differential equations, which are rather reintroduced by CdGL (Chapter 5). Discrete systems (dual-free games) have the same meaning in a GL as they do in the corresponding DL, but we describe the shared operators of DL and GL using the more general game terminology. The *test game* $?\phi$ is won by exhibiting a *constructive* proof that formula ϕ currently holds. If the player does not exhibit a proof, the opponent wins by default: informally, the player who lost "broke the rules". In deterministic assignment games $x := f$, neither player makes a choice, but the new value of program variable x is computed by evaluating a term f. The nondeterministic assignment game $x := *$ is played by constructively picking a value for $x : \mathbb{Q}$. The choice game $\alpha \cup \beta$ is played by constructively choosing whether to play game α or game β, informing the opponent of the choice, then playing the chosen branch. In the sequential composition game $\alpha; \beta$, game α is played first, then β is played from the resulting

[3]In contrast with constructs which resolve strategy choices, tests $?\phi$ are always constructive in the sense that they test constructive truth of ϕ.

state. The repetition game α^* is played by repeatedly deciding whether to terminate the loop or to play for (at least) one more round. Loop durations are finite but not bounded; alternatively, we lose the game by default if we choose to repeat forever. Notably, the exact number of repetitions can depend computably on the opponent's moves, so a player need not know, let alone announce, the exact number of iterations in advance. In the dual game α^d, the players switch between the "player" role and "opponent" role: if Angel was next-to-move in α^d, Demon is next-to-move in α, and vice versa. We parenthesize games with braces $\{\alpha\}$ when necessary. Sequential and nondeterministic composition both associate to the right, i.e., $\alpha \cup \beta \cup \gamma \equiv \{\alpha \cup \{\beta \cup \gamma\}\}$. This does not affect their semantics as both operators are associative, but aids in reading proof terms.

Definition 4.3 (CGL Formulas)**.** The core grammar of CGL *formulas* ϕ (also ψ, ρ) is given recursively:

$$\phi \ ::= \ \langle\alpha\rangle\phi \mid [\alpha]\phi \mid f \sim g$$

where $\sim \in \{\leq, <, =, \neq, >, \geq\}$ is a binary comparison predicate on rationals.

The defining constructs in CGL (and GL) are the modalities $\langle\alpha\rangle\phi$ and $[\alpha]\phi$, which are interdefinable in accordance with the axiom $\langle\alpha^d\rangle\phi \leftrightarrow [\alpha]\phi$. The CGL modalities $\langle\alpha\rangle\phi$ and $[\alpha]\phi$ both indicate that the constructive player Angel has an effective strategy to achieve postcondition ϕ in game α. The difference is that Angel is the player next-to-move in $\langle\alpha\rangle\phi$ while Demon is the player next-to-move in $[\alpha]\phi$. We do not develop modalities for the existence of the Demon player's classical strategies because classical strategies are already studied in classical dGL (Platzer, 2015a) and by definition their existence does not imply the existence of executable code implementing the strategy. That being said, studying the various combinations of classical and constructive players in future work could be useful as a way to crystallize the similarities and differences between GL and CGL (corr. dGL and CdGL): one might wish to know which valid GL formulas are not valid in CGL and one might also wish to know how the set of valid CGL formulas would be impacted by a constructivity requirement for Demon. We assume the presence of interpreted comparison predicates $\sim \in \{\leq, <, =, \neq, >, \geq\}$. Because CGL operates over rational numbers, the comparison predicates are all decidable.

The standard connectives of first-order constructive logic can be derived from games and comparisons. Verum (*true*) is defined $1 > 0$ and falsum (*false*) is $0 > 1$. Conjunction $\phi \wedge \psi$ is defined $\langle ?\phi\rangle\psi$, disjunction $\phi \vee \psi$ is defined $\langle ?\phi \cup ?\psi\rangle \mathit{true}$, implication $\phi \rightarrow \psi$ is defined $[?\phi]\psi$, universal quantification $\forall x\, \phi$ is defined $[x := *]\phi$, and existential quantification $\exists x\, \phi$ is defined $\langle x := *\rangle\phi$. As usual in logic, equivalence $\phi \leftrightarrow \psi$ can also be defined $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$. As usual in constructive logics, negation $\neg\phi$ is defined $\phi \rightarrow \mathit{false}$ and inequality is defined by $f \neq g \equiv \neg(f = g)$. We will use the derived constructs freely but present semantics and proof rules only for the core constructs to minimize duplication. Indeed, it will aid in understanding of the proof term language to keep the definitions above in mind, because the proof terms for many first-order programs follow those from first-order constructive logic.

For convenience, we also write derived operators where the opponent is given control of a single choice before returning control to the player. The *dual choice* $\alpha \cap \beta$, defined $\{\alpha^d \cup \beta^d\}^d$, says the opponent chooses which branch to take, then the player regains control

of the subgames. We write $\phi\frac{y}{x}$ (likewise for α and f) for the *renaming* of program variable x for y and vice versa in formula ϕ, and write ϕ_x^f for the *substitution* of term f for program variable x in ϕ, if the substitution is admissible (Def. 4.14 in Section 4.8). The full theory (Appendix A.4) extends single term substitutions $\cdot_x^f$ to a (program variable) substitution σ that replaces any finite set of variables x with terms $\sigma(x)$. While the theory is developed for program variable substitutions σ that can replace multiple variables, we primarily use the singleton substitution $\cdot_x^f$ in this chapter.

4.5 Example Games

We provide example games to demonstrate the meaning and usage of the CGL constructs. We start with contrived toy expressions and build to two classic introductory examples: Nim and cake-cutting. While introductory, these examples are intended to ease the learning curve of CGL before addressing full hybrid games in Chapter 5 in order to verify CPSs.

Nondeterministic Programs. Every (possibly nondeterministic) program is also a one-player game. For example, the program $n := 0; \{n := n + 1\}^*$ can nondeterministically set n to any natural number because the player has a choice whether to continue after every repetition of the loop, but is not allowed to continue forever. Conversely, games are like programs where the environment (Demon) is adversarial, and the program (Angel) strategically resolves nondeterminism to overcome the environment.

Demonic Counter. Angel's choices often must *react* to Demon's choices. Suppose Angel moves first in the game $c := 10; \{c := c - 1 \cap c := c - 2\}^*; ?(0 \leq c \leq 2)$ where Demon repeatedly decreases c by 1 or 2, and Angel chooses when to stop. Angel only wins because she can pass the test $?(0 \leq c \leq 2)$, which she can do by simply repeating the loop until $0 \leq c \leq 2$ holds. Because the initial value of c is $10 > 2$ and Demon must subtract either 1 or 2 at each step, there must exist a loop iteration in which $0 \leq c \leq 2$ holds. It is crucial that Angel need not choose and announce the loop duration in advance, because fixing or announcing the duration would allow Demon to falsify the test $?(0 \leq c \leq 2)$. For example, if Angel announces a duration of 5 turns, Demon can repeatedly choose the branch $c := c - 1$ and achieve a final value of 5, or if Angel announces a duration of 10 turns, Demon can achieve a final value of -10 by choosing the $c := c - 2$ branch each time.

Coin Toss. Games are perfect-information and do not have randomness in a probabilistic sense, only (possibilistic) nondeterminism. This standard limitation is shown by attempting to express a coin-guessing game:

$$\{\mathsf{coin} := 0 \cap \mathsf{coin} := 1\}; \{\mathsf{guess} := 0 \cup \mathsf{guess} := 1\}; ?\mathsf{guess} = \mathsf{coin}$$

The Demon player sets the value of a tossed coin, but does so adversarially, not randomly, since strategies in CGL are not only computable strategies, but also *pure* strategies. The Angel player has perfect knowledge of coin and can set guess equivalently, thus easily

passing the test $\mathsf{guess} = \mathsf{coin}$, unlike a real blind coin toss. Partial-information games are valuable future work and might be modeled by limiting the variables visible in a strategy.

Even though partial information is not modeled explicitly in CGL, careful ordering of game statements allows us to approximate private decision-making with adversarial decision-making. Consider the following rearrangement of the coin toss game:

$$\{\mathsf{guess} := 0 \cup \mathsf{guess} := 1\}; \{\mathsf{coin} := 0 \cap \mathsf{coin} := 1\}; ?\mathsf{guess} = \mathsf{coin}$$

In this game, Angel guesses first and announces the guess to Demon, after which Demon adversarially chooses the coin value. This adversarial model is conservative in the sense that Demon can always choose a value of coin that disagrees with guess because he can simply view the value of guess. The conservative model does still accurately capture the fact that Angel does not have a strategy to win blind coin toss because Angel does not know the value of the coin when making her guess.

In a real blind coin toss, Demon's best strategy is to play randomly and win half of the time, rather than winning every time through perfect information. However, victory in CGL is possibilistic rather than probabilistic: if Angel's best random strategy would win less than 100% of the time, she does not have a winning strategy according to CGL, where strategies must be pure.

The possibilistic nature of victory in CGL serves to justify the use of adversarial choice as an imperfect approximation of random choice. In a random strategy, Demon's coin flip would not *always* disagree with Angel's guess, but any impure Demon strategy would *sometimes* agree with the adversarial choice, i.e., the opposite of Angel's guess. Because the random choice would sometimes agree with the adversarial choice, an adversarial model faithfully models the fact that Angel would not have a *pure* winning strategy which beats a random Demon 100% of the time.

We discussed the impact of reordering game statements here because the same close attention to statement ordering will prove helpful in CdGL when modeling and proving systems that are simultaneously safe and live (Section 5.2.2).

Nim. Nim is the standard introductory example of a discrete, 2-player, zero-sum, perfect-information game. We consider misère play (last player loses) for a version of Nim that is also known as the *subtraction game*. The name Nim stands for the loop body from the following CGL model of the game Nim:

$$\textsc{Nim}^* = \Big\{\big\{\{c := c-1 \cup c := c-2 \cup c := c-3\}; ?c > 0\big\}; \big\{\{c := c-1 \cup c := c-2 \cup c := c-3\}; ?c > 0\big\}^d\Big\}^*$$

The game state consists of a single counter c containing a natural number, which each player chooses ($\cup$) to reduce by 1, 2, or 3 ($c := c - k$). So long as the counter is positive, the game can repeat with a duration controlled by whichever player moved first. If the loop repeats long enough, some player will empty the counter, at which point that player is declared the loser ($?c > 0$).

Proposition 4.1 (Second-player winning region)**.** *Suppose* $c \equiv 1 \pmod 4$ *and Demon moves first. The opponent Angel has a strategy to ensure* $c \equiv 1 \pmod 4$ *as an invariant. That is, the following* CGL *formula is provable in the* CGL *proof calculus (Section 4.7):*

$$c > 0 \to c \mathsf{\ mod\ } 4 = 1 \to [\textsc{Nim}^*]\, c \mathsf{\ mod\ } 4 = 1$$

This implies that Angel wins the game because Demon violates the rules once $c = 1$ and no move is valid. We now state the winning region in the case that Angel moves first.

Proposition 4.2 (First-player winning region)**.** *Suppose* $c \in \{0, 2, 3\} \pmod 4$ *initially and Angel moves first. Then Angel can achieve* $c \in \{2, 3, 4\}$*:*

$$c > 0 \to c \mathsf{\ mod\ } 4 \in \{0, 2, 3\} \to \langle \textsc{Nim}^* \rangle\, c \in \{2, 3, 4\}$$

At that point, Angel wins in the next turn by choosing whichever branch results in $c = 1$ thus forcing Demon to set $c = 0$ and fail the test $?c > 0$.

Cake-cutting. Another classic 2-player game, from the study of equitable division, is the cake-cutting problem (Pauly & Parikh, 2003): The first player cuts the cake in two, then the opponent gets first choice of a piece. This is an optimal protocol for splitting the cake in the sense that the first player is incentivized to split the cake evenly, else the second player could take the larger piece. Cake-cutting is also a simple use of fractional numbers. The constant CC defines the cake-cutting game. Here x is the relative size (from 0 to 1) of the first piece, y is the size of the second piece, a is the size of the piece chosen by the first player, and d is the size of the piece chosen by the second player.

$$\begin{aligned} \text{CC} \equiv\ & x := *; ?0 \le x \le 1; y := 1 - x; \\ & \{a := x; d := y \cap a := y; d := x\} \end{aligned}$$

The game is played only once. The first player picks the division of the cake, which must be a fraction $0 \le x \le 1$. The second player then picks which slice goes to whom.

The first player has a tight strategy to get a 0.5 share, as stated in Proposition 4.3.

Proposition 4.3 (First-player winning region)**.** *The following formula is valid:*

$$\langle CC \rangle\, a \ge 0.5$$

The second player also has a computable strategy to get at least a 0.5 share (Proposition 4.4). Division is fair since each player has a strategy to get a fair 0.5 share.

Proposition 4.4 (Second-player winning region)**.** *The following formula is valid:*

$$[CC]\, d \ge 0.5$$

Computability and Numeric Types. Perfect fair division is only achieved for $a, d \in \mathbb{Q}$ because rational equality is decidable. Trichotomy ($a < 0.5 \lor a = 0.5 \lor a > 0.5$) is a tautology, so the second player's strategy can inspect the first player's choice of a. Notably,

CdGL (Chapter 5) uses constructive reals, for which exact equality is not decidable and trichotomy is not an axiom. Chapter 5 employs approximate comparison techniques as is typical for constructive reals (Bishop, 1967; Bridges & Vita, 2007; Weihrauch, 2000). The examples in this section have been proven (Appendix A.3) using the proof calculus that will be defined in Section 4.7.

4.6 Semantics

We now develop the semantics of CGL. In contrast to classical GL, whose semantics are well-understood (Parikh, 1983), one semantic challenge for CGL is capturing the competition between a *constructive* Angel and *classical* Demon. Specifically, Angel must resolve discrete choices, loops, and nondeterministic assignments constructively, while Demon need not. We base our approach on realizability semantics (van Oosten, 2002; Lipton, 1992). The basic idea of realizability semantics is that semantic truth of a formula is witnessed by a program, called a realizer, which performs any computations that are necessary to show its truth. For example, a realizer for a disjunction $\phi \vee \psi$ would decide whether the disjunction should be proved by showing the ϕ branch vs. the ψ branch. We chose to use realizability semantics because this chapter seeks to give a generic treatment of CGL and realizability semantics allow us to express which computational constructs are fundamental to all CGLs while leaving the language open to future extension with domain-specific constructs. In that sense, realizability semantics are well-suited to a generic presentation. The realizability approach also makes the relationship between constructive proofs and programs particularly clear, which is important because generating programs from CGL proofs is one of our motivations.

Unlike previous applications of realizability, games feature two agents, and one could imagine a semantics with two realizers, one for each of Angel and Demon. However, we choose to use only one realizer, for Angel, which captures the fact that only Angel is restricted to a computable strategy, not Demon. Moreover, a single realizer crucially makes it clear that Angel cannot inspect the internal structure of Demon's strategy, only the game state and values explicitly passed to Angel by Demon. The single realizer also simplifies notations and proofs. Because Angel is computable but Demon is classical, our semantics has the flavor of both realizability semantics and Kripke semantics.

While the CdGL proof calculus (Chapter 5) and refinement calculus (Chapter 6) will ultimately adopt a type-theoretic semantics for the sake of technical elegance, the role of realizers in CGL also foreshadows the refinement relationships that hold (Chapter 6) between hybrid systems and hybrid games: A game proof specifies a single strategy, whose possible behaviors are a subset (refinement) of the behaviors allowed by the game. Once one player has committed to a strategy, all remaining decisions belong to their opponent, and a system (one-player game) suffices (Chapter 6) to describe the remaining behaviors. Once a player has committed to a realizer, CGL gameplay proceeds much like execution of a system, as reflected in our semantics. The superficial similarity in appearance between CGL and DL semantics does not alter their deep difference: game theorems ask whether there *exist* strategies (Def. 4.8) of a game satisfying some property. Quantification over strategy

existence (Def. 4.8) entails significant differences in the proof rules available in DLs vs. GLs and profound differences (Platzer, 2015a) in their expressive power. Constructivity specifically entails differences in expressive power as well: as usual, the law of the excluded middle cannot be applied to undecidable properties. For example, the winnability of a given game is presumed to be undecidable by analogy to undecidability of first-order (classical) dynamic logic (Harel et al., 2000, Thm. 13.1). In the case of CGL for discrete games however, the differences in expressive power between classical and constructive games are not our focus because practical discrete game proofs rarely branch on undecidable properties. Rather, we are interested in the *semantic* difference between classical and constructive games: in CGL semantics, realizers give explicit witnesses for computable strategies, which will serve our goal of providing a computational interpretation for CGL proofs.

The semantic functions employ *game states* $\omega \in \mathcal{S}$ where we write $\mathcal{S}$ for the set of all states. Each state $\omega \in \mathcal{S}$ maps each $x \in \mathcal{V}$ to a value $\omega(x) \in \mathbb{Q}$. We use update notation $\omega[x \mapsto v]$ to mean the state that agrees with state ω except that x is assigned value v where $v \in \mathbb{Q}$.

Definition 4.4 (Arithmetic and Boolean term semantics)**.** The interpretation $[\![f]\!]\omega$ of rational term f in state ω inductively applies each term operator $(+, \cdot, \mathsf{div}, \mathsf{mod})$ to the values of subterms, where rational literals are their own values and the value of variable x is $\omega(x)$. Division-with-remainder is defined as it is described in Section 4.4. The interpretation $[\![f]\!]\omega$ of Boolean term f in state ω employs the rational term semantics to evaluate terms, then applies the standard semantics of propositional connectives to interpret the Boolean term.

4.6.1 Realizers

To define the semantics of games, we first define realizers, the programs which implement strategies. The language of realizers is a higher-order lambda calculus where variables can range over numbers or over realizers which realize a given proposition ϕ. Realizers contain arithmetic and Boolean terms which are used to resolve individual moves during gameplay. Those terms are *open* in the sense that they can mention program variables, whose values will be taken from the state at which a move is chosen. When one realizer accepts another realizer as its argument, any open terms in the argument realizer are evaluated lazily, i.e., it is permissible to evaluate the argument realizer at multiple, arbitrary states. Gameplay proceeds in continuation-passing style: invoking a realizer returns another realizer which performs any remaining moves.

We describe the typing constraints for realizers informally, and say realizer b is a $\langle\alpha\rangle\phi$-realizer (written $\mathsf{b} \in \langle\alpha\rangle\phi\,\mathcal{R}\mathbf{z}$) if it provides strategic decisions exactly when $\langle\alpha\rangle\phi$ demands them. We colloquially say b is well-typed if b is a ϕ-realizer for some ϕ where the choice of ϕ is clear in context. The typing constraints for a realizer *do not* address whether a given formula ϕ is true, but address whether the shape of the realizer is compatible with the shapes expected by the formula semantics so that the truth or falsehood of ϕ as a postcondition of α can be assessed. A major limitation of our realizability semantics is that precise formal descriptions of the typing constraints are awkward. A major motivation for our eventual adoption of type-theoretic semantics in Chapter 5 is the fact that the typing rules of existing type theories can more elegantly and implicitly capture the constraints

which a realizer semantics must express explicitly. While one could argue that the use of type-theoretic semantics in Chapter 5 will require us to embrace some complexity that comes with the underlying type theory, the use of type theory will ultimately simplify the semantics of Chapter 5 because it is simpler to appeal to a system whose complexities are already well-studied than to reinvent the same complexities from scratch.

Definition 4.5 (Realizers)**.** The syntax of realizers $\mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{rz} \in \mathcal{R}\mathbf{z}$ (where $\mathcal{R}\mathbf{z}$ is the set of all realizers) is defined coinductively:

$$\begin{aligned}\mathsf{b}, \mathsf{c}, \mathsf{d} \text{ (sometimes } \mathsf{rz}) ::= x \mid f \mid \epsilon \mid (\mathsf{b}, \mathsf{c}) \mid \pi_0\mathsf{b} \mid \pi_1\mathsf{b} \mid (\Lambda x : \mathbb{Q}.\ \mathsf{b}) \\ \mid (\Lambda x : \phi\,\mathcal{R}\mathbf{z}.\ \mathsf{b}) \mid \mathsf{b}\,v \mid \mathsf{b}\,\mathsf{c} \mid \mathtt{if}\ (f)\ \mathsf{b}\ \mathtt{else}\ \mathsf{c}\end{aligned}$$

where x is a variable over realizers and f is an open term which can be evaluated in any ambient state ω. The Roman $\mathsf{b}, \mathsf{c}, \mathsf{d}$ should not be confused with the Greek β, γ, δ which range over games. Individual strategic decisions are realized by term realizers f of type $\mathbb{Q}$ or $\mathbb{B}$: rational realizers reuse the CGL term language, while Boolean terms can use comparisons of rational terms and propositional connectives. The unit realizer ϵ makes no choices and its value is understood as a unit tuple, written $()$. Units ϵ realize $f \sim g$ because *rational* comparisons, in contrast to real comparisons, are decidable. Conditional strategic decisions are realized by $\mathtt{if}\ (f)\ \mathsf{b}\ \mathtt{else}\ \mathsf{c}$ where f is a Boolean term; we execute b if f returns truth when evaluated in ambient state ω, else c.

The realizer constructs are not in one-to-one correspondence with the CGL connectives. The base realizer f resolves both Angelic choices (with a Boolean term) and Angelic assignments (with a rational term), for example. Conditional realizers $\mathtt{if}(f)\mathsf{b}\mathtt{else}\mathsf{c}$ can be used in any CGL game: assuming f is a well-formed Boolean term, then $\mathtt{if}(f)\mathsf{b}\mathtt{else}\mathsf{c}$ is a $\langle\alpha\rangle\phi$-realizer whenever both b and c are, for any α and ϕ. Realizer (f, b) is a $\langle\alpha \cup \beta\rangle\phi$-realizer if $([\![f]\!]\omega, \mathsf{b}) \in (\{0\} \times \langle\alpha\rangle\phi\,\mathcal{R}\mathbf{z}) \cup (\{1\} \times \langle\beta\rangle\phi\,\mathcal{R}\mathbf{z})$ for all ω. That is, the typing constraint for $\langle\alpha \cup \beta\rangle\phi$ models a tagged union. The first component determines which branch is taken, while the second component is a continuation which must be able to play the corresponding branch. In practice, it is a common idiom to use a conditional realizer $\mathtt{if}(f)(0, \mathsf{b})\mathtt{else}(1, \mathsf{c})$ to decide which branch α or β should be played: if f holds, then $(0, \mathsf{b})$ is used to play the α branch, else $(1, \mathsf{c})$ is used to play the β branch. No new typing constraints are needed by this idiom: if $\mathsf{b} \in \langle\alpha\rangle\phi\,\mathcal{R}\mathbf{z}$ and $\mathsf{c} \in \langle\beta\rangle\phi\,\mathcal{R}\mathbf{z}$, then $\{(0, \mathsf{b}), (1, \mathsf{c})\} \subseteq \langle\alpha \cup \beta\rangle\phi\,\mathcal{R}\mathbf{z}$ by the Angelic choice rule and $\mathtt{if}\ (f)\ (0, \mathsf{b})\ \mathtt{else}\ (1, \mathsf{c}) \in \langle\alpha \cup \beta\rangle\phi\,\mathcal{R}\mathbf{z}$ by the conditional rule.

Pair realizer (f, b) is a $\langle x := *\rangle\phi$-realizer if $[\![f]\!]\omega \in \mathbb{Q}$ for all ω and $\mathsf{b} \in \phi\,\mathcal{R}\mathbf{z}$. The first component determines a new value v of variable x while the second component demonstrates the postcondition ϕ in state $\omega[x \mapsto v]$. That is, a $\langle x := *\rangle\phi$ realizer is a pair realizer (f, b) of a scalar function f of the state and a continuation realizer b for ϕ. When a realizer b computes a pair, the left and right projections are written $\pi_0\mathsf{b}$ and $\pi_1\mathsf{b}$, respectively.

Realizer pairing, written (b, c), is also used to realize both Angelic tests $\langle ?\phi\rangle\psi$ and Demonic choices $[\alpha \cup \beta]\phi$. Angelic tests $\langle ?\phi\rangle\psi$ are realized by pairs because Angel is responsible for proving both ϕ and ψ, while Demonic choices $[\alpha \cup \beta]\phi$ are realized by pairs because Angel must prepare proofs of both $[\alpha]\phi$ and $[\beta]\phi$ because she does not know in advance which branch α or β Demon will choose to play. A pair realizer is identified with a pair of realizers: $(\mathsf{b}, \mathsf{c}) \in \mathcal{R}\mathbf{z} \times \mathcal{R}\mathbf{z}$. In the Angelic test case, realizers b and c must realize

the test and postcondition formulas ϕ and ψ, while the Demonic choice case requires b and c to realize $[\alpha]\phi$ and $[\beta]\phi$. These typing constraints are a direct reflection of the proof rules for tests and choices.

Angel's realizers for box modalities $[\alpha]\phi$ are *Demonic* in the sense that Demon is allowed to play an arbitrary classical strategy which Angel passively observes, remembering Demon's moves so that they can inform Angel's computable strategy once Angel's next turn arrives. Once Angel's turn arrives, her strategy uses computable functions over the state to resolve her own strategic choices. The first-order realizer $(\Lambda x : \mathbb{Q}.\ \mathsf{c})$ is a $[x := *]\phi$-realizer when c_x^v is a ϕ-realizer for every value $v \in \mathbb{Q}$ that Demon might choose; Demon tells Angel the desired value of x, which informs Angel's continuation c. That is, the typing constraints of first-order realizers are the typing constraints of dependent functions with scalar arguments. The higher-order realizer $(\Lambda x : \phi\, \mathcal{R}\mathbf{z}.\ \mathsf{c})$ realizes $[?\phi]\psi$ when $\mathsf{c}_x^{\mathsf{d}}$ realizes ψ for every ϕ-realizer d. That is, higher-order realizers are simply-typed anonymous functions. Demon announces the realizer for ϕ which Angel's continuation c may use. Because Demon is entitled to classical strategies, Angel only uses Demon's strategy in blackbox fashion, typically by invoking it at the state of her choice or passing it as the argument to another higher-order realizer. Two functional realizers are considered equal if they are extensionally equivalent, i.e., if they agree on all arguments[4]. Tuples are inspected with projections $\pi_0\mathsf{b}$ and $\pi_1\mathsf{b}$. An anonymous function is inspected by applying arguments $\mathsf{b}\,v$ for first-order lambdas and $\mathsf{b}\,\mathsf{c}$ for higher-order. Realizers for sequential compositions $\langle\alpha;\beta\rangle\phi$ (likewise $[\alpha;\beta]\phi$) are $\langle\alpha\rangle\langle\beta\rangle\phi$-realizers: first α is played, and in every case the continuation plays β then shows ϕ.

In principle, realizers for repetitions α^* do not need new realizer language constructs, as they can be implemented as coinductive streams. However, direct definitions of repetition realizers as streams are awkward, so we will introduce derived realizer constructs $\mathsf{ind}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}}$ and $\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})$ which closely follow the corresponding proof rules we develop in Section 4.7, but which can be implemented as coinductive streams (Appendix A.4). The implementations (Def. A.3) are in the appendix because their technically-involved construction is best understood after gaining familiarity with realizer semantics and because the construction uses a low-level implementation (Lemma A.2) of monotonicity (rule M), which is best understood after reading the proof calculus (Section 4.7).

For the sake of generality, we describe the typing constraints for arbitrary repetition realizers, not only those which employ the derived constructs. The typing constraints for repetition realizers work by unrolling the iterations of the loop. The realizers of $[\alpha^*]\phi$ are exactly the realizers of the infinite nested conjunction ψ which is the greatest fixed point of the equation $\psi \leftrightarrow \phi \wedge [\alpha]\psi$, that is, the nested infinite conjunction which proves the postcondition ϕ after every finite number of iterations. The realizers of $\langle\alpha^*\rangle\phi$ are realizers

[4]The presence of an extensionality rule does not mean that we assume an extensional meta-logic in the sense of extensional type theory. In fact, the type-theoretic development of Chapter 5 will use an intensional type theory. In typical usage, the difference between an intensional and extensional theory is whether or not the definition of *type* (or proposition) equivalence contains an extensionality rule. We need not even distinguish between an intensional and extensional theory because we define no equivalence judgement on propositions. The object-level equivalence reasoning used to show equivalence of realizers in semantic proofs is a standard feature even in intensional theories.

of the infinite disjunction ψ which is the greatest fixed point of the equation $\psi \leftrightarrow \phi \vee \langle\alpha\rangle\psi$, but only those realizers which always eventually take the left branch of some disjunction. The requirement to eventually take the left branch amounts to a requirement that every Angelic loop eventually terminates, which is a standard requirement in the semantics of loops in DLs and GLs. The nested disjunctive structure is faithful to the requirement that the duration of a loop is decided interactively rather than in advance: Angel repeatedly decides whether to stop after each individual loop iteration, whereas a single top-level infinite disjunction would be incorrect by forcing Angel to choose the exact duration before playing even the first iteration.

Demonic loop realizers, in contrast, do not feature any special typing constraint to ensure termination, because it is Demon's job, not Angel's job, to ensure they terminate. If Angel happens to write a realizer for a Demon loop that is capable of playing infinitely many loop iterations, she has exceeded her requirements, because she was only required to support finite plays. Because it is Demon's responsibility, termination of Demonic loops will be implicitly enforced by the fixed-point definition of loop semantics (Def. 4.11) rather than by a realizer typing constraint.

The derived realizer construct for Demonic loops is $\mathsf{gen}(\mathsf{b}, \mathsf{c}, \mathsf{d})$, which corresponds directly to the loop invariant rule $[*]$I which we will introduce in Section 4.7. Demonic loop realizers are like coinductively-generated streams: the generator b proves the base case of an invariant, the inductive step c updates the generator (invariant proof) at each loop iteration, and the postcondition step d generates a proof of the postcondition given current generator value (i.e., current proof of the invariant).

The derived Angelic loop realizer construct $\mathsf{ind}(\mathsf{b}, \mathsf{c}, \mathsf{d})_{\mathcal{M}}$ corresponds directly to the convergence rule $\langle*\rangle$I which we will introduce in Section 4.7. Both Angelic and Demonic loop proofs work by invariant arguments, but Angelic proofs are additionally equipped with termination metrics $\mathcal{M}$ which decrease under some ordering $\succ$ after every loop iteration until terminating when the metric reaches some bound $\mathbf{0}$. In an Angelic loop realizer, b shows that some invariant condition holds initially, c shows that the invariant is maintained in each iteration while a termination metric $\mathcal{M}$ decreases, and d shows that a postcondition follows from the invariant and termination of the metric.

We require that the termination metric $\mathcal{M}$ is effectively-well-founded (Hofmann, van Oosten, & Streicher, 2006) under the ordering $\succ$. Well-foundedness is the foundation of sound loop induction rules. Our proof calculus (Section 4.7) will introduce one such rule $\langle*\rangle$I, whose soundness proof relies on the effective well-foundedness property. Strictly speaking, we only require an effective descending chain condition, which we derive from effective-well-foundedness. Our effective descending chain condition is a constructive counterpart to the standard notation of descending chain condition which can be used to show well-foundedness of classical relations. Nonetheless, we discuss our condition in the context of effective well-foundedness because effective well-foundedness is a more widely-known and widely-studied condition than ours.

In Def. 4.6, we repeat the standard definition of effective well-foundedness from the literature (Hofmann et al., 2006).

Definition 4.6 (Standard Definition of Effective Well-Foundedness)**.** Let $\succ$ be a relation

on some set X, then the metric $\succ$ is effectively well-founded iff the following induction principle is sound constructively:

- Assume $Y \subseteq X$.
- Assume $\forall x\,(\forall y\ \ (x \succ y \to y \in Y) \to x \in Y)$
- Then $X = Y$.

We define a relation $\succ$ on set S to have the effective descending chain condition if it is constructively provable that every strictly descending sequence of elements of S is finite:

Definition 4.7 (Effective Descending Chain Condition)**.** Let $\succ$ be a relation on some set X. Let $\mathsf{Seq}(X)$ be the set of sequences over X and S_i be the ith element of S for $S \in \mathsf{Seq}(X)$. Element S_i is defined for all $i \in \mathit{Dom}(S)$ where $\mathit{Dom}(S)$ is the domain of S. Set $\mathsf{FiniteSeq}$ stands for the set of finite sequences drawn from X. Then $\succ$ has the effective descending chain condition iff the following condition holds constructively:

$$(\forall S \in \mathsf{Seq}(X)\ (\forall i\, j \in \mathit{Dom}(S)\ i < j \to S_i \succ S_j)) \to S \in \mathsf{FiniteSeq}$$

That is, a relation has the effective descending chain condition if it is constructively provable that every descending sequence is finite.

The literature (Hofmann et al., 2006) shows that effective-well-foundedness implies a condition (which we call no-infinite-descent) which is classically equivalent to our effective descending chain condition: there does not exist an infinite descending chain. Rather than try to prove that the no-infinite-descent property implies the effective descending chain condition, we directly show that effective well-foundedness implies the effective descending chain condition. Note that the literature observes (Hofmann et al., 2006) that the effective-well-foundedness condition is stronger than the no-infinite-descent condition, but we have found the effective descending chain condition strong enough for our purposes of showing that inductive reasoning on loops is sound.

Lemma 4.5 (Effectively Well-Founded Implies Effectively Descending)**.** *Let $\succ$ be an effectively well-founded relation on some set X. Then $\succ$ satisfies the effective descending chain condition on X.*

Proof sketch. The proof is in Appendix A.4. The proof is by induction on the value of the first element of an arbitrary descending sequence using the induction principle on effectively-well-founded relations. Assume that all descending sequences whose initial value is lesser are finite. Every descending sequence is either at most one element (thus finite) or has a head and nonempty tail. The tail has a smaller initial element and is thus finite by the inductive hypothesis. To complete the induction, finiteness is preserved when adding back the head.

The induction shows that all non-empty descending sequences are finite. The empty sequence is clearly finite, which completes the proof. □

In this thesis, realizers are already a semantic concept, used to define the meaning of games and formulas. For that reason, an in-depth semantics *of* realizers would likely be circular or at least provide limited insight. Thus, we do not develop the semantics *of* realizers in depth, yet the CGL soundness proof will necessarily include semantic lemmas

about realizers, characterizing how their meaning changes across different states or under syntactic transformations.

Those lemmas employ an eager *forcing* semantics of realizers: $[\![\mathsf{b}]\!]\omega$ is the realizer value that results from replacing the free variables x of b with $\omega(x)$ and evaluating. Because the realizer language includes functions with scalar arguments (for example), the result of forcing could be a function. The body of the result is computed by forcing the original body after updating the state in accordance with the value of the function argument.

The use of an eager semantics contrasts with the lazy use of realizers in CGL semantics, but enables a simpler technical development. The gap between eager and lazy semantics is bridged wherever realizer lemmas are applied by locally applying lemmas when the CGL semantics do evaluate of a realizer. One prominent lemma for reasoning about realizer forcing will be the realizer case of the coincidence lemma (Lemma 4.11), which relies on a notion of free variables $\mathrm{FV}(\mathsf{b})$ of a realizer b. The free variables $\mathrm{FV}(\mathsf{b})$ of a realizer b are those which appear syntactically in a free position anywhere in realizer b, even in parts of b which will not be used until future states. Thus, the set $\mathrm{FV}(\mathsf{b})$ can be surprisingly large, but luckily the large size of $\mathrm{FV}(\mathsf{b})$ will not interfere with the key use of Lemma 4.11: reasoning about fresh ghost variables, which definitionally do not appear free.

4.6.2 Formula and Game Semantics

A state ω paired with a (continuation) realizer b that plays any following game is called a (proper) *possibility*. In addition to proper possibilities, the semantics also feature two distinguished (pseudo-)possibilities $\top, \bot$ (not to be confused with formulas *true* and *false*) indicating that Angel or Demon respectively has won the game early by forcing the other to fail a test. Thus, the set Poss of all possibilities is defined by $\mathsf{Poss} = (\mathcal{R}\mathbf{z} \times \mathcal{S}) \cup \{\top, \bot\}$. We write variable poss for an arbitrary possibility which need not be proper.

A *region* (written X, Y, Z) is a set of possibilities, e.g., $X \subseteq \mathsf{Poss}$. A region X is a proper region if $X \cap \{\top, \bot\} = \emptyset$. We write $[\![\phi]\!] \subseteq \phi\,\mathcal{R}\mathbf{z} \times \mathcal{S}$ for the *proper* region which realizes formula ϕ. We now define validity of formulas, which is an essential concept since soundness of our proof calculus (Section 4.7) shows all provable formulas are valid.

Definition 4.8 (Validity)**.** A formula ϕ is *valid* iff there exists some realizer b that uniformly realizes formula ϕ in every state, i.e., $\{\mathsf{b}\} \times \mathcal{S} \subseteq [\![\phi]\!]$. A sequent $\Gamma \vdash \phi$ is *valid* iff the formula $\bigwedge \Gamma \to \phi$ is valid, where $\bigwedge \Gamma$ is the conjunction of all assumptions in Γ.

The game semantics are region-oriented, i.e., they process possibilities in bulk, though Angel chooses a possibility from a starting region $X \subseteq \mathsf{Poss}$ and commits to it before gameplay begins. In contrast to the formula semantics, the game semantics allow initial and final regions to contain $\top$ and $\bot$. The Angelic semantics $X\langle\!\langle\alpha\rangle\!\rangle \subseteq \mathsf{Poss}$ and Demonic semantics $X[\![\alpha]\!] \subseteq \mathsf{Poss}$ compute the game's ending region as a union of possibilities which Demon can achieve through adversarial play once Angel has committed at the start to any possibility drawn from the starting region X. The semantics $X\langle\!\langle\alpha\rangle\!\rangle$ and $X[\![\alpha]\!]$ differ only in whether Angel or Demon moves first. Recall that pseudo-possibilities $\top$ and $\bot$ represent early wins by each of Angel and Demon, respectively. For the sake of readability, the definitions below describe the case where X is a proper region, but they extend to the case $\bot \in X$ (likewise $\top \in X$) using the equations $(X \cup \{\bot\})[\![\alpha]\!] = X[\![\alpha]\!] \cup \{\bot\}$ and

$(X \cup \{\bot\})\langle\langle\alpha\rangle\rangle = X\langle\langle\alpha\rangle\rangle \cup \{\bot\}$. That is, if Demon has already won by forcing an Angel violation initially, any remaining game can be skipped with an immediate Demon victory, and vice-versa.

The game semantics exploit the *Angelic* projections $Z_{\langle 0\rangle}, Z_{\langle 1\rangle}$ and *Demonic* projections $Z_{[0]}, Z_{[1]}$, which represent binary decisions made by a constructive Angel and a classical Demon, respectively. The Angelic projections, which are defined $Z_{\langle 0\rangle} = \{(\pi_1\mathsf{b}, \omega) \mid [\![\pi_0\mathsf{b}]\!]\omega = 0, (\mathsf{b}, \omega) \in Z\}$ and $Z_{\langle 1\rangle} = \{(\pi_1\mathsf{b}, \omega) \mid [\![\pi_0\mathsf{b}]\!]\omega = 1, (\mathsf{b}, \omega) \in Z\}$, first project $\pi_0\mathsf{b}$ to get a Boolean term which encodes Angel's strategy for choosing a branch. The projection inspects Angel's choice $[\![\pi_0\mathsf{b}]\!]\omega$ for the current state ω to filter only those possibilities which take the left or right branch, respectively, as determined by Angel's strategy. The projection $\pi_1\mathsf{b}$ gives the continuation realizer for any following game or formula. The Demonic projections, which are defined $Z_{[0]} \equiv \{(\pi_0\mathsf{b}, \omega) \mid (\mathsf{b}, \omega) \in Z\}$ and $Z_{[1]} \equiv \{(\pi_1\mathsf{b}, \omega) \mid (\mathsf{b}, \omega) \in Z\}$, contain the same states as Z, but project the realizer to tell Angel which branch Demon took. Every projection operator is a no-op when applied to a pseudo-possibility $\top$ or $\bot$, returning the same pseudo-possibility $\top$ or $\bot$.

Definition 4.9 (Formula semantics)**.** The formula semantics $[\![\phi]\!] \subseteq \mathcal{R}\mathbf{z} \times \mathcal{S}$ is inductively defined, simultaneous with the definition of the semantics of games. Specifically, the formula semantics are defined as:

$$
\begin{aligned}
(\epsilon, \omega) &\in [\![f \sim g]\!] \text{ iff } [\![f]\!]\omega \sim [\![g]\!]\omega \\
(\mathsf{b}, \omega) &\in [\![\langle\alpha\rangle\phi]\!] \text{ iff } \{(\mathsf{b}, \omega)\}\langle\langle\alpha\rangle\rangle \subseteq ([\![\phi]\!] \cup \{\top\}) \\
(\mathsf{b}, \omega) &\in [\![[\alpha]\phi]\!] \text{ iff } \{(\mathsf{b}, \omega)\}[\![\alpha]\!] \subseteq ([\![\phi]\!] \cup \{\top\})
\end{aligned}
$$

Comparisons $f \sim g$ defer to the term semantics, so the interesting cases are the game modalities. The similarity between the semantics of $[\alpha]\phi$ and $\langle\alpha\rangle\phi$ is almost startling when compared to dynamic logics such as dL. The dynamic logic modalities $\langle\alpha\rangle\phi$ and $[\alpha]\phi$ respectively say that some or all behaviors of game α satisfy postcondition ϕ, so it may be surprising that the CGL semantics of both $\langle\alpha\rangle\phi$ and $[\alpha]\phi$ say that *all* behaviors of the game must satisfy postcondition ϕ. The key difference between the dL semantics and the CGL semantics is that nondeterminism in the CGL semantics *always* represents the opponent Demon, while the realizer always captures the strategy of our player Angel. The semantics of $\langle\alpha\rangle\phi$ and $[\alpha]\phi$ both require that there constructively exists an Angel strategy which achieves the postcondition ϕ *for all classical Demon strategies*, with the crucial difference being whether Angel or Demon makes the first move. In both cases, early Angel wins $\top$ are a special case which arises when Demon fails a Demonic test. Early Demon wins $\bot$ are a special case which arises when Angel fails an Angelic test which, though not explicitly mentioned in the definition of $[\![\langle\alpha\rangle\phi]\!]$ or $[\![[\alpha]\phi]\!]$, play an important role. If Angel fails a test, then it will be the case (in the Angelic semantics, for example) that $\bot \in \{(\mathsf{b}, \omega)\}\langle\langle\alpha\rangle\rangle$ and thus the inclusion $\{(\mathsf{b}, \omega)\}\langle\langle\alpha\rangle\rangle \subseteq ([\![\phi]\!] \cup \{\top\})$ will not hold (because $\bot$ never belongs to $[\![\phi]\!]$ for any ϕ). This is exactly what we desire: if Angel loses a test, then the test failure will cause her to lose the game early, in which case she should not be able to prove the truth of a modal formula, because such formulas represent her victory.

Definition 4.10 (Angel game-playing forward semantics)**.** We inductively define the region

$X\langle\!\langle\alpha\rangle\!\rangle \subseteq \mathsf{Poss}$ in which α can end when Angel plays first from $X \subseteq \mathsf{Poss}$.

$$\begin{aligned}
X\langle\!\langle ?\phi\rangle\!\rangle &= \{(\pi_1\mathsf{b},\omega) \mid (\pi_0\mathsf{b},\omega) \in [\![\phi]\!] \text{ for some } (\mathsf{b},\omega) \in X\ \} \\
&\quad \cup \{\bot \mid (\pi_0\mathsf{b},\omega) \notin [\![\phi]\!] \text{ for all } (\mathsf{b},\omega) \in X\ \} \\
X\langle\!\langle x := f\rangle\!\rangle &= \{(\mathsf{b},\omega[x \mapsto [\![f]\!]\omega]) \mid (\mathsf{b},\omega) \in X\} \\
X\langle\!\langle x := *\rangle\!\rangle &= \{(\pi_1\mathsf{b},\omega[x \mapsto [\![\pi_0\mathsf{b}]\!]\omega]) \mid (\mathsf{b},\omega) \in X\} \\
X\langle\!\langle \alpha;\beta\rangle\!\rangle &= (X\langle\!\langle\alpha\rangle\!\rangle)\langle\!\langle\beta\rangle\!\rangle \\
X\langle\!\langle \alpha \cup \beta\rangle\!\rangle &= X_{\langle 0\rangle}\langle\!\langle\alpha\rangle\!\rangle \cup X_{\langle 1\rangle}\langle\!\langle\beta\rangle\!\rangle \\
X\langle\!\langle \alpha^*\rangle\!\rangle &= \bigcap\{Z_{\langle 0\rangle} \subseteq \mathsf{Poss} \mid X \cup (Z_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle) \subseteq Z\} \\
X\langle\!\langle \alpha^d\rangle\!\rangle &= X[\![\alpha]\!]
\end{aligned}$$

Definition 4.11 (Demon game-playing forward semantics)**.** We inductively define the region $X[\![\alpha]\!] \subseteq \mathsf{Poss}$ in which α can end when Demon plays first from $X \subseteq \mathsf{Poss}$:

$$\begin{aligned}
X[\![?\phi]\!] &= \{(\mathsf{b}\,\mathsf{c},\omega) \mid (\mathsf{b},\omega) \in X, (\mathsf{c},\omega) \in [\![\phi]\!], \text{ for some } \mathsf{c} \in \mathcal{R}\mathbf{z}\} \\
&\quad \cup \{\top \mid (\mathsf{b},\omega) \in X, \text{ but there is no } (\mathsf{c},\omega) \in [\![\phi]\!]\} \\
X[\![x := f]\!] &= \{(\mathsf{b},\omega[x \mapsto [\![f]\!]\omega]) \mid (\mathsf{b},\omega) \in X\} \\
X[\![x := *]\!] &= \{(\mathsf{b}\,q,\omega[x \mapsto q]) \mid q \in \mathbb{Q}\} \\
X[\![\alpha;\beta]\!] &= (X[\![\alpha]\!])[\![\beta]\!] \\
X[\![\alpha \cup \beta]\!] &= X_{[0]}[\![\alpha]\!] \cup X_{[1]}[\![\beta]\!] \\
X[\![\alpha^*]\!] &= \bigcap\{Z_{[0]} \subseteq \mathsf{Poss} \mid X \cup (Z_{[1]}[\![\alpha]\!]) \subseteq Z\} \\
X[\![\alpha^d]\!] &= X\langle\!\langle\alpha\rangle\!\rangle
\end{aligned}$$

Angelic tests $?\phi$ end in the current state ω with remaining realizer $\pi_1\mathsf{b}$ if Angel can realize ϕ with $\pi_0\mathsf{b}$, else end in early Demon victory $\bot$. Demonic tests dually require Demon to present a realizer c as evidence that the precondition holds, which becomes the argument to Angel's higher-order realizer. While Demon is permitted to resolve choices classically, tests require him to present constructive evidence because of proof-relevance: Angel's strategy may branch on *how* Demon proved a test condition. If Demon cannot present a realizer (i.e., because none exists), then the game ends in $\top$ so Angel wins by default. Angelic deterministic assignments consume no realizer and simply update the state, then end. Demonic deterministic assignments $x := f$ deterministically store the value of f in x, just as Angelic assignments do. Angelic nondeterministic assignments $x := *$ retrieve from the realizer a term $\pi_0\mathsf{b}$ which is evaluated in the current state to compute a new value for x. In Demonic nondeterministic assignment $x := *$, Demon chooses to set x to *any* value, which is announced to Angel. Angelic compositions $\alpha;\beta$ first play α, then play β from the resulting state using the resulting continuation. Angelic choice games $\alpha \cup \beta$ use the Angelic projections, which appeal to the realizer ($\pi_0\mathsf{b}$) to decide which branch is taken because Angelic projection means Angel chooses the branch. When Demon plays

the choice game $\alpha \cup \beta$, Demon chooses classically between α and β. The realizer $\pi_1\mathsf{b}$ may be reused between α and β, since $\pi_1\mathsf{b}$ could just invoke $\pi_0\mathsf{b}$ if it must decide which branch has been taken. This definition of Angelic choice (corresponding to constructive disjunction) captures the reality that realizers in CGL, in contrast with most constructive logics, are entitled to observe a game state, but they must do so in computable fashion. In contrast, Demonic choice (Def. 4.11) will use Demonic projections which represent the case where Demon chooses the branch taken. The semantics of Demonic choices will require that Angel is prepared to play *either* branch, rather than a specific branch of her choice ($\pi_0\mathsf{b}$). The semantics of dual game α^d switches control to the opposite player and plays α. Constructivity of Angelic duration control of CGL loops is enforced easily: it is an immediate result of the fact that termination conditions ($\pi_0(\mathsf{b}\,\mathsf{c})$) are computable.

The paragraphs that follow will discuss the semantics of repetition and duality in much greater depth. Before doing so, we note that semantic functions satisfy a monotonicity lemma which will prove useful in the discussion of repetition. For example, monotonicity guarantees that the least fixed-point constructions for Angelic and Demonic loops have least fixed points by Knaster-Tarski (Harel et al., 2000, Thm. 1.12).

Lemma 4.6 (Monotonicity)**.** *Assume $X \subseteq Y$ and let ϕ be an arbitrary CGL formula. In each respective claim, let $\mathsf{b} \in \langle\alpha\rangle\phi\,\mathcal{R}\mathbf{z}$ or $\mathsf{b} \in [\alpha]\phi\,\mathcal{R}\mathbf{z}$ for every realizer b in Y so that the realizers in Y have the shapes expected by the semantic functions, e.g., realizers of Angelic tests are pairs.*

- $X\langle\!\langle\alpha\rangle\!\rangle \subseteq Y\langle\!\langle\alpha\rangle\!\rangle$
- $X[\![\alpha]\!] \subseteq Y[\![\alpha]\!]$

Proof summary. By simultaneous induction on games α for Angel and Demon. For the proof, see Appendix A.4. □

Repetition Semantics. The semantics of Angelic and Demonic loops are defined as (least) fixed points. The use of fixed points is motivated by the fact that, as in other GLs (Platzer, 2015a, 2017b; Parikh, 1983) which also use fixed-point semantics, the semantics of loops must carefully avoid giving an opponent advance notice of loop duration. Notably, the loop game α^* is distinct from a game which chooses a finite duration k and then plays for k repetitions, which we denote $\alpha^{\mathbb{N}}$. In both CGL and GL, an advance-notice loop is distinct from a no-advance-notice loop, and differs in its winning conditions. That is, there exist games α and postconditions ϕ (such as the Demonic Counter example from Section 4.5) such that

$$\langle\alpha^*\rangle\phi \not\leftrightarrow \langle\alpha^{\mathbb{N}}\rangle\phi$$

where the advance-notice loop can be defined by

$$\alpha^{\mathbb{N}} \equiv k := *; \{?k > 0; \alpha; k := k - 1\}^*; ?k \leq 0;$$

for fresh variable k representing the loop duration. The subtlety of game loop semantics also manifests as differences in the set of axioms supported for games loops vs. system loops. For example, the following axiom of DL holds in neither GL nor CGL, because on

the left-hand side, the decision whether to repeat the loop body is made after running α, but on the right-hand side, that decision is made before the final execution of α:

$$\langle\alpha\rangle\langle\alpha^*\rangle\phi \leftrightarrow \langle\alpha^*\rangle\langle\alpha\rangle\phi$$

We first describe our own semantics and then discuss the relation to a different fixed-point semantics for loops which is commonly used in GLs (Platzer, 2015a, 2017b; Parikh, 1983). The final region of Angelic loop α^* from starting region X is the projection $\cdot_{\langle 0\rangle}$ of the intersection of all regions Z which contain both X and $Z_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle$. We describe the intuition behind this construction. The outer left Angelic projection $Z_{\langle 0\rangle}$ says that the final possibilities of the loop should only be possibilities where Angel actually decided to stop the loop, i.e., $(\mathsf{b},\omega) \in Z$ such that Angel decided to stop the loop by returning $[\![\pi_0\mathsf{b}]\!]\omega = 0$. Recall that the typing constraint for loops says that Angel will (always) eventually terminate the loop. The fixed-point construction for Z captures intermediate states of the loop, i.e., it captures loop execution until but not including the decision to terminate, which is instead captured by $Z_{\langle 0\rangle}$. The intuition for this construction is that a loop is allowed to either run for zero iterations ($X \subseteq Z$) or run for at least one iteration $Z_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle \subseteq Z$, where the latter case uses the right Angelic projection $Z_{\langle 1\rangle}$ to ensure the loop body α is only played starting from (proper) possibilities $(\mathsf{b},\omega) \in Z$ where Angel's realizer decides to continue the loop ($[\![\pi_0\mathsf{b}]\!]\omega = 1$).

The Demonic loop semantics are also defined as a *least* fixed point, which is notable because *greatest* fixed points[5] are often used in the semantics of Demonic loops (Platzer, 2015a, 2017b; Parikh, 1983). The semantics of Demonic loops is highly symmetric to the Angelic semantics. The final states of the loop are defined by the left Demonic projection $Z_{[0]}$. Recall that in contrast to Angelic projection, the Demonic projection $Z_{[0]}$ always contains every state of Z. This is as it should be because Demon can choose to stop the loop after any finite number of repetitions. Rather, the difference between Z and $Z_{[0]}$ is that in $Z_{[0]}$, Demon tells Angel that he has chosen to stop the loop by projecting the left element $\pi_0\mathsf{b}$ of each realizer b. In the fixed-point construction, $X \subseteq Z$ indicates that Demon can choose to run the loop for zero iterations, while $Z_{[1]}[\![\alpha]\!] \subseteq Z$ indicates that Demon can choose to run for at least one execution, in which case he uses $Z_{[1]}$ to tell Angel that he has chosen to continue playing the loop.

We briefly reflect on why Demon's loop semantics are defined as least fixed points here, rather than the greatest fixed points used in both the literature (Platzer, 2015a, 2017b; Parikh, 1983) and in Chapter 5. To do so, we briefly discuss the loop semantics from dGL (Platzer, 2015a, 2017b), the classical predecessor of CdGL (Chapter 5). In dGL, as in proposition GL (Parikh, 1983), the semantics of games are defined in backward-chaining fashion, in the sense that *initial* winning-region of a game is determined as a function of the *final* goal region. For a game α and final region (of states) $X \subseteq \mathcal{S}$, dGL writes $\varsigma_\alpha(X)$ for the initial Angelic winning region and $\delta_\alpha(X)$ for the initial Demonic winning region. In

[5]The difference between our use of least fixed points and others' use of greatest fixed points is unrelated to the difference between our description of box modalities as modalities where Angel wins when she plays second and others' description of box modalities as modalities where Demon wins. It is a result, rather, of the fact that our semantics determine final regions from initial regions and not vice versa.

the backward-chaining, winning-region semantics, the semantics of loops are defined by:

$$\varsigma_{\alpha^*}(X) = \bigcap \{Z \subseteq \mathcal{S} \mid X \cup \varsigma_\alpha(Z) \subseteq Z\}$$

$$\delta_{\alpha^*}(X) = \bigcup \{Z \subseteq \mathcal{S} \mid Z \subseteq X \cap \delta_\alpha(Z)\}$$

Angel's winning region is constructed inductively as a least fixed point: Angel can either win in zero turns if she is already in the goal region, or she can win in at least one round if the first round takes her to another state from which the loop is winnable. Demon's winning region is constructed coinductively as a greatest fixed point: for Demon to win, he must already be in the winning region and must remain able to win the loop game (indefinitely into the future) after playing the body α. Coinduction is the natural choice for Demonic loops because winning a Demonic loop requires staying in the goal region indefinitely, no matter how many times we are forced to repeat the loop.

We introduced the dGL semantics of loop games because doing so makes it clear that our forward-chaining definition looks significantly different on the surface, particularly in the case of Demon. Our Demonic loop semantics are constructed using a *least* fixed point, but the winning-region semantics use a *greatest* fixed point. Because (Angel's) strategies for Demonic loops are coinductive in nature, we should expect coinduction to make an appearance in the semantics of Demonic loops, just as it did in dGL. It does, but CGL uses coinduction in the *realizer* language rather than the definition of the semantic function. Coinductive realizers allow writing strategies which can play a Demonic loop indefinitely.

The use of a least fixed point for Demonic loop semantics is made less surprising upon remembering that we compute the *final* region as a function of the *initial* region, and that realizers are given to the semantic function as an argument, i.e., as one component of each proper possibility in the initial region. Given an initial state and realizer for Angel, the set of final possibilities is naturally understood as inductive for both Angelic and Demonic loops. Specifically, the forward-chaining semantics of both Angelic and Demonic loops are constructed by splitting into two cases: either execution finishes in zero iterations or one iteration is executed after which the loop potentially repeats. The *least* such set of states is the set of all final states reached after a finite execution, regardless of whether the loop is Angelic or Demonic. Rather, Angelic and Demonic loops differ in their projection operators and in which player controls the body. Angelic projection operators consult Angel's strategy to decide whether to stop the loop, so that for a fixed realizer, the loop duration will depend only on choices made by the opponent Demon during each iteration of the loop body. In contrast, the use of Demonic projection in the Demonic semantics means that the loop is always allowed to repeat for any finite number of repetitions, without consulting Angel. Not only is an inductive definition natural, but any attempt at a coinductive definition of the forward-chaining semantics would likely be incorrect, because we wish to capture *only* execution traces where Demon eventually stops the loop, and a coinductive definition of the final region would likely include infinite execution traces.

We now discuss how our fixed-point semantics can be related to an iterative semantics which is sometimes more convenient for use in proofs: upon proving that every loop execution terminates in finitely many steps, we can write a semantics which partitions the

execution traces of a loop by their duration. This iterative semantics is useful because it allows inductive semantic proofs to use induction on the natural numbers, which is often easier to understand than induction on fixed-point constructions. Before introducing the iterative semantics and proving its equivalence, we prove a key formal result: our forward-chaining semantics is Scott-continuous. Scott-continuity holds in the backward-chaining dGL semantics when applied to *systems* (Platzer, 2015a, Lem. 3.7), but not all games. Scott-continuity is important to understanding our semantics because Kleene's fixed-point theorem (Cousot & Cousot, 1979) guarantees that iteration of a Scott-continuous operator reaches its (least) fixed point within at most ω iterations (i.e., its closure ordinal is at most ω), thus the solution of the least fixed-point construction is equal to the union of all finite iterations: i.e., the final region of a loop is the union of all finite-duration loop executions.

As alluded to in the introduction of Section 4.6, our forward-chaining semantics bridge the worlds of games and systems by requiring Angel to commit to a strategy, after which the execution of a fixed strategy behaves like a system. Because system semantics are Scott-continuous, it is unsurprising that our semantics, which describe how a game evolves once fixing Angel's strategy has caused it to behave like a system, is as well.

Lemma 4.7 (Scott-continuity)**.** *Let α be a game and ϕ a formula. Let $\{X_i\}$ be a (non-empty) family of regions where i ranges over some index set J.*

In each claim and for all $i \in J$, respectively let $\mathsf{b} \in \langle\alpha\rangle\phi\,\mathcal{R}\mathbf{z}$ or $\mathsf{b} \in [\alpha]\phi\,\mathcal{R}\mathbf{z}$ for all realizers b in each X_i, to ensure the semantics of α are well-defined. Then

$$\bigcup_{i\in J}(X_i\langle\!\langle\alpha\rangle\!\rangle) = (\bigcup_{i\in J} X_i)\langle\!\langle\alpha\rangle\!\rangle$$
$$\bigcup_{i\in J}(X_i[\![\alpha]\!]) = (\bigcup_{i\in J} X_i)[\![\alpha]\!]$$

In addition, the Angelic and Demonic projection operators are all Scott-continuous.

Proof summary. The proofs for the projection operators are direct. The proof of each game claim shows that the set on each side includes the other as a subset, thus they are equal. The converse directions (i.e., left-hand side is a subset of right-hand-side) hold by Lemma 4.6 because $X_i \subseteq \bigcup_{i\in J} X_i$ for each $i \in J$. The forward directions are proved by simultaneous induction on games for Angel and Demon. Each of the two cases for loops uses an inner induction on the fixed-point construction from the loop semantics. The full proof is in Appendix A.4. □

Next, we will show that the fixed-point semantics of a loop is equivalent to the union of finite iterations of the loop. To do so, we introduce an "iteration" operator which iterates the semantics of a game:

Definition 4.12 (Iterative CGL semantics)**.** We define the k-step Angelic and Demonic iterations recursively for $k \in \mathbb{N}$, i.e.,

$$X\langle\!\langle\alpha\rangle\!\rangle^0 = X \qquad X\langle\!\langle\alpha\rangle\!\rangle^{k+1} = (X\langle\!\langle\alpha\rangle\!\rangle^k)_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle$$
$$X[\![\alpha]\!]^0 = X \qquad X[\![\alpha]\!]^{k+1} = (X[\![\alpha]\!]^k)_{\langle 1\rangle}[\![\alpha]\!]$$

Remark 4.1 (Pre-iteration and post-iteration). Pre-iteration and post-iteration agree in the following sense:

$$(X_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle)\langle\!\langle\alpha\rangle\!\rangle^k = (X\langle\!\langle\alpha\rangle\!\rangle^k)_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle \qquad (X_{[1]}[\![\alpha]\!])[\![\alpha]\!]^k = (X[\![\alpha]\!]^k)_{[1]}[\![\alpha]\!]$$

Proof summary. By induction on k (Appendix A.4). □

The iterative semantics agree with the fixed-point semantics:

Lemma 4.8 (Alternative semantics)**.** *The CGL fixed-point definition of repetition agrees with the iterative definition with closure ordinal ω:*

$$X\langle\!\langle\alpha^*\rangle\!\rangle = \bigcup_{k\in\mathbb{N}}(X\langle\!\langle\alpha\rangle\!\rangle^k)_{\langle 0\rangle} \qquad X[\![\alpha^*]\!] = \bigcup_{k\in\mathbb{N}}(X[\![\alpha]\!]^k)_{[0]}$$

Proof Summary. By Scott-continuity and Kleene's fixed-point theorem. □

While winning-region semantics of GLs also admit an inflationary characterization along the lines of our iterative semantics (Platzer, 2015a), closure ordinals in winning-region semantics often exceed ω (Platzer, 2015a) so that an inflationary characterization often requires (Platzer, 2015a) unions indexed by large ordinals. Scott-continuity simplifies our semantics in that we can characterize loop semantics with unions over natural numbers and avoid the use of large ordinals. Because the iterative and fixed-point semantics are equivalent, we could have chosen to simply define the semantics of a loop as the union of finite repetitions of the loop body. However, we chose to define the semantics as a fixed-point both in order to evoke a mental connection with the backward-chaining semantics of loops and in order to force ourselves to explore the relationship between iterative and fixed-point semantics.

We summarize our exploration of loop semantics by emphasizing that the fundamental nature and fundamental challenges of game loop semantics are the same between GL and CGL and that Lemma 4.8 does *not* imply an advance-notice semantics for CGL, nor is the following loop-reordering axiom of DL a valid axiom of CGL:

$$\langle\alpha\rangle\langle\alpha^*\rangle\phi \leftrightarrow \langle\alpha^*\rangle\langle\alpha\rangle\phi$$

In CGL, (and GL generally) winning the right-hand side of the loop-reordering axiom is more difficult than the left because Angel is not allowed to decide to stop *after* ϕ is achieved; she must commit to terminate one round before termination actually occurs. Over *systems* in dL, in contrast, the two sides are equal because system loop durations are only nondeterministic, not adversarial. Constructivity aside, the major difference between GL and CGL is not the meaning of loops, but the style of semantic presentation. Because a game with a fixed strategy acts much like a system, CGL is amenable to a forward-chaining semantics which determines final states as a function of initial states. The missing link between CGL semantics and DL semantics, and the reason for different axioms between DL and CGL, lies in the definition of validity for CGL formulas: a CGL game modality $\langle\alpha\rangle\phi$, is valid iff there *exists* an Angelic strategy for α where postcondition ϕ holds for *all* classical

Demon behaviors. To show a modality, Angel first picks her strategy, then analyzes the system-like evolution of the game under the strategy. Needing to find a strategy can make a formula harder to show: in the loop reordering axiom, for example, the right-hand side does not always have a strategy for the postcondition when the left side does.

We have discussed this topic at length not because the forward-chaining semantics are fundamentally more complicated, but because the departure from backward-chaining GL semantics may come as a surprise to readers familiar with the backward-chaining semantics. On the contrary, forward-chaining semantics allow our semantic proofs about games to use techniques such as induction on natural numbers that are familiar from analysis of loops in DL. Insofar as natural-number induction is simpler than fixed-point induction, we enable simpler semantic proofs.

Duality Semantics. To play the dual game α^d, the players take turns: if Angel was playing, Demon takes over gameplay of α with Angel in the "opponent" role, and vice-versa. In *classical* GL, this characterization of duality is interchangeable with the definition of α^d as the game where Angel plays against herself as an adversary, i.e. Angel tries to lose α and wins α^d if α is not losable. The two characterizations are *not* interchangeable in CGL because the Determinacy Axiom (all games have winners) of GL is not valid in CGL:

Remark 4.2 (Indeterminacy). The classically-equivalent determinacy axiom schemata of classical GL, $\neg\langle\alpha\rangle\neg\phi \to [\alpha]\phi$ and $\langle\alpha\rangle\neg\phi \vee [\alpha]\phi$, are not valid in CGL, because they imply double negation elimination. Intuitively, they should not be valid in CGL because if they were, the existence of Angelic winning strategies for arbitrary games would be decidable.

Remark 4.3 (Classical duality). In classical GL, Angelic dual games are characterized by the axiom schema $\langle\alpha^d\rangle\phi \leftrightarrow \neg\langle\alpha\rangle\neg\phi$, which is not valid in in CGL. It is classically interdefinable with $\langle\alpha^d\rangle\phi \leftrightarrow [\alpha]\phi$.

The determinacy axiom is not valid in CGL, so CGL relies on the classically-equivalent duality axiom $\langle\alpha^d\rangle\phi \leftrightarrow [\alpha]\phi$ which *is* valid in CGL. The difference between classical and constructive duality axioms helps to explain the different terminologies for Angel and Demon in classical vs. constructive GL. In classical GL, two-player games can be understood as games with one player who competes with themself, so it is natural to say that a single player Angel is always the player next-to-move. In CGL, gameplay against an opponent cannot be reduced to self-play, so we characterize duality as true turn-taking between distinct players Angel and Demon. In our player terminology, we read the axiom $\langle\alpha^d\rangle\phi \leftrightarrow [\alpha]\phi$ as saying that Angel has a winning strategy where she moves first in α^d with postcondition ϕ iff she has a winning strategy for α with postcondition ϕ where Demon moves first.

Semantics Examples. The realizability semantics of games are subtle on a first read, so we provide examples of realizers. In these examples, the state argument ω is implicit, and we refer to $\omega(x)$ simply as x for brevity. To understand the (defined) realizers for loops, you are encouraged to read ahead to the definitions of rules $\langle*\rangle$I and $[*]$I in Section 4.7, because the defined realizers follow the same structure as the proof rules.

Recall that $[?\phi]\psi$ and $\phi \to \psi$ are equivalent. For any ϕ, the identity function $(\Lambda x : \phi\, \mathcal{R}\mathbf{z}.\ x)$ is a $(\phi \to \phi)$-realizer: for every ϕ-realizer x which Demon presents, Angel can

present the same x as evidence of ϕ. This confirms expected behavior per propositional constructive logic: the identity function is the proof of self-implication.

In example formula $\langle\{x := *\}^d; \{x := x \cup x := -x\}\rangle x \geq 0$, Demon gets to set x, then Angel decides whether to negate x in order to make it nonnegative. It is realized by $(\Lambda x : \mathbb{Q}.\ ((\texttt{if}\ (x < 0)\ 1\ \texttt{else}\ 0), \epsilon))$ where the comparison $x < 0$ is a rational comparison, which is decidable. First, Demon announces a new value of x. Once the new value of x has been updated in the state, Angel's strategy is to check the sign of x, taking the right branch when x is negative and the left branch otherwise. In general, a conditional realizer tests the truth of a Boolean term (here, $x < 0$) in the current state, then reduces to the corresponding branch. In this example, each branch contains a deterministic assignment which consumes no realizer, then the postcondition $x \geq 0$ has trivial realizer ϵ.

Consider the formula $\langle\{x := x + 1\}^*\rangle x \geq y$, where Angel's winning strategy is to repeat the loop until $x \geq y$, which will occur as x increases. While Angelic loop realizers can be defined manually, most realizers will follow a fixed format (Def. A.3) in practice. The Angelic loop realizer $\mathsf{ind}(\mathsf{b}, \mathsf{c}, \mathsf{d})_{\mathcal{M}}$ encodes a convergence argument built on a loop termination metric $\mathcal{M}$, a base case b, inductive step c, and postcondition step d. We now identify which values of $\mathsf{b}, \mathsf{c}, \mathsf{d}$, and $\mathcal{M}$ will realize formula $\langle\{x := x + 1\}^*\rangle x \geq y$.

The base case establishes that some invariant predicate φ holds initially. In this example, the trivial predicate $\varphi \equiv \mathit{true}$ suffices. Thus, $\mathsf{b} \equiv \epsilon$ suffices to realize the base case, which simply proves that φ holds initially.

The metric $\mathcal{M}$ is defined by a term x. Like all termination metrics, $\mathcal{M}$ has strict and non-strict ordering relations $\succcurlyeq$ and $\succ$, which in this example are the reverse ordering:

$$x \succcurlyeq y \leftrightarrow x \leq y$$
$$x \succ y \leftrightarrow x \leq y - 1$$

which decreases until terminating at $x \geq y$. The metric is effectively-well-founded because $\mathcal{M}$ decreases by at least the constant amount 1 in each iteration and because the bound y is constant with respect to the loop.

Realizers for the inductive step are more technically involved. They first accept an argument $\mathcal{M}_0 : \mathbb{Q}$ which computes the value of the termination metric at the start of the loop. They then take a realizer for the conjunction $(\varphi \wedge (\mathcal{M}_0 = \mathcal{M} \wedge \mathcal{M} \succ \mathbf{0}))$ as an argument, that is, a triple of realizers. After playing α, the inductive step realizer must yield $(\varphi \wedge \mathcal{M}_0 \succ \mathcal{M})$, meaning it must demonstrate the invariant predicate and also show that the metric has decreased. In this example, the trivial loop body $x := x + 1$ contributes nothing to the realizer because no realizer is consumed by evaluating a deterministic assignment which requires making no decisions. Thus, the body of our inductive step realizer is simply a pair of trivial realizers which would realize φ and $\mathcal{M}_0 \succ \mathcal{M}$ when evaluated *after* the assignment:

$$\mathsf{c} \equiv (\Lambda \mathcal{M}_0 : \mathbb{Q}.\ \Lambda \mathsf{rz} : (\mathit{true} \wedge (\mathcal{M}_0 = x \wedge x < y))\, \mathcal{R}\mathbf{z}.\ (\epsilon, \epsilon))$$

where rz is a realizer variable name.

The postcondition realizer only needs to conclude $x \geq y$ from $(\mathit{true} \wedge x \geq y)$, thus it suffices to choose $\mathsf{d} \equiv \epsilon$.

The implementation of $\mathsf{ind}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}}$ assembles the Angelic loop realizer from the components $\mathsf{b}, \mathsf{c}, \mathsf{d}$, and $\mathcal{M}$. In short, it first tests whether the metric has terminated ($x > y$). Recall that Angelic loop realizers return pairs whose first component is 0 when the loop terminates or 1 when the loop continues, so the test $x > y$ determines the first component. In the case where the loop terminates immediately, b and d are used to show that the postcondition already holds. In the case where the loop continues, c is applied to execute one step of the loop, after which a corecursive call of ind continues execution. While termination arguments for loops are inductive, the realizer is constructed coinductively because even though Angelic loops have finite durations, the duration can depend on dynamic choices by Demon and thus there may not be a statically-knowable bound on duration. A coinductive construction easily handles executions whose durations are not bounded in advance because it is infinitary.

We now consider a subtle example of a Demonic loop realizer for the following formula:

$$[?x > 0; \{x := x + 1\}^*]\exists y\,(y \leq x \wedge y > 0)$$

Typical realizers for Demonic loops use the defined constructor $\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})$ for invariant arguments. The realizer b establishes that some invariant ψ holds initially, the coinductive step c shows that the invariant is maintained after any one iteration, and the postcondition step d shows that a postcondition follows from the invariant. In this example, our strategy for Angel is to record the initial value of x in y, then maintain a proof that $y \leq x$ as x increases. We maintain the invariant $\psi \equiv \exists y\,(y \leq x \wedge y > 0)$. Angel's strategy is represented by the realizer:

$$\Lambda w : (x > 0)\,\mathcal{R}\mathbf{z}.\ \mathsf{gen}((x, (\epsilon, w)),\ \Lambda z : y \leq x\,\mathcal{R}\mathbf{z}.\ (\pi_0 z, (\epsilon, \pi_1(\pi_1 z))),\ \Lambda z : y \leq x\,\mathcal{R}\mathbf{z}.\ z)$$

Initially Demon announces a proof w of $x > 0$. Angel specifies the initial element of the realizer stream by witnessing $\exists y\,(y \leq x \wedge y > 0)$ with $\mathsf{b} = (x, (\epsilon, w))$, where the first component instantiates $y = x$, the trivial second component indicates that $y \leq y$ trivially, and the third component reuses w as a proof of $y > 0$. Demon can choose to repeat the loop arbitrarily. When Demon demands the kth repetition, Demon must supply the proof of repetition $k - 1$, which is bound to z. The kth repetition is then computed by the coinductive step c's body $(\pi_0 z, (\epsilon, \pi_1(\pi_1 z)))$, which plays the next iteration. Because deterministic assignments do not consume a realizer, the body immediately witnesses the invariant $\exists y\,(y \leq x \wedge y > 0)$ again by assigning the same value (stored in $\pi_0 z$) to y, reproving $y \leq x$ with ϵ, then reusing the proof (stored in $\pi_1(\pi_1 z)$) that $y > 0$.

4.7 Proof Calculus

Having settled on the meaning of a game in Section 4.6, we proceed to develop a calculus for proving CGL formulas syntactically. The goal is twofold: the practical motivation, as always, is that when verifying a concrete example, the realizability semantics provide a notion of ground truth, but are impractical for proving large formulas. By developing a syntactic proof calculus, we lay the groundwork for the development of a Logic-User-facing proof language in Chapter 7. The theoretical motivation is that we wish to expose

the computational interpretation of the modalities $\langle\alpha\rangle\phi$ and $[\alpha]\phi$ as the types of winning strategies for a game α that has ϕ as its goal condition, respectively when Angel moves first or second. Since CGL is constructive, such a strategy constructs a proof of the postcondition ϕ while playing the game. By developing a computational interpretation, we lay a theoretical foundation on top of which we will reimplement the (Engineer-facing) synthesis tool VeriPhy in Chapter 8.

As is standard for natural-deduction calculi, a proof in our calculus proves a natural-deduction sequent $(\Gamma \vdash \phi)$ where Γ is a comma-separated list of named formulas. Because the calculus is sound (Section 4.8), every provable sequent is valid, where sequent $(\Gamma \vdash \phi)$ is valid (Def. 4.8) iff formula $\bigwedge\Gamma \to \phi$ is, where $\bigwedge\Gamma$ is the conjunction of all elements of Γ.

To study the computational nature of proofs, we write proof terms explicitly: the main proof judgement $\Gamma \vdash M : \phi$ says proof term M is a proof of ϕ in context Γ, or equivalently a proof of sequent $(\Gamma \vdash \phi)$.

Definition 4.13 (Proof terms)**.** We define the grammar of proof terms M, N, O (sometimes A, B, C, D, E, F) here, before describing the meaning of each proof term when we describe its corresponding proof rule:

$$\begin{aligned} M, N ::= {} & \lambda x : \mathbb{Q}.\ M \mid M\ f \mid \lambda p : \phi.\ M \mid M\ N \mid \langle \ell \cdot M\rangle \mid \langle r \cdot M\rangle \\ & \mid \langle \text{case}_*\ A \text{ of } s \Rightarrow\ B \mid g \Rightarrow C\rangle \mid \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \\ & \mid \text{for}(p : \varphi(\mathcal{M}) = A; q; B)\,\{\alpha\}\,C \mid FP(A, s.\ B, g.\ C) \\ & \mid M \text{ rep } p : \psi.\ N \text{ in } O \mid [\text{unroll } M] \mid \langle \text{stop } M\rangle \mid \langle \text{go } M\rangle \\ & \mid \langle f\tfrac{y}{x} :* p.\ M\rangle \mid \langle\!|\pi_L M|\!\rangle \mid \langle\!|\pi_R M|\!\rangle \mid \langle\!|M, N|\!\rangle \mid \langle\!|\iota\ M|\!\rangle \mid \langle\!|\text{yield } M|\!\rangle \\ & \mid \langle\!|x := f\tfrac{y}{x} \text{ in } p.\ M|\!\rangle \mid \text{unpack}(M, py.\ N) \mid M \circ_p N \mid \mathsf{FO}[\phi](M) \mid p \end{aligned}$$

where p, q, ℓ, r, s, and g are *proof variables*, that is, variables that range over proof terms of a given proposition. In contrast to the assignable *program variables*, the proof variables are given their meaning by substitution and are scoped lexically, not globally.

Metavariables M, N, O (sometimes A, B, C, D, E, F) range over arbitrary proof terms. We adapt propositional proof terms such as pairing, disjoint union, and lambda-abstraction to our context of game logic. To support first-order games, we include first-order proof terms and new terms for new features: dual, assignment, and repetition games. Some constructs, such as pairing, arise in proof terms for both box and diamond formulas. The notation $\langle\!|M, N|\!\rangle$, for example, is used when we wish to uniformly discuss pairing proof terms for both diamond and box formulas.

We now develop the calculus by starting with standard constructs and working toward the novel constructs of CGL. The assumptions p in Γ are named, so that they may appear as variable proof terms p. We write $\Gamma\frac{y}{x}$ and $M\frac{y}{x}$ for the renaming of program variable x to y and vice versa in context Γ or proof term M, respectively. Proof rules for state-modifying constructs perform explicit renamings, which both ensures they are applicable as often as possible and also ensures that references to proof variables support an intuitive notion of lexical scope. Likewise, Γ_x^f and M_x^f are the substitutions of term f for program variable x. We use distinct notation to substitute proof terms for proof variables while avoiding capture: $[N/p]M$ substitutes proof term N for proof variable p in proof term M.

$$(\langle\cup\rangle\text{E})\quad \frac{\Gamma \vdash A : \langle\alpha\cup\beta\rangle\phi \quad \Gamma, \ell : \langle\alpha\rangle\phi \vdash B : \psi \quad \Gamma, r : \langle\beta\rangle\phi \vdash C : \psi}{\Gamma \vdash \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : \psi}$$

$$(\langle\cup\rangle\text{I1})\quad \frac{\Gamma \vdash M : \langle\alpha\rangle\phi}{\Gamma \vdash \langle \ell \cdot M\rangle : \langle\alpha\cup\beta\rangle\phi}$$

$$(\langle\cup\rangle\text{I2})\quad \frac{\Gamma \vdash M : \langle\beta\rangle\phi}{\Gamma \vdash \langle r \cdot M\rangle : \langle\alpha\cup\beta\rangle\phi}$$

$$([\cup]\text{I})\quad \frac{\Gamma \vdash M : [\alpha]\phi \quad \Gamma \vdash N : [\beta]\phi}{\Gamma \vdash [M, N] : [\alpha\cup\beta]\phi}$$

$$(\langle?\rangle\text{I})\quad \frac{\Gamma \vdash M : \phi \quad \Gamma \vdash N : \psi}{\Gamma \vdash \langle M, N\rangle : \langle?\phi\rangle\psi}$$

$$([?]\text{I})\quad \frac{\Gamma, p : \phi \vdash M : \psi}{\Gamma \vdash (\lambda p : \phi.\ M) : [?\phi]\psi}$$

$$([?]\text{E})\quad \frac{\Gamma \vdash M : [?\phi]\psi \quad \Gamma \vdash N : \phi}{\Gamma \vdash (M\ N) : \psi}$$

$$([\cup]\text{E1})\quad \frac{\Gamma \vdash M : [\alpha\cup\beta]\phi}{\Gamma \vdash [\pi_L M] : [\alpha]\phi}$$

$$([\cup]\text{E2})\quad \frac{\Gamma \vdash M : [\alpha\cup\beta]\phi}{\Gamma \vdash [\pi_R M] : [\beta]\phi}$$

$$(\text{hyp})\quad \frac{}{\Gamma, p : \phi \vdash p : \phi}$$

$$(\langle?\rangle\text{E1})\quad \frac{\Gamma \vdash M : \langle?\phi\rangle\psi}{\Gamma \vdash \langle \pi_L M\rangle : \phi}$$

$$(\langle?\rangle\text{E2})\quad \frac{\Gamma \vdash M : \langle?\phi\rangle\psi}{\Gamma \vdash \langle \pi_R M\rangle : \psi}$$

Figure 4.1: CGL proof calculus: propositional rules.

Some proof terms such as pairs prove both a diamond formula and a box formula. We write $\langle M, N\rangle$ and $[M, N]$ respectively to distinguish the terms or $\langle\!\langle M, N\rangle\!\rangle$ to treat them uniformly. Analogously, a handful of symmetric rules write $\langle\!\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\!\rangle$ to range over $\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$ and $\langle \text{case}_* \ A \text{ of } s \Rightarrow\ B \mid g \Rightarrow C\rangle$, likewise they write $\langle\!\langle \ell \cdot M\rangle\!\rangle$ to range over $\ell \cdot M$ and $\langle \text{stop } A\rangle$, and $\langle\!\langle r \cdot M\rangle\!\rangle$ to range over $\langle r \cdot M\rangle$ and $\langle \text{go } M\rangle$. That is, the brackets $\langle\!\langle\cdot\rangle\!\rangle$ are used in some rules for a uniform treatment of the disjunction-style proof terms for Angelic choices and Angelic loops. In other rules, however, a symmetric treatment is not possible.

Likewise, we abbreviate $\langle\!\langle\alpha\rangle\!\rangle\phi$ when the same rule works for both diamond and box modalities. When the notation $\langle\!\langle\alpha\rangle\!\rangle\phi$ occurs multiple times in a single rule, each occurrence refers to the same kind of modality: diamond or box. We write $[\!(\alpha)\!]\phi$ to denote the dual modality (box or diamond, respectively). For example, when $\langle\!\langle\alpha\rangle\!\rangle\phi$ is $[\alpha]\phi$, then $[\!(\alpha)\!]\phi$ is $\langle\alpha\rangle\phi$ and vice-versa. The proof terms $\langle x := f\frac{y}{x} \text{ in } p.\ M\rangle$ and $[x := f\frac{y}{x} \text{ in } p.\ M]$ introduce an auxiliary ghost variable y for the old value of x, which improves completeness without requiring manual ghost steps. In $\text{for}(p : \varphi(\mathcal{M}) = A; q; B)\,\{\alpha\}\,C$, the metric term $\mathcal{M}$ is not literally an argument to formula φ, rather the notation $\varphi(\mathcal{M})$ is suggestive of the fact that φ and $\mathcal{M}$ both depend on changing state, and is convenient notation for making $\mathcal{M}$ explicit in the proof term syntax.

The propositional proof rules of CGL are in Fig. 4.1. Formula $[?\phi]\psi$ is constructive implication, so rule [?]E with proof term $M\ N$ eliminates M by supplying an argument proof term N that proves the test condition. Lambda terms $(\lambda p : \phi.\ M)$ are introduced by rule [?]I by extending the context Γ. While this rule is standard, it is worth emphasizing that here p is a *proof variable* for which a proof term (like N in rule [?]E) may be substituted, and that the *game state* is untouched by rule [?]I. Constructive disjunction (between the

$$([*]\text{E})\quad \frac{\Gamma \vdash M : [\alpha^*]\phi}{\Gamma \vdash [\text{unroll } M] : \phi \wedge [\alpha][\alpha^*]\phi} \qquad ([*]\text{R})\quad \frac{\Gamma \vdash M : \phi \wedge [\alpha][\alpha^*]\phi}{\Gamma \vdash [\text{roll } M] : [\alpha^*]\phi}$$

$$(\langle*\rangle\text{S})\quad \frac{\Gamma \vdash M : \phi}{\Gamma \vdash \langle \text{stop } M\rangle : \langle\alpha^*\rangle\phi} \qquad (\langle\!\langle^d\rangle\!\rangle\text{I})\quad \frac{\Gamma \vdash M : [\![\alpha]\!]\phi}{\Gamma \vdash \langle\!\langle \text{yield } M\rangle\!\rangle : \langle\!\langle\alpha^d\rangle\!\rangle\phi}$$

$$(\langle*\rangle\text{G})\quad \frac{\Gamma \vdash M : \langle\alpha\rangle\langle\alpha^*\rangle\phi}{\Gamma \vdash \langle \text{go } M\rangle : \langle\alpha^*\rangle\phi} \qquad (\langle\!\langle;\rangle\!\rangle\text{I})\quad \frac{\Gamma \vdash M : \langle\!\langle\alpha\rangle\!\rangle\langle\!\langle\beta\rangle\!\rangle\phi}{\Gamma \vdash \langle\!\langle\iota\ M\rangle\!\rangle : \langle\!\langle\alpha;\beta\rangle\!\rangle\phi}$$

$$(\langle*\rangle\text{C})\quad \frac{\Gamma \vdash A : \langle\alpha^*\rangle\phi \quad \Gamma, s : \phi \vdash B : \psi \quad \Gamma, g : \langle\alpha\rangle\langle\alpha^*\rangle\phi \vdash C : \psi}{\Gamma \vdash \langle \text{case}_*\ A \text{ of } s \Rightarrow\ B \mid g \Rightarrow C\rangle : \psi}$$

$$(\text{M})\quad \frac{\Gamma \vdash M : \langle\alpha\rangle\phi \quad \Gamma_{\frac{\vec{y}}{\text{BV}(\alpha)}}, p : \phi \vdash N : \psi}{\Gamma \vdash M \circ_p N : \langle\alpha\rangle\psi}\ {}^{1}$$

[1]Variables $\vec{y}$ are fresh and $|\vec{y}| = |\,\text{BV}(\alpha)|$

Figure 4.2: CGL proof calculus: some non-propositional rules.

branches $\langle\alpha\rangle\phi$ and $\langle\beta\rangle\phi$) is the choice $\langle\alpha \cup \beta\rangle\phi$. The introduction rules for injections are $\langle\cup\rangle$I1 and $\langle\cup\rangle$I2, and case-analysis is performed with rule $\langle\cup\rangle$E, with two branches that prove a common consequence from each disjunct. The modal formulas $\langle?\phi\rangle\psi$ and $[\alpha \cup \beta]\phi$ are conjunctive. Conjunctions are introduced by rules $\langle?\rangle$I and $[\cup]$I as pairs, and eliminated by rules $\langle?\rangle$E1, $\langle?\rangle$E2, $[\cup]$E1, and $[\cup]$E2 as projections. Lastly, rule hyp says formulas in the context hold by assumption.

We now begin considering non-propositional rules, starting with the simplest ones. The majority of the rules in Fig. 4.2, while thoroughly useful in proofs, are computationally trivial. The repetition rules ($[*]$E and $[*]$R) fold and unfold the notion of repetition as iteration. The rolling and unrolling terms are named in analogy to the *iso-recursive* treatment of recursive types (Vanderwaart et al., 2003), where an explicit operation is used to expand and collapse the recursive definition of a type.

Rules $\langle*\rangle$C,$\langle*\rangle$S and $\langle*\rangle$G are the destructor and injectors for $\langle\alpha^*\rangle\phi$, which are similar to those for $\langle\alpha \cup \beta\rangle\phi$. The duality rules ($\langle\!\langle^d\rangle\!\rangle$I) say the dual game is proved by proving the game where roles are reversed, i.e., by alternating between box and diamond modalities. The sequencing rules ($\langle\!\langle;\rangle\!\rangle$I) say a sequential game is played by playing the first game with the goal of reaching a state where the second game is winnable.

Among these rules, monotonicity M is especially computationally rich. The notation $\Gamma_{\frac{\vec{y}}{\text{BV}(\alpha)}}$ says that in the second premise, the assumptions in Γ have all bound variables of α (written $\text{BV}(\alpha)$) renamed to fresh variables $\vec{y}$ for completeness. The definition of $\text{BV}(\cdot)$ is in Appendix A.4. In practice, Γ usually contains some assumptions on variables that are not bound, which we wish to access without writing them explicitly in ϕ. Rule M is used to execute programs right-to-left, giving shorter, more efficient proofs. It can also be used to derive the Hoare-style sequential composition rule, which is frequently used to reduce the number of case splits. Note that like every GL, CGL is subnormal, so the modal modus ponens axiom K and Gödel generalization rule G (axiom K and rule G in Chapter 2) are not sound, and rule M takes over much of the role they usually serve. On the surface, rule

$$(\langle:*\rangle\text{E})\quad \frac{\Gamma \vdash M : \langle x := * \rangle \phi \quad \Gamma\tfrac{y}{x}, p : \phi \vdash N : \psi}{\Gamma \vdash \text{unpack}(M, py.\ N) : \psi}\ {}^{1} \qquad ([:*]\text{E})\quad \frac{\Gamma \vdash M : [x := *]\phi}{\Gamma \vdash (M\ f) : \phi_x^f}\ {}^{4}$$

$$(\langle:*\rangle\text{I})\quad \frac{\Gamma\tfrac{y}{x}, p : (x = f\tfrac{y}{x}) \vdash M : \phi}{\Gamma \vdash \langle f\tfrac{y}{x} :* p.\ M\rangle : \langle x := * \rangle \phi}\ {}^{2} \qquad ([:*]\text{I})\quad \frac{\Gamma\tfrac{y}{x} \vdash M : \phi}{\Gamma \vdash (\lambda x : \mathbb{Q}.\ M) : [x := *]\phi}\ {}^{5}$$

$$(\langle\!|:=|\!\rangle\text{I})\quad \frac{\Gamma\tfrac{y}{x}, p : (x = f\tfrac{y}{x}) \vdash M : \phi}{\Gamma \vdash \langle\!|x := f\tfrac{y}{x} \text{ in } p.\ M|\!\rangle : \langle\!|x := f|\!\rangle \phi}\ {}^{3}$$

[1] y fresh in $\Gamma, \langle x := * \rangle \phi$, and ψ, and $x \notin \text{FV}(\psi)$
[2] y fresh in Γ, f, and $\langle x := * \rangle \phi$, and p fresh in Γ
[3] y fresh in Γ and $\langle\!|x := f|\!\rangle \phi$
[4] ϕ_x^f admiss.
[5] y fresh in Γ and $[x := *]\phi$

Figure 4.3: CGL proof calculus: first-order games.

M simply says games are monotonic: a game's goal proposition may freely be replaced with a weaker one. One might wonder whether rule M is essential to the expressive power of the proof calculus, i.e., whether or not it can be implemented using the other rules. This topic is explored in Section 4.9: while we do not go so far as to make rule M an admissible rule whose instances all have derivations using other rules, the operational rules for proof terms of rule M demonstrate how many of its instances can be reduced to other proof rules which reason left-to-right. In future work, those rules suggest it is possible to design a calculus where rule M would be admissible. Admissibility of rule M is interesting because rule M reasons about programs from right-to-left while other rules reason left-to-right, thus the admissibility of rule M would imply that right-to-left reasoning can be reduced to left-to-right reasoning.

Note that in checking $M \circ_p N$, the context Γ has the bound variables of α renamed freshly to some $\vec{y}$ within N, as required to maintain soundness across execution of α.

Next, we consider *first-order* rules, i.e., those which deal with first-order programs that modify *program* variables. The first-order rules are given in Fig. 4.3. In rule $\langle:*\rangle$E, $\text{FV}(\psi)$ is the set of *free variables* of ψ, the variables which can influence its meaning. The definition of $\text{FV}(\cdot)$ is in Appendix A.4 and is identical to the discrete fragment of the definition of syntactic free variables in classical dGL, for example as implemented in KeYmaera X (Fulton et al., 2015). Rather, proofs of properties about free variables in CGL (such as Lemma 4.11 in Section 4.8) differ from those of dGL in their justification, because CGL is constructive. Nondeterministic assignment provides quantification over rational-valued *program* variables. Rule $[:*]$I is universal, with proof term $(\lambda x : \mathbb{Q}.\ M)$. While this notation is suggestive, the difference vs. the function proof term $(\lambda p : \phi.\ M)$ is essential: the proof term M is checked (resp. evaluated) in a state where the program variable x has changed from its initial value. For soundness, rule $[:*]$I renames x to fresh program variable y throughout context Γ, written $\Gamma\frac{y}{x}$. This means that M can freely refer to all facts of the full context, but they now refer to the state as it was before x received a new value. Elimination rule $[:*]$E then allows instantiating x to a term f. Existential quantification is introduced by rule $\langle:*\rangle$I whose proof term $\langle f\frac{y}{x} :* p.\ M\rangle$ is like a dependent pair plus bound

$$(\langle*\rangle\text{I})\quad \frac{\Gamma \vdash A : \varphi \quad p : \varphi, q : \mathcal{M}_0 = \mathcal{M} \succ \mathbf{0} \vdash B : \langle\alpha\rangle(\varphi \wedge \mathcal{M}_0 \succ \mathcal{M}) \quad p : \varphi, q : \mathbf{0} \succcurlyeq \mathcal{M} \vdash C : \phi}{\Gamma \vdash \text{for}(p : \varphi(\mathcal{M}) = A; q; B)\,\{\alpha\}\,C : \langle\alpha^*\rangle\phi}\ {}^{1}$$

$$([*]\text{I})\quad \frac{\Gamma \vdash M : \psi \quad p : \psi \vdash N : [\alpha]\psi \quad p : \psi \vdash O : \phi}{\Gamma \vdash (M \text{ rep } p : \psi.\ N \text{ in } O) : [\alpha^*]\phi}$$

$$(\text{FP})\quad \frac{\Gamma \vdash A : \langle\alpha^*\rangle\phi \quad s : \phi \vdash B : \psi \quad g : \langle\alpha\rangle\psi \vdash C : \psi}{\Gamma \vdash FP(A, s.\ B, g.\ C) : \psi}$$

[1] $\mathcal{M}_0$ fresh, $\mathcal{M}$ effectively-well-founded

Figure 4.4: CGL proof calculus: loops.

renaming of variable x to y. The witness f is a CGL term, as always. We write $\langle f_{\frac{y}{x}} :* M\rangle$ for short when y is not referenced in M. It is eliminated in rule $\langle:*\rangle$E by unpacking the pair, with side condition $x \notin \text{FV}(\psi)$ for soundness. The assignment rules $\langle\!|:=|\!\rangle$I do not quantify, per se, but always update x to the value of the term f, and in doing so introduce an assumption that x and f (suitably renamed) are now equal. Just as the assignment rules do not directly correspond to quantifiers, their proof terms do not directly correspond to any standard first-order proof term. The proof term $\langle\!|x := f\frac{y}{x} \text{ in } p.\ M|\!\rangle$ indicates that the old value of x will be remembered in fresh ghost variable y, then the equality induced by assigning $x := f$ will be remembered in proof variable p while proving the postcondition with proof term M. The proof term $\langle f\frac{y}{x} :* p.\ M\rangle$ for rule $\langle:*\rangle$I likewise says that x will be assigned to the existential witness term f in the proof M of the postcondition, with y storing the old value of x and with proof variable p remembering the equality induced by the assignment. In rules $\langle:*\rangle$I and $\langle\!|:=|\!\rangle$I, program variable y is fresh.

The looping rules in Fig. 4.4, especially $\langle*\rangle$I, are arguably the most sophisticated in CGL. Rule $\langle*\rangle$I provides a strategy to repeat a game α until the postcondition ϕ holds. That strategy provides a kind of correctness proof known as a convergence argument.

There exist several equivalent presentations of convergence arguments. In $\langle*\rangle$I, a convergence argument is divided into two parts: an invariant predicate φ which remains true throughout the loop and a termination metric $\mathcal{M}$ which decreases under a well-ordering $\succ$ until reaching its termination condition $\mathbf{0}$. Our particular presentation of metrics was chosen so that the text of the rule need not change if we wish to use advanced termination metrics in the future. In other logics including dL (Platzer, 2008a, Rule G4), convergence is encoded with a single *variant* predicate $\varphi(x)$ whose argument x decreases, typically by a constant value of 1, in each iteration. Both presentations support computational interpretations, and for example the convergence construct developed in Chapter 7 will follow the traditional dL style more closely than it does ours. In this chapter, our motivation for using an explicit term as a termination metric is mostly a subjective aesthetic preference, though each presentation leads to different subtleties when developing a computational interpretation. Rules in the traditional style bake in the modification of x, so their computational interpretation must bake in code which modifies x. In Chapter 7, for example, x is understood as the index variable of a `for` loop, whose value is modified as part of the `for` loop construct. Our rule also introduces a variable $\mathcal{M}_0$ which stands for the old

value of the metric, but its computational interpretation is arguably simpler in the sense that no special handling is required to modify the value of a special index variable x. On the other hand, our rule can only express termination arguments in terms of pre-existing variables, which means it cannot express (for example) a convergence argument for a loop whose body is a no-op. The traditional style builds in an index variable, which makes it easier to reason about loops which either do not change the state or which change the state so little that a metric cannot be found in terms of existing variables. This is one reason why Chapter 7 uses explicit index variables.

When making a convergence argument, one exhibits an invariant formula φ and termination metric $\mathcal{M}$ with terminal value $\mathbf{0}$ and well-ordering $\succ$. Proof term A shows φ holds initially. Proof term B guarantees $\mathcal{M}$ decreases with every iteration where $\mathcal{M}_0$ is a fresh metric variable which is equal to $\mathcal{M}$ at the antecedent of B and is never modified. Proof term C allows any postcondition ϕ which follows from the invariant and the fact that the metric has terminated ($\varphi \wedge \mathbf{0} \succcurlyeq \mathcal{M}$). Proof term $\mathrm{for}(p : \varphi(\mathcal{M}) = A; q; B)\,\{\alpha\}\,C$ suggests the computational interpretation as a for-loop: proof A shows the invariant holds in the initial state, B shows that each step reduces the termination metric while maintaining the predicate, and C shows that the postcondition follows from the invariant upon termination. In the proof term $\mathrm{for}(p : \varphi(\mathcal{M}) = A; q; B)\,\{\alpha\}\,C$, the proof term C is written after the closing brace as a reminder that it is a proof about the *postcondition* of the loop and is applied *after* the induction. The game α repeats until convergence is reached ($\mathbf{0} \succcurlyeq \mathcal{M}$). By the assumption that metrics are well-founded, convergence is guaranteed in finitely (but arbitrarily) many iterations.

Recall from Section 4.6.1 that our termination metrics must be strong enough to prove loop termination constructively. To ensure loop termination, we require that termination metrics have the effective descending chain condition (Def. 4.7), which is a consequence of effective-well-foundedness (Hofmann et al., 2006). A metric has the effective descending chain condition if it is constructively provable that every chain that is descending under the strict ordering $\succ$ is finite. The effective descending chain condition aids in the soundness proof of rule $\langle * \rangle$I. The intuition behind rule $\langle * \rangle$I is that α^* can be played by repeating the strategy of B until the metric terminates. The effective descending chain condition guarantees that the metric does eventually terminate and that we can detect when it does.

The values of our termination metrics are rational numbers, which are dense (in themselves), thus simple rational orderings $x \geq y$ are not even classically well-founded, let alone do they satisfy the effective descending chain condition. For any constant $c > 0$, the inflated comparison $x \geq y + c$ does induce an order with the effective descending chain condition, call it $\succ_c$. The order is effective because for any descending sequence S ordered by $\succ_c$, a finite upper bound on its length can be computed as a function of its first element S_0, proving that it is finite. Because each successive element decreases by at least c, the length is at most $1 + (S_0 - \mathbf{0})/c$, where $S_0 \succcurlyeq \mathbf{0}$ because $\mathbf{0}$ is the terminal value. In summary, the comparisons $\succcurlyeq$ and $\succ$ cannot merely test that the metric $\mathcal{M}$ has decreased, but must typically test that it has decreased by at least some lower bound c. The lower bound c ensures the effective descending chain condition, which ensures a finite number of iterations until metric $\mathcal{M}$ achieves terminal value $\mathbf{0}$, as desired because premise C expects the guard to have been reached in the final state of the loop which, as with any CGL loop,

must occur after only finitely many iterations.

The same rule $\langle * \rangle$I will be reused in Chapter 5, so we briefly foreshadow the unique challenges that arise when using $\langle * \rangle$I in CdGL as opposed to discrete CGL. Because rational-number comparisons are decidable, loop guards in discrete CGL are easily implemented by comparing the metric term against the bound $\mathbf{0}$. Because the constructive real numbers that will be used in CdGL do not admit exact comparisons, the loop guards in Chapter 5 will be significantly more subtle. That is, CGL assumes loop guards are decidable (because doing so simplifies some semantic proofs), while CdGL does not.

We mention these future challenges now because one might wonder whether soundness of $\langle * \rangle$I relies on decidability of the exact comparison $\mathbf{0} \succcurlyeq \mathcal{M} \vee \mathcal{M} \succ \mathbf{0}$. We will see in Chapter 5 that the text of the rule $\langle * \rangle$I will remain unchanged and it will remain both sound and useful, and that the challenges unique to CdGL will be offloaded into metrics $\mathcal{M}$ and relations $\succ, \succcurlyeq$ which employ additional types and additional term constructs while relaxing the guard decidability assumption.

Rule FP eliminates a loop $\langle \alpha^* \rangle \phi$ by working backwards from its final state. To show that a formula ψ holds in the initial state, rule FP says it suffices to show that ψ holds in the final state and that it is preserved when executing the loop body α "in reverse," so that it (inductively) holds in the initial state. Rule $[*]$I is the well-understood invariant rule for loops, which applies as well to repeated games. Premise O ensures rule $[*]$I supports the generalization of postconditions, a feature which is used in the operational semantics rules of proof terms (Section 4.9), specifically when defining the interaction between rule $[*]$I and rule M. The elimination rule for $[\alpha^*]\phi$ is simply rule $[*]$E. Note that rules such as $\langle * \rangle$I and $[*]$I represent a different design choice regarding ghost variables when compared to rules such as $\langle := \rangle$I: the latter explicitly introduces ghosts for the old values of bound variables, while the former do not. Both approaches are viable and the latter choice better foreshadows the treatment of historical state in Chapter 7, but the former was chosen for looping rules to reduce notational overhead.

Like any program logic, reasoning in CGL consists of first applying program-logic rules to decompose a program until the program has been entirely eliminated, then applying first-order logic principles at the leaves of the proof. The *constructive* theory of rationals is undecidable because it can express[6] the undecidable (Robinson, 1949) *classical* theory of rationals. Thus, facts about rationals can require proof in practice. For the sake of space and since our focus is on program reasoning, we defer an axiomatization of rational arithmetic to future work. We provide a (non-effective!) rule FO which says valid first-order formulas are provable.

$$\text{(FO)} \quad \frac{\Gamma \vdash M : \rho}{\Gamma \vdash \mathsf{FO}[\phi](M) : \phi} \quad \text{where exists } \mathsf{b} \text{ s.t. } \{\mathsf{b}\} \times \mathcal{S} \subseteq [\![\rho \to \phi]\!],\ \rho, \phi \text{ first-order}$$

A special case of rule FO is rule splitRat, which is an effective rule in the setting of rational numbers because all rational term comparisons are decidable.

$$\text{(splitRat)} \quad \Gamma \vdash (\text{split } [f \sim g]) : f \leq g \vee f > g$$

[6]By the standard Gödel-Gentzen translation (Buss, 1998, §3.1.4) from classical first-order logic into constructive first-order logic

The notation $f \sim g$ in the proof term (split $[f \sim g]$) of rule splitRat is mnemonic of the use of $\sim$ to stand for an arbitrary comparison symbol in Section 4.4. The notation $f \sim g$ was chosen rather than $f \leq g$ because we do not know in advance whether $f \leq g$ holds and the purpose of splitRat is rather to tell us whether $f \leq g$ or $f > g$ is the case. Rule splitRat can be generalized to decide termination metrics ($\mathbf{0} \succcurlyeq \mathcal{M} \vee \mathcal{M} \succ \mathbf{0}$).

One useful derivable rule (rule iG) says the value of term f can be remembered in fresh ghost variable x:

$$\text{(iG)} \quad \frac{\Gamma, p : x = f \vdash M : \phi}{\Gamma \vdash \text{Ghost}[x = f](p.\ M) : \phi} \quad \text{where } x \text{ fresh except free in } M,\ p \text{ fresh}$$

Rule iG derives from the core proof terms (Def. A.1) using arithmetic and quantifiers:

$$\text{Ghost}[x = f](p.\ M) \equiv (\lambda x : \mathbb{Q}.\ (\lambda p : (x = f).\ M))\ f\ (\mathsf{FO}[f = f]())$$

Closure Ordinals and Loop Expressiveness. The verification of loops in game logics deserves special attention because loops in game logic are known (Platzer, 2015a) to have a higher expressive power than they do in other dynamic logics. Formally speaking, every loop in a dynamic logic or game logic can be assigned a *closure ordinal*, the number of iterations after which the fixed-point construction of its (usually backward-chaining (Platzer, 2015a)) semantics converges. More expressive logics can reason about loops with higher closure ordinals. Loops in non-game dynamic logics like dL have closure ordinals of at most ω, while loops in game logics can have much higher closure ordinals, such as the Church-Kleene ordinal (Platzer, 2015a). Thus, the verification of game loops places greater demands on the loop convergence rule $\langle * \rangle$I than does verification of non-game loops. Loops with closure ordinal ω can be verified using scalar termination metrics, which is why it is sufficient for dynamic logic convergence rules to employ scalar termination metrics, such as in dL (Platzer, 2008a, Rule G4). In contrast, rule $\langle * \rangle$I must support larger closure ordinals in a game logic. The most direct way to support large closure ordinals is to support arbitrary well-founded relations or, in CGL, effectively-well-founded relations. Because we wish to extract for-loops from convergence proofs (Chapter 8), we do not directly support arbitrary effectively-well-founded relations, but rather express those relations through well-founded termination metrics.

Simpler formulations of loop convergence for systems (Platzer, 2015a, 2012b) correspond to the scalar-valued termination metrics that decrease at a fixed rate after each iteration. For greater generality, the termination metric $\mathcal{M}$ of rule $\langle * \rangle$I need not be read as a single scalar variable. More general metrics such as lexicographic metrics are supported with a *virtual* reading which interprets $\varphi, \mathcal{M}_0 \succ \mathcal{M}$, and $\mathbf{0} \succcurlyeq \mathcal{M}$ as formulas over several scalar variables. Lexicographic metrics in particular are sometimes desirable in practical proofs. We conjecture that lexicographic metrics have a higher expressive power (and higher closure ordinals) than scalar metrics, because lexicographic metrics enable disjunctive and conjunctive combinations of scalar metrics. Intuitively, the virtual reading of termination metrics increases expressive power because it allows an arbitrary formula for the termination condition ($\mathbf{0} \succcurlyeq \mathcal{M}$), not only comparisons, let alone scalar comparisons. See the discussion of "clockwork" formulas in the literature (Platzer, 2015a, App.

C) for examples of properties which have high closure ordinals and thus might benefit from lexicographic termination metrics.

In practice, useful examples and case studies can be proved with scalar metrics, but non-scalar metrics are important for improving completeness. Even in our examples (Section 4.5), it will be important for soundness that scalar metrics allow inequalities rather than exact equality comparisons in the termination condition: a decreasing metric $\mathcal{M}$ suffices to show that the bound $\mathbf{0}$ is eventually crossed, but we cannot guarantee that the exact equality $\mathcal{M} = \mathbf{0}$ ever holds for a rational metric $\mathcal{M}$ without strong assumptions on the initial value $\mathcal{M}$.[7] Chapter 5 will use a very minor generalization of scalar metrics: in the context of constructive reals, it proves useful to extend termination metrics with a distinguished value for $\mathbf{0}$ indicating that the loop is ready to terminate. Thus, the type of a metric in Chapter 5 will not just be a scalar, but a disjoint union between a scalar and a distinguished (unit) value.

4.8 Theory: Soundness

The full CGL soundness proof with additional lemmas is given in the appendix (Appendix A.4). We have introduced a proof calculus for CGL which can prove winning strategies for Nim and cake-cutting. For any new proof calculus, it is essential to convince ourselves of soundness, which can be done within several prominent schools of thought. In proof-theoretic semantics, for example, the proof rules are taken as the ground truth, but are validated by showing the rules obey expected properties such as harmony or, for a sequent calculus, cut-elimination. While we will investigate proof terms separately (Section 4.10), we are already equipped to show soundness by direct appeal to the realizability semantics (Section 4.6), which we take as an independent notion of ground truth. We show soundness of CGL proof rules against the realizability semantics, i.e., that every provable natural-deduction sequent is valid. An advantage of this approach is that it explicitly connects the notions of provability and computability! We build up to the proof of soundness by proving lemmas on structurality, renaming and substitution.

Lemma 4.9 (Structurality)**.** *The structural rules W, X, and C are admissible, i.e., the conclusions are provable using existing rules whenever the premises are provable.*

$$\text{(W)}\quad \frac{\Gamma \vdash M : \phi}{\Gamma, p : \psi \vdash M : \phi} \qquad \text{(X)}\quad \frac{\Gamma, p : \phi, q : \psi \vdash M : \rho}{\Gamma, q : \psi, p : \phi \vdash M : \rho} \qquad \text{(C)}\quad \frac{\Gamma, p : \phi, q : \phi \vdash M : \rho}{\Gamma, p : \phi \vdash [p/q]M : \rho}$$

Proof summary. Each rule is proved admissible by induction on M. Observe that the only premises regarding Γ are of the form $\Gamma(p) = \phi$, which are preserved under rule W. Premises are trivially preserved under rule X because contexts are treated as sets, and preserved modulo renaming by contraction (rule C) as it suffices to have *any* assumption of a given formula, regardless of its name. The context Γ is allowed to vary in applications of the inductive hypothesis, e.g., in rules that bind program variables. Some rules discard Γ in checking the subterms inductively, in which case the IH need not be applied at all. □

[7] This is in contrast to integral for-loops which commonly increment a natural-number index until it is exactly equal to a bound.

Lemma 4.10 (Uniform renaming)**.** *Let $M\frac{y}{x}$ be the renaming of program variable x to y (and vice-versa by transposition) within M, even if x and y are not fresh. If $\Gamma \vdash M : \phi$ then $\Gamma\frac{y}{x} \vdash M\frac{y}{x} : \phi\frac{y}{x}$.*

Proof summary. Straightforward induction on the structure of M. Renaming within proof terms (whose definition we omit as it is quite tedious) follows the usual homomorphisms, from which the inductive cases follow. In the case that M is a proof variable z, then $\left(\Gamma\frac{y}{x}\right)(z) = \Gamma(z)\frac{y}{x}$ from which the case follows. The interesting cases are those which modify program variables, e.g., $\langle z := f\frac{w}{z} \text{ in } p.\ M\rangle$. The bound variable z is renamed to $z\frac{y}{x}$. If the renaming of x and y causes auxiliary variable w to no longer be fresh, it is renamed to a new fresh variable. Renaming is applied recursively in M. □

Substitution will use proofs of coincidence and bound effect lemmas. The lemmas are phrased in terms of free variables ($\mathrm{FV}(e)$) of expression e and bound and must-bound variables ($\mathrm{BV}(\alpha)$ or $\mathrm{MBV}(\alpha)$) of a game α (Appendix A.4). The definitions of free, bound, and must-bound variables, which agree with the definitions from dGL, are listed in Appendix A.4 for reference. As always, free variables are those which may influence the meaning of an expression, bound variables are those which are written on at least one execution path, and must-bound variables are those which are written on every execution path. We write $\omega = \tilde{\omega}$ on V for states $\omega, \tilde{\omega}$ and for $V \subseteq \mathcal{V}$ when $\omega(x) = \tilde{\omega}(x)$ for all $x \in V$. For proper regions X and Y, we write $X = Y$ on V when for all $(\mathsf{b}, \omega) \in X$ there exists $\tilde{\omega}$ where $\omega = \tilde{\omega}$ on V such that $(\mathsf{b}, \tilde{\omega}) \in Y$ and vice versa. For (not necessarily proper) regions X and Y, we write $X = Y$ on V iff $X \setminus \{\top, \bot\} = Y \setminus \{\top, \bot\}$ on V and $X \cap \{\top, \bot\} = Y \cap \{\top, \bot\}$.

Lemma 4.11 (Coincidence)**.** *Only the free variables of an expression and of its realizer influence its semantics. That is, assume $\omega = \tilde{\omega}$ on $V \supseteq \mathrm{FV}(e)$ (where e is f or ϕ or α or b). In the claims for projections, games, and formulas, additionally assume $V \supseteq \mathrm{FV}(\mathsf{b})$. Recall the free variables $\mathrm{FV}(\mathsf{b})$ of realizer b are those which appear syntactically free at any point in b, not just components which get used in the current state. Then:*

- $[\![f]\!]\omega = [\![f]\!]\tilde{\omega}$
- $[\![\mathsf{b}]\!]\omega = [\![\mathsf{b}]\!]\tilde{\omega}$
- $\{(\mathsf{b}, \omega)\}_{[0]} \overset{V}{=} \{(\mathsf{b}, \tilde{\omega})\}_{[0]}$ *and* $\{(\mathsf{b}, \omega)\}_{[1]} \overset{V}{=} \{(\mathsf{b}, \tilde{\omega})\}_{[1]}$
- $\{(\mathsf{b}, \omega)\}_{\langle 0\rangle} \overset{V}{=} \{(\mathsf{b}, \tilde{\omega})\}_{\langle 0\rangle}$ *and* $\{(\mathsf{b}, \omega)\}_{\langle 1\rangle} \overset{V}{=} \{(\mathsf{b}, \tilde{\omega})\}_{\langle 1\rangle}$
- $\{(\mathsf{b}, \omega)\}\langle\!\langle \alpha \rangle\!\rangle = \{(\mathsf{b}, \tilde{\omega})\}\langle\!\langle \alpha \rangle\!\rangle$ *on* $\mathrm{MBV}(\alpha) \cup V$
- $\{(\mathsf{b}, \omega)\}[\![\alpha]\!] = \{(\mathsf{b}, \tilde{\omega})\}[\![\alpha]\!]$ *on* $\mathrm{MBV}(\alpha) \cup V$
- $(\mathsf{b}, \omega) \in [\![\phi]\!]$ *iff* $(\mathsf{b}, \tilde{\omega}) \in [\![\phi]\!]$

Proof summary. By induction on the expression, in analogy to (Platzer, 2017a). □

Note that the simplicity of the statements of the coincidence claims for games is a result of the fact that the CGL semantics are forward chaining, by comparison to backward-chaining semantics such as those used in dGL (Platzer, 2015a).

Lemma 4.12 (Bound effect)**.** *Only the bound variables of a game are modified by execution. Let $X = \{(\mathsf{b}, \omega)\}$. Let $(\mathsf{c}, \nu) \in X[\![\alpha]\!]$ or $(\mathsf{c}, \nu) \in X\langle\!\langle \alpha \rangle\!\rangle$. Then $\omega = \nu$ on $\mathrm{BV}(\alpha)^{\complement}$, the complement of set $\mathrm{BV}(\alpha)$.*

Proof summary. The proof is by induction on the expression. It is analogous to bound effect for dL (Platzer, 2017a). □

Definition 4.14 (Term substitution admissibility)**.** A program variable substitution σ is admissible for expression e (likewise for proof terms M and contexts Γ) if e does not bind any variable x substituted by σ (any $x \in Dom(\sigma)$), nor mention that x free under any binder that binds any free variable of $\sigma(x)$.

While the assumption that $x \in Dom(\sigma)$ are not bound in ϕ may appear to be a strong assumption, it is a reasonable assumption in the context of CGL, where substitutions are often applied to ghost variables. Moreover, the assumption can be met in practice by α-renaming any binders of x and introducing stuttering assignments $x := x$, if needed. Aside from the restriction on binding x in ϕ, our notion of admissibility is analogous to existing admissibility notions for dL (Platzer, 2008a, Def. 6).

Lemma 4.13 (Arithmetic-term substitution)**.** *If $\Gamma \vdash M : \phi$ and the program variable substitution σ is admissible for Γ, M, and ϕ, then $\sigma(\Gamma) \vdash \sigma(M) : \sigma(\phi)$.*

Proof summary. By induction on M. Admissibility holds recursively, and so can be assumed at each step of the induction. For non-atomic M that bind no variables, the proof follows from the inductive hypotheses. For M that bind variables, we appeal to Lemma 4.11 and Lemma 4.12. □

Just as arithmetic terms are substituted for program variables, proof terms are substituted for proof variables.

Lemma 4.14 (Proof term substitution)**.** *Let $[N/p]M$ by the result of (proof term) substitution of proof term N for p in M. Proof term substitution implicitly renames proof variables if necessary to avoid (proof) variable capture. If $\Gamma, p : \psi \vdash M : \phi$ and $\Gamma \vdash N : \psi$ then $\Gamma \vdash [N/p]M : \phi$.*

Proof. By induction on M. When substituting N for p into a term that binds program variables such as $\langle z := f\frac{y}{z} \text{ in } q.\ M\rangle$, we avoid capture by renaming within occurrences of N in the recursive call, that is we rename $[N/p]\langle z := f\frac{y}{z} \text{ in } q.\ M\rangle = \langle z := f\frac{y}{z} \text{ in } q.\ [N\frac{z}{y}/p]M\rangle$, preserving soundness by Lemma 4.10. □

Soundness of the proof calculus exploits renaming and substitution.

Theorem 4.15 (Soundness of proof calculus)**.** *If the proof judgement $\Gamma \vdash M : \phi$ holds then the CGL natural deduction sequent $(\Gamma \vdash \phi)$ is valid. As a special case for empty context $\cdot$, if $\cdot \vdash M : \phi$, then ϕ is valid.*

Proof summary. By induction on M. Modus ponens case $(A\ B)$ reduces to Lemma 4.14. Cases that bind program variables, e.g., assignment, hold by Lemma 4.13 and Lemma 4.10. Rule W is employed when substituting under a binder. □

We have now shown that the CGL proof calculus is sound, the *sine qua non* condition of any proof system. Because soundness was w.r.t. a realizability semantics, we have shown CGL is constructive in the sense that provable formulas correspond to realizable strategies, i.e., imperative programs executed in an adversarial environment. We will

revisit constructivity again in Section 4.10 when we develop the theory of proof terms as *functional* programs.

4.9 Operational Semantics

The Curry-Howard interpretation of games is not complete without exploring the interpretation of proof simplification as normalization of functional programs. To this end, we now introduce a structural operational semantics for CGL proof terms. This semantics provides a view complementary to the realizability semantics: not only do provable formulas correspond to realizers, but proof terms can be directly executed as functional programs, resulting in a *normal* proof term. The chief subtlety of our operational semantics is that in contrast to realizer execution, proof simplification is a static operation, so it does not inspect game state. Thus, the normal form of a proof which branches on the game state is, of necessity, also a proof which branches on the game state. This static-dynamic phase separation need not be mysterious: the difference between proof term simplification and game execution is analogous to the monadic phase separation between a functional program which returns an imperative command vs. the execution of the returned command. While the theoretical motivation for our operational semantics is to complete the Curry-Howard interpretation of CGL, proof normalization is also of practical use when implementing software tools which process proof artifacts, since code that consumes a normal proof is often easier to implement than code that consumes an arbitrary proof.

The operational semantics consist of two main judgments: M normal says that M is in normal form, while $M \mapsto M'$ says that M reduces to term M' in one step of evaluation. The top-level connective of a normal proof must be either an introduction form (in which case the normal form is also a canonical form) or a case operation $\langle \text{case } A \text{ of } \ell \Rightarrow B \mid r \Rightarrow C \rangle$ over an Angelic choice proof A (as opposed to case analysis on an Angelic loop proof). Nested top-level cases are permitted as well, as are cases which are not at top-level. Normal proofs M without state-casing are called *simple*, written M simp. The requirement that cases are top-level (when possible to soundly lift them there) ensures that proofs which differ only in where the case was applied share a common normal form, and ensures that β-reduction is never blocked by a case interceding between introduction-elimination pairs. Top-level case analyses are analogous to case-tree normal forms in lambda calculi with coproducts (Altenkirch, Dybjer, Hofmann, & Scott, 2001). Proof term reduction is eager.

Definition 4.15 (Normal forms)**.** We say M is *simple*, written M simp, if proof terms of elimination rules occur only under a binding (position within a) proof term. We say M is *normal*, written M normal, if M simp or M has shape $\langle \text{case } A \text{ of } \ell \Rightarrow B \mid r \Rightarrow C \rangle$ where A is a term such as (split $[f \sim g]$) that inspects the state to prove a disjunction or other Angelic choice (in contrast to case analysis over Angelic loops). Subterms B and C need not be normal since they occur under the binding of ℓ or r.

That is, a normal term has no top-level beta-redexes, and state-dependent cases are top-level. We consider rules $[*]$R, $[:*]$I, $[?]$I, and $\langle\!:=\!\rangle$I binding. The rule $\langle * \rangle$I has multiple premises but only the inductive step and postcondition step are binding. While rule $[*]$R does not introduce a proof variable, it is still considered binding to prevent divergence,

which is in keeping with a coinductive reading of formula $[\alpha^*]\phi$. If we did not care whether terms diverge, we could make rule $[*]$R non-binding.

We first give the β-rules (Fig. 4.5), then structural and commuting-conversion rules, as well as what we call *monotonicity conversion* rules: a proof term $M \circ_p N$ is simplified by structural recursion on M. The capture-avoiding substitution of M for p in N is written $[M/p]N$ (Lemma 4.14).

$$(\lambda\phi\beta)\quad (\lambda p : \phi.\ M)\ N \mapsto [N/p]M \qquad\qquad (\pi_L\beta)\quad \langle\!|\pi_L \langle\!|M, N|\!\rangle|\!\rangle \mapsto M$$

$$(\lambda\beta)\quad (\lambda x : \mathbb{Q}.\ M)\ f \mapsto M^f_x \qquad\qquad (\pi_R\beta)\quad \langle\!|\pi_R \langle\!|M, N|\!\rangle|\!\rangle \mapsto N$$

$$(\text{case}\beta\text{L})\quad \langle\!|\text{case } \langle\!|\ell \cdot A|\!\rangle \text{ of } p \Rightarrow\ B \mid q \Rightarrow\ C|\!\rangle \mapsto [A/p]B$$

$$(\text{case}\beta\text{R})\quad \langle\!|\text{case } \langle\!|r \cdot A|\!\rangle \text{ of } p \Rightarrow\ B \mid q \Rightarrow\ C|\!\rangle \mapsto [A/q]C$$

$$(\text{unroll}\beta)\quad [\text{unroll } [\text{roll } M]] \mapsto M$$

$$(\text{FO}\forall\beta)\quad \mathsf{FO}[\forall x\, \phi](M) \mapsto (\lambda x : \mathbb{Q}.\ \mathsf{FO}[\phi](M))$$

$$(\text{FO}\wedge\beta)\quad \mathsf{FO}[\phi \wedge \psi](M) \mapsto \langle\!|\mathsf{FO}[\phi](M), \mathsf{FO}[\psi](M)|\!\rangle$$

$$(\text{FO}\exists\beta)\quad \mathsf{FO}[\exists x\, \phi](M) \mapsto \langle f\tfrac{y}{x} :* p.\ \mathsf{FO}[\phi]([\mathsf{FO}[f\tfrac{y}{x} = f\tfrac{y}{x}]()/p](M_x^{\left(f\frac{y}{x}\right)}))\rangle$$

$$(\text{FO}\vee\beta)\quad \mathsf{FO}[\phi \vee \psi](M) \mapsto \langle \text{case } \pi_0\mathsf{b} \text{ of } p \Rightarrow \langle \ell \cdot \mathsf{FO}[\phi](M, p)\rangle \mid q \Rightarrow \langle r \cdot \mathsf{FO}[\psi](M, q)\rangle\rangle$$ [1]

$$(\text{unpack}\beta)\quad \text{unpack}(\langle f\tfrac{y}{x} :* q.\ M\rangle, py.\ N) \mapsto (\text{Ghost}[x = f\tfrac{y}{x}](q.\ [M/p]N))$$

$$(\text{FP}\beta)\quad FP(A, s.\ B, g.\ C) \mapsto \langle \text{case}_*\ A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow [(r \circ_t FP(t, s.\ B, g.\ C))/g]C\rangle$$

$$(\text{rep}\beta)\quad (M \text{ rep } p : \psi.\ N \text{ in } O) \mapsto [\text{roll } \langle M, ([M/p]N) \circ_q (q \text{ rep } p : \psi.\ N \text{ in } O)\rangle]$$

$$(\text{for}\beta)\quad \text{for}(p : \varphi(\mathcal{M}) = A; q; B)\ \{\alpha\}\ C \mapsto$$

$$\begin{array}{l}\langle \text{case split } [\mathcal{M} \sim 0] \text{ of}\\ \quad \ell \Rightarrow\ \langle \text{stop } [(A, \ell)/(p, q)]C\rangle\\ \mid r \Rightarrow\ \text{Ghost}[\mathcal{M}_0 = \mathcal{M}](rr.\ \langle \text{go } (([A, \langle rr, r\rangle/p, q]B) \circ_t (\text{for}(p : \varphi(\mathcal{M}) = \langle \pi_L t\rangle; q; B)\ \{\alpha\}C))\rangle)\rangle\end{array}$$

[1] Here, b is the realizer corresponding to M. Recall that rule FO is non-effective by design, hence the reflection of semantic constructs (b) into a syntactic rule.

Figure 4.5: Operational semantics: β-rules.

The propositional rules $\lambda\phi\beta$, $\lambda\beta$, caseβL, caseβR, $\pi_L\beta$, and $\pi_R\beta$ are standard reductions for applications, cases, and projections. Projection terms $\langle\!|\pi_L M|\!\rangle$ and $\langle\!|\pi_R M|\!\rangle$ should not be confused with projection realizers $\pi_0\mathsf{b}$ and $\pi_1\mathsf{b}$. Rule unpackβ makes the witness of an existential available in its client as a ghost variable.

Rules FPβ, repβ, and forβ reduce introductions and eliminations of loops. Rule FPβ,

which reduces a proof $FP(A, s.\ B, g.\ C)$, says that if α^* has already terminated according to A, then B proves the postcondition. Else the inductive step C applies, but every reference to the IH g is transformed to a recursive application of FP. If A uses only rules $\langle * \rangle$S and $\langle * \rangle$G, then $FP(A, s.\ B, g.\ C)$ reduces to a simple term, else if A uses rule $\langle * \rangle$I, then $FP(A, s.\ B, g.\ C)$ reduces to a case. Rule repβ says loop invariant proof term $(M \text{ rep } p : \psi.\ N \text{ in } O)$ reduces to a delayed pair of the "stop" and "go" cases, where the "go" case first shows $[\alpha]\psi$, for loop invariant ψ, then expands $\psi \to [\alpha^*]\phi$ in the postcondition. Note the treatment of the [roll] proof term as binding (i.e., its laziness) is essential for normalization: when $(M \text{ rep } p : \psi.\ N \text{ in } O)$ is understood as a coinductive proof, it is clear that normalization would diverge if rule repβ were applied indefinitely. Rule forβ with term $\text{for}(p : \varphi(\mathcal{M}) = A; q; B)\,\{\alpha\}\,C$ checks whether the termination metric $\mathcal{M}$ has reached terminal value $\mathbf{0}$, which is indicated by proof term split $[\mathcal{M} \sim 0]$ whose notation $\mathcal{M} \sim 0$ is mnemonic for the use of $\sim$ to range over the set of binary comparison operators in Section 4.4. If so, the loop $\langle \text{stop} \rangle$'s and A proves it has converged. Else, we remember $\mathcal{M}$'s value in a ghost term $\mathcal{M}_0$, and $\langle \text{go} \rangle$ forward, using A and $\langle rr, r \rangle$ to satisfy the preconditions of inductive step B, then run the loop $\text{for}(p : \varphi(\mathcal{M}) = \langle \pi_L t \rangle; q; B)\,\{\alpha\}\,C$ in the postcondition. Rule forβ reflects the fact that the number of iterations is state dependent.

We now list monotonicity, commuting conversion, and structural rules. Monotonicity rules are listed in Fig. 4.6. Commuting conversion rules are listed in Fig. 4.7 and Fig. 4.8. Structural rules are listed in Fig. 4.9. Operational rules are also listed again in Appendix A.1. While the operational rules are many[8], there is much similarity between the rules, especially among the commuting conversion, monotonicity, and structural rules.

When reading Fig. 4.6, recall that the proof term for the monotonicity rule M is written $M \circ_p N$, where M and N are the premises of the rule and p is the proof variable with which the context is extended in the checking of N. The monotonicity rules show how normalization of monotonicity proofs can reduce monotonicity to other proof rules. The monotonicity rules almost amount to a proof that the monotonicity rule is admissible, except that the loop monotonicity rule $[*]\circ$ works by loop unrolling and in the process produces a monotonicity instance $(t \circ_p N)$ which is no simpler than the input. Thus, applications of rule $[*]\circ$ would never suffice to completely eliminate monotonicity proof steps that are applied to loops. In principle, it should be possible to define monotonicity rules which rely on induction and invariant arguments and which in doing so would allow eagerly, entirely eliminating monotonicity proofs and thus amount to a proof of admissibility.

The monotonicity rules for atomic games identify the subterm that proves the original postcondition, then perform substitutions to produce a proof of the new postcondition. The monotonicity rules for composite game inductively apply monotonicity on simpler subgames: for example, the rule $[\iota]\circ$ for sequential composition $\alpha; \beta$ applies nested monotonicity arguments, first on α, then on β. Substitutions such as $\cdot^{\vec{a}}_{\vec{a}'}$ are performed in rules stop$\circ$, $[\cup]\circ$, and $[*]\circ$. The reason for the substitutions is subtle: in checking the second premise of a monotonicity rule application, the context contains ghosts for the original values of bound variables of the game. In a recursive call, monotonicity may be applied to

[8]There are 15 β rules, 17 monotonicity rules, 25 commuting conversion rules, and 22 structural rules, for a total of 79 operational rules.

$$(\lambda\circ)\quad (\lambda x:\mathbb{Q}.\ M)\circ_q N \mapsto (\lambda x:\mathbb{Q}.\ ([M/q]N)) \qquad ([\iota]\circ)\quad [\iota\ M]\circ_p N \mapsto [\iota\ (M\circ_q(q\circ_p N))]$$

$$(\lambda\phi\circ)\quad (\lambda p:\phi.\ M)\circ_q N \mapsto (\lambda p:\phi.\ ([M/q]N)) \qquad (\langle\iota\rangle\circ)\quad \langle\iota\ M\rangle\circ_p N \mapsto \langle\iota\ (M\circ_q(q\circ_p N))\rangle$$

$$([^d]\circ)\quad [\text{yield } M]\circ_p N \mapsto [\text{yield } (M\circ_p N)] \qquad (\langle\ell\cdot\rangle\circ)\quad \langle\ell\cdot M\rangle\circ_p N \mapsto \langle\ell\cdot(M\circ_p N)\rangle$$

$$(\langle^d\rangle\circ)\quad \langle\text{yield } M\rangle\circ_p N \mapsto \langle\text{yield } (M\circ_p N)\rangle \qquad (\langle r\cdot\rangle\circ)\quad \langle r\cdot M\rangle\circ_p N \mapsto \langle r\cdot(M\circ_p N)\rangle$$

$$([\cup]\circ)\quad [A,B]\circ_q N \mapsto [A\circ_p(N^{\vec{a}}_{\vec{a}'}), B\circ_p(N^{\vec{b}}_{\vec{b}'})] \qquad (\langle?\rangle\circ)\quad \langle A,B\rangle\circ_p N \mapsto \langle A,[B/p]N\rangle$$

$$(\langle{:}*\rangle\circ)\quad \langle f\tfrac{y}{x} :* q.\ M\rangle\circ_p N \mapsto \langle f\tfrac{y}{x} :* q.\ [M/p]N\rangle$$

$$(\langle{:}{=}\rangle\circ)\quad \langle x := f\tfrac{y}{x} \text{ in } q.\ M\rangle\circ_p N \mapsto \langle x := f\tfrac{y}{x} \text{ in } q.\ [M/p]N\rangle$$

$$([{:}{=}]\circ)\quad [x := f\tfrac{y}{x} \text{ in } q.\ M]\circ_p N \mapsto [x := f\tfrac{y}{x} \text{ in } q.\ [M/p]N]$$

$$(\langle\!|\text{case}|\!\rangle\circ)\quad \langle\!|\text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C|\!\rangle\circ_p D \mapsto \langle\!|\text{case } A \text{ of } \ell \Rightarrow\ B\circ_p D \mid r \Rightarrow\ C\circ_p D|\!\rangle$$

$$([*]\circ)\quad [\text{roll } M]\circ_p N \mapsto [\text{roll } \langle([\pi_L M])\circ_p(N^{\vec{a}}_{\vec{a}'}), ([\pi_R M])\circ_t(t\circ_p N)\rangle]$$

$$(\text{stop}\circ)\quad \langle\text{stop } M\rangle\circ_p N \mapsto \langle\text{stop } ([M/p](N^{\vec{a}}_{\vec{a}'}))\rangle$$

$$(\text{go}\circ)\quad \langle\text{go } M\rangle\circ_p N \mapsto \langle\text{go } (M\circ_q q\circ_p N)\rangle$$

Figure 4.6: Operational semantics: monotonicity rules.

a simpler game, which contain fewer bound variables and thus fewer ghosts. The renamed proof terms are substituted in order to ensure they continue to be proofs in a context which contains fewer ghosts. In rule $[*]\circ$ for game α^*, for example, $\vec{a}$ is a vector containing exactly $\mathrm{BV}(\alpha)$ and $\vec{a}'$ is a vector of corresponding ghost variables. The substitution in rule $[*]\circ$ effectively undoes the ghosting operation. The same meaning applies in rule stop$\circ$. In rule $[\cup]\circ$ for game $\alpha\cup\beta$, the vectors $\vec{a}$ and $\vec{b}$ do not contain *all* variables of $\mathrm{BV}(\alpha)$ and $\mathrm{BV}(\beta)$, they contain variables which are exclusive to the opposing branch, i.e., $\mathrm{BV}(\beta)\setminus\mathrm{BV}(\alpha)$ and $\mathrm{BV}(\alpha)\setminus\mathrm{BV}(\beta)$, respectively. Vectors $\vec{a}'$ and $\vec{b}'$ are again vectors of ghost variables corresponding to $\vec{a}$ and $\vec{b}$.

The commuting conversion rules describe how case expressions can be lifted to the top level starting from any location except under a binder. The structural rules enable normalizing subterms in context, so long as the context does not bind. For constructs with multiple subterms that are not under binders, the structural rules require left-to-right normalization, so that the leftmost term is fully normalized before beginning normalization of the term to its right.

$$(\llbracket\pi_L\rrbracket\mathrm{C}) \quad \llbracket\pi_L\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rrbracket \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \llbracket\pi_L B\rrbracket \mid r \Rightarrow\ \llbracket\pi_L C\rrbracket\rangle$$

$$(\llbracket\pi_R\rrbracket\mathrm{C}) \quad \llbracket\pi_R\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rrbracket \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \llbracket\pi_R B\rrbracket \mid r \Rightarrow\ \llbracket\pi_R C\rrbracket\rangle$$

$$([\cup]\mathrm{C1}) \quad [\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle, N] \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ [B, N] \mid r \Rightarrow\ [C, N]\rangle$$

$$([\cup]\mathrm{C2}) \quad [M, \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle] \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ [M, B] \mid r \Rightarrow\ [M, C]\rangle$$

$$(\langle ?\rangle\mathrm{C1}) \quad \langle\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle, N\rangle \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \langle B, N\rangle \mid r \Rightarrow\ \langle C, N\rangle\rangle$$

$$(\langle ?\rangle\mathrm{C2}) \quad \langle M, \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rangle \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \langle M, B\rangle \mid r \Rightarrow\ \langle M, C\rangle\rangle$$

$$\begin{aligned}(\text{stopC}) \quad & \langle \text{stop } \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rangle \\ \mapsto & \langle \text{case } A \text{ of } \ell \Rightarrow\ \langle \text{stop } B\rangle \mid r \Rightarrow\ \langle \text{stop } C\rangle\rangle\end{aligned}$$

$$(\text{goC}) \quad \langle \text{go } \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rangle \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \langle \text{go } B\rangle \mid r \Rightarrow\ \langle \text{go } C\rangle\rangle$$

$$(\langle \ell\cdot\rangle\mathrm{C}) \quad \langle \ell \cdot \langle \text{case } A \text{ of } p \Rightarrow\ B \mid q \Rightarrow\ C\rangle\rangle \mapsto \langle \text{case } A \text{ of } p \Rightarrow\ \langle \ell \cdot B\rangle \mid q \Rightarrow\ \langle \ell \cdot C\rangle\rangle$$

$$(\langle r\cdot\rangle\mathrm{C}) \quad \langle r \cdot \langle \text{case } A \text{ of } p \Rightarrow\ B \mid q \Rightarrow\ C\rangle\rangle \mapsto \langle \text{case } A \text{ of } p \Rightarrow\ \langle r \cdot B\rangle \mid q \Rightarrow\ \langle r \cdot C\rangle\rangle$$

$$\begin{aligned}(\text{caseC}^*) \quad & \langle \text{case}_* \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \text{ of } s \Rightarrow\ D \mid g \Rightarrow E\rangle \\ \mapsto & \langle \text{case } A \text{ of} \\ & \quad s \Rightarrow\ \langle \text{case}_* B \text{ of } s \Rightarrow\ D \mid g \Rightarrow E\rangle \\ & \mid g \Rightarrow\ \langle \text{case}_* C \text{ of } s \Rightarrow\ D \mid g \Rightarrow E\rangle\rangle\end{aligned}$$

$$\begin{aligned}(\text{caseC}) \quad & \langle \text{case } \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \text{ of } \ell \Rightarrow\ D \mid r \Rightarrow\ E\rangle \\ \mapsto & \langle \text{case } A \text{ of} \\ & \quad \ell \Rightarrow\ \langle \text{case } B \text{ of } \ell \Rightarrow\ D \mid r \Rightarrow\ E\rangle \\ & \mid r \Rightarrow\ \langle \text{case } C \text{ of } \ell \Rightarrow\ D \mid r \Rightarrow\ E\rangle\rangle\end{aligned}$$

Figure 4.7: Operational semantics: commuting conversion rules.

$$\text{(unrollC)}\quad \begin{array}{l}[\text{unroll } \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle] \\ \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ [\text{unroll } B] \mid r \Rightarrow\ [\text{unroll } C]\rangle\end{array}$$

$$\text{(repC)}\quad \begin{array}{l}\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \text{ rep } p : \psi.\ N \text{ in } O \\ \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ (B \text{ rep } p : \psi.\ N \text{ in } O) \mid r \Rightarrow\ (C \text{ rep } p : \psi.\ N \text{ in } O)\rangle\end{array}$$

$$\text{(forC)}\quad \begin{array}{l}\text{for}(p : \varphi(\mathcal{M}) = \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle; q; N)\,\{\alpha\}\,O \\ \mapsto \langle \text{case } A \text{ of} \\ \quad \ell \Rightarrow\ \text{for}(p : \varphi(\mathcal{M}) = B; q; N)\,\{\alpha\}\,O \\ \quad \mid r \Rightarrow\ \text{for}(p : \varphi(\mathcal{M}) = C; q; N)\,\{\alpha\}\,O\end{array}$$

$$\text{(FPC)}\quad \begin{array}{l}FP(\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle, s.\ D, g.\ E) \\ \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ FP(B, s.\ D, g.\ E) \mid r \Rightarrow\ FP(C, s.\ D, g.\ E)\rangle\end{array}$$

$$\text{(}\langle\iota\rangle\text{C)}\quad \langle \iota\ \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rangle \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \langle \iota\ B\rangle \mid r \Rightarrow\ \langle \iota\ C\rangle\rangle$$

$$\text{([}\iota\text{]C)}\quad [\iota\ \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle] \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ [\iota\ B] \mid r \Rightarrow\ [\iota\ C]\rangle$$

$$\text{(}\langle^d\rangle\text{C)}\quad \begin{array}{l}\langle \text{yield } \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rangle \\ \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \langle \text{yield } B\rangle \mid r \Rightarrow\ \langle \text{yield } C\rangle\rangle\end{array}$$

$$\text{([}^d\text{]C)}\quad \begin{array}{l}[\text{yield } \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle] \\ \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ [\text{yield } B] \mid r \Rightarrow\ [\text{yield } C]\rangle\end{array}$$

$$\text{(app1C)}\quad \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\ N \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ B\ N \mid r \Rightarrow\ C\ N\rangle$$

$$\text{(app2C)}\quad M\ \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ M\ B \mid r \Rightarrow\ M\ C\rangle$$

$$\text{([:}*\text{]C)}\quad \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\ f \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ B\ f \mid r \Rightarrow\ C\ f\rangle$$

$$\text{(}\circ\text{C)}\quad \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \circ_p N \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ (B \circ_p N) \mid r \Rightarrow\ (C \circ_p N)\rangle$$

$$\text{(}\langle:*\rangle\text{C)}\quad \begin{array}{l}\text{unpack}(\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle, py.\ N) \\ \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \text{unpack}(B, py.\ N) \mid r \Rightarrow\ \text{unpack}(C, py.\ N)\rangle\end{array}$$

Figure 4.8: Operational semantics: commuting conversion rules (contd.)

$$(\pi_1\text{S})\quad \frac{M \mapsto M'}{\langle\!|\pi_L M|\!\rangle \mapsto \langle\!|\pi_L M'|\!\rangle} \qquad (\circ\text{S})\quad \frac{M \mapsto M'}{M \circ_p N \mapsto M' \circ_p N}$$

$$(\pi_2\text{S})\quad \frac{M \mapsto M'}{\langle\!|\pi_R M|\!\rangle \mapsto \langle\!|\pi_R M'|\!\rangle} \qquad (\ell\cdot\text{S})\quad \frac{M \mapsto M'}{\langle\!|\ell \cdot M|\!\rangle \mapsto \langle\!|\ell \cdot M'|\!\rangle}$$

$$(\text{repS})\quad \frac{M \mapsto M'}{(M \text{ rep } p : \psi.\ N \text{ in } O) \mapsto (M' \text{ rep } p : \psi.\ N \text{ in } O)} \qquad (r\cdot\text{S})\quad \frac{M \mapsto M'}{\langle\!|r \cdot M|\!\rangle \mapsto \langle\!|r \cdot M'|\!\rangle}$$

$$(\text{unroll})\quad \frac{M \mapsto M'}{[\text{unroll } M] \mapsto [\text{unroll } M']} \qquad ([\cup]\text{S1})\quad \frac{M \mapsto M'}{[M, N] \mapsto [M', N]}$$

$$([:*]\text{S})\quad \frac{M \mapsto M'}{M\ f \mapsto M'\ f} \qquad (\langle?\rangle\text{S1})\quad \frac{M \mapsto M'}{\langle M, N\rangle \mapsto \langle M', N\rangle}$$

$$([\iota]\text{S})\quad \frac{M \mapsto M'}{[\iota\ M] \mapsto [\iota\ M']} \qquad ([\cup]\text{S2})\quad \frac{M \text{ normal} \quad N \mapsto N'}{[M, N] \mapsto [M, N']}$$

$$(\langle\iota\rangle\text{S})\quad \frac{M \mapsto M'}{\langle\iota\ M\rangle \mapsto \langle\iota\ M'\rangle} \qquad (\langle?\rangle\text{S2})\quad \frac{M \text{ normal} \quad N \mapsto N'}{\langle M, N\rangle \mapsto \langle M, N'\rangle}$$

$$([^d]\text{S})\quad \frac{M \mapsto M'}{[\text{yield } M] \mapsto [\text{yield } M']} \qquad (\text{appS1})\quad \frac{M \mapsto M'}{M\ N \mapsto M'\ N}$$

$$(\langle^d\rangle\text{S})\quad \frac{M \mapsto M'}{\langle\text{yield } M\rangle \mapsto \langle\text{yield } M'\rangle} \qquad (\text{appS2})\quad \frac{M \text{ normal} \quad N \mapsto N'}{M\ N \mapsto M\ N'}$$

$$(\text{forS})\quad \frac{A \mapsto A'}{\text{for}(p : \varphi(\mathcal{M}) = A; q; B)\ \{\alpha\}\ C \mapsto \text{for}(p : \varphi(\mathcal{M}) = A'; q; B)\ \{\alpha\}\ C}$$

$$(\text{FPS})\quad \frac{A \mapsto A'}{FP(A, s.\ B, g.\ C) \mapsto FP(A', s.\ B, g.\ C)}$$

$$(\text{caseS})\quad \frac{A \mapsto A'}{\langle\!|\text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C|\!\rangle \mapsto \langle\!|\text{case } A' \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C|\!\rangle}$$

$$(\langle:*\rangle\text{S})\quad \frac{M \mapsto M'}{\text{unpack}(M, py.\ N) \mapsto \text{unpack}(M', py.\ N)}$$

Figure 4.9: Operational semantics: structural rules.

4.10 Theory: Constructivity

We now complete the study of CGL's constructivity. We validate the operational semantics on proof terms by proving that progress and preservation hold, and thus the CGL proof calculus is sound as a type system for the functional programming language of CGL proof terms. The full proofs of properties stated in this section are given in Appendix A.4.

Lemma 4.16 (Progress)**.** *If $\cdot \vdash M : \phi$, then either M is normal or $M \mapsto M'$ for some M'.*

Proof summary. By induction on the proof term M. If M is an instance introduction rule, by the inductive hypotheses each subterm is a valid CGL proof of the respective premise.

If they are all simple, then M simp. Else if some subterm (not under a binder) steps, then M steps by a structural rule. Else some subterm is an irreducible case expression not under a binder, so it lifts by the commuting conversion rule. If M is an elimination rule, structural and commuting conversion rules are applied as above. Else by Def. 4.15 M is an elimination applied to an introduction and M reduces with a β-rule. Lastly, if M has form $A \circ_x B$ and A simp, then, by Def. 4.15, A is an introduction form, thus reduced by some monotonicity conversion rule. □

Lemma 4.17 (Preservation)**.** *Let $\mapsto^*$ be the reflexive, transitive closure of the $\mapsto$ relation. If $\cdot \vdash M : \phi$ and $M \mapsto^* M'$, then $\cdot \vdash M' : \phi$.*

Proof summary. Induct on the derivation $M \mapsto^* M'$, then induct on $M \mapsto M'$. The β cases follow by Lemma 4.14 (for base constructs), and Lemma 4.14 and Lemma 4.10 (for assignments). C-rules and $\circ$-rules lift across binders. By rule W, the lifting is sound. Soundness of each S-rule is direct by the IH. □

We gave two understandings of proofs in CGL: imperative strategies and as functional programs. We now give a final perspective: CGL proofs support synthesis in principle, one of our main motivations. In the development of a constructive logic, it is common to prove properties known as the Existence Property (EP) and Disjunction Property (DP) which justify synthesis (Degen & Werner, 2006) for existentials and disjunctions: whenever an existential or disjunction has a proof, one can compute an instance or disjunct that has a proof. The standard (strong) notion of DP simply cannot hold in CGL or in any constructive dynamic logic because different branches of the disjunction may hold in different states. Likewise, a strong EP would require a strong term language: if a proof of an existential performs branching and then uses different existential witnesses in each branch, the EP would only hold if the term language is powerful enough to express terms which branch in the same fashion. Because we have chosen a simple term language for CGL which does not feature conditionals, we should not expect a strong EP to hold.

However, we can and do state and prove weaker counterparts of the DP and EP which still serve to prove the idea that computable witnesses for disjunctions and existentials can be extracted from proofs. Thus, our weak DP and EP still serve as important evidence for the constructivity of CGL. Rather than extract a disjunct or instance that is *provable* in our proof calculus, we extract realizers which witness truth in our *semantics*. These semantic properties hold respectively because a semantic characterization allows us to characterize the idea that *some* disjunct of every provable disjunction is true in each state, and because the expressive realizer language used by our semantics is strong enough to describe existential witnesses. After we state and prove a weak DP and EP for CGL, we then introduce a (weak) Strategy Property, their counterpart for synthesizing strategies from game modalities.

Before giving a proof of the weak DP, we demonstrate an example where disjunction strategies can depend on the state, showing that the naïve DP does not hold.

Example 4.1 (Naïve DP). When $\Gamma \vdash M : \phi \vee \psi$ there need not be a proof term N such that $\Gamma \vdash N : \phi$ or $\Gamma \vdash N : \psi$ hold.

Consider $\phi \equiv x > 0$ and $\psi \equiv x < 1$. Then $\cdot \vdash \text{split}\,[x \sim 0]\,:(\phi \vee \psi)$, but neither $x < 1$ nor $x > 0$ is valid, let alone provable.

Instead, the following Weak DP holds.

Lemma 4.18 (Weak Disjunction Property)**.** *When $\Gamma \vdash M : \phi \vee \psi$ there exists a realizer c and a computable function f from states to Booleans s.t. for every ω and b such that $(\mathsf{b}, \omega) \in [\![\bigwedge \Gamma]\!]$, either $[\![f]\!]\omega = 0$ and $(\pi_0\mathsf{c}, \omega) \in [\![\phi]\!]$, else $[\![f]\!]\omega = 1$ and $(\pi_1\mathsf{c}, \omega) \in [\![\psi]\!]$.*

Because disjunctions are defined as Angelic choices, the Disjunction Property is a direct corollary of a corresponding property for Angelic choices. When $\Gamma \vdash M : \langle\alpha \cup \beta\rangle\phi$ there exists realizer c and computable f, s.t. for every ω and b such that $(\mathsf{b}, \omega) \in [\![\bigwedge \Gamma]\!]$, either $[\![f]\!]\omega = 0$ and $(\pi_0\mathsf{c}, \omega) \in [\![\langle\alpha\rangle\phi]\!]$, or else $[\![f]\!]\omega = 1$ and $(\pi_1\mathsf{c}, \omega) \in [\![\langle\beta\rangle\phi]\!]$.

Proof. We prove the property for Angelic choices. By Theorem 4.15, the sequent $\Gamma \vdash \langle\alpha \cup \beta\rangle\phi$ is valid. Since $(\mathsf{b}, \omega) \in [\![\bigwedge \Gamma]\!]$, then by the definition of sequent validity, there exists a common realizer d such that $(\mathsf{d}\,\mathsf{b}, \omega) \in [\![\langle\alpha \cup \beta\rangle\phi]\!]$. Now let $f = \pi_0(\mathsf{d}\,\mathsf{b})$ and $\mathsf{c} = \pi_1(\mathsf{d}\,\mathsf{b})$ and the result is immediate by the semantics of Angelic choices. The claim for disjunctions follows directly by the definition $\phi \vee \psi \equiv \langle ?\phi \cup ?\psi\rangle \mathit{true}$. □

In stating and proving the EP, special attention must be paid to the CGL term language, because the EP seeks to find a term which witnesses every existential. It turns out the witnesses of existentials sometimes require conditional branches, which are not, strictly speaking, part of the CGL term language. Because the syntactic CGL term language is not strong enough to express all existential witnesses, we prove a semantic version of EP instead. Throughout our discussion, recall that existentials are defined as Angelic nondeterministic assignments, so that when we speak of the semantics or proof rules for existentials, those for Angelic nondeterministic assignments apply.

The limited expressive power of the CGL term language raises an important related point in regards to completeness. Existentials in CGL (equivalently, Angelic nondeterministic assignments) are proved using rule $\langle{:}*\rangle$I, which requires a CGL term as a witness. Rule $\langle{:}*\rangle$I is patently incomplete with respect to the realizability semantics of CGL because the semantics permit any computable function as the witness of an existential, but the CGL term language does not even contain all first-order-definable terms, let alone all computable functions. It is thus worth noting that rule $\langle{:}*\rangle$I can be made more complete by extending the term language of CGL, because simple term languages like the one chosen in this chapter cannot express all witnesses permitted by the semantics. It is for this reason that Chapter 5 will directly use computable functions as its term language: strong term languages are important for expressive existentials.

We give an example existential formula that is provable even by CGL's existential rule, but whose witness is a conditional term, inexpressible in the CGL term language.

Example 4.2 (Rich terms help)*.* There are provable existential formulas over polynomial terms whose witnesses must be non-polynomial.

Let $\phi \equiv (x = y \wedge x > 0) \vee (x = -y \wedge x \leq 0)$. Then $f = |x|$ witnesses $\exists y : \mathbb{Q}\ \phi$. Formula ϕ is provable in CGL by casing on the sign of x and applying rule $\langle{:}*\rangle$I with witness $y = x$ on the branch where $x > 0$ holds and again with witness $y = -x$ in the branch where

$x \leq 0$ holds. In the semantics, the existential can easily be witnessed by the realizer $\texttt{if}\,(x \leq 0)\; -x\,\texttt{else}\,x$, which is another way of writing $|x|$.

Lemma 4.19 (Existence Property)**.** *If $\Gamma \vdash A : (\exists x : \mathbb{Q}\ \phi)$ in CGL then there exists a realizer c and (base) realizer f such that for all $(\mathsf{b}, \omega) \in [\![\bigwedge \Gamma]\!]$, we have $(\mathsf{c}\,\mathsf{b}, \omega[x \mapsto [\![f]\!]\omega]) \in [\![\phi]\!]$.*

Proof. By Theorem 4.15, the sequent $(\Gamma \vdash \exists x : \mathbb{Q}\ \phi)$ is valid. Since $(\mathsf{b}, \omega) \in [\![\bigwedge \Gamma]\!]$, then by the definition of sequent validity, there exists a common realizer d such that $(\mathsf{d}\,\mathsf{b}, \omega) \in [\![\exists x : \mathbb{Q}\ \phi]\!]$. Now let $f = \pi_0 \mathsf{d}\,\mathsf{b}$ and $\mathsf{c} = \pi_1 \mathsf{d}\,\mathsf{b}$ and the result is immediate by the semantics of existentials. □

Note that the conference version of this work (Bohrer & Platzer, 2020a) allows terms to be arbitrary computable functions of the state. While the conference version of the work still opts for a semantic EP because of its simple proof, a syntactic version of the EP is more likely to hold in the setting where all computable functions are permitted as terms as opposed to our present setting.

We now turn our attention to a Strategy Property which generalizes the idea of weak EP and DP to show that every provable game modality has a computable witness. Since realizers serve to witness computability of game strategies in CGL, the Strategy Property shows that all provable modalities have realizers which witness them.

Theorem 4.20 (Strategy Property for Angel's Turn)**.** *If $\Gamma \vdash M : \langle\alpha\rangle\phi$, then there exists a realizer c such that for all ω and realizers b such that $(\mathsf{b}, \omega) \in [\![\bigwedge \Gamma]\!]$, then we have that $\{(\mathsf{c}\,\mathsf{b}, \omega)\}\langle\!\langle\alpha\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\}$.*

Theorem 4.21 (Strategy Property for Demon's Turn)**.** *If $\Gamma \vdash M : [\alpha]\phi$, then there exists a realizer c such that for all ω and realizers b such that $(\mathsf{b}, \omega) \in [\![\bigwedge \Gamma]\!]$, then we have that $\{(\mathsf{c}\,\mathsf{b}, \omega)\}[\![\alpha]\!] \subseteq [\![\phi]\!] \cup \{\top\}$.*

Proof. Direct by Theorem 4.15 and the definition of the semantics of $\langle\alpha\rangle\phi$ and $[\alpha]\phi$ in each claim, respectively. □

The Strategy Property is a highly abstract notion of code extraction from game proofs because realizers are a highly abstract formal language. Throughout the thesis, computable winning strategies will be extracted from proofs in several ways which mirror our extraction of realizers. In Chapter 6, strategies of games are reified by defining a system which commits to the same choices (corr. same realizers for nondeterministic branching and assignments) as expressed in the proof. By thus reducing game verification to system verification, the Logician provides the Logic-User a gentle learning curve wherein well-established hybrid systems proof techniques can be applied to games. In Chapter 8, strategies are extracted to a program in a concrete executable intermediate representation which expresses the same computations that we express as realizers here. By providing this systematic treatment of program extraction, the Logician makes it easier to fulfill his promise to the Engineer of providing a robust synthesis tool implementation. Chapter 6 and Chapter 8 are specifically inspired by the case where Demon moves first (Theorem 4.21), but we present both cases for the sake of completeness.

While the proofs of the EP, DP, and Strategy Property are short and direct, we note that this is by design: the challenge in developing CGL is not so much the proofs of

this section, rather these proofs become simple because our realizability semantics was specifically designed to make them so. The challenge was in developing the semantics and adapting the proof calculus and theory to that semantics.

4.11 Summary

In this chapter, we developed a (discrete) Constructive Game Logic CGL, from syntax and realizability semantics to a proof calculus and operational semantics on the proof terms. We developed two understandings of proofs as programs: semantically, every proof of game winnability corresponds to a realizer which computes the game's winning strategy, while the language of proof terms is also a functional programming language where proofs reduce to their normal forms according to the operational semantics. Realizability semantics show how dynamic execution of game proofs can be implemented while the operational semantics identify the key fragment of normal proofs which must be handled when implementing any static transformation over proofs. We completed the Curry-Howard interpretation for games by showing (weak) Existence, Disjunction, and Strategy properties: programs can be synthesized that decide which instance, disjunct, or moves are taken in existentials, disjunctions, and games.

This chapter gave a generic presentation of a basic, discrete variant of CGL in part because the literature on constructive dynamic logics and their Curry-Howard interpretations is relatively sparse (Kamide, 2010). It is our hope that the results of this chapter are of use in other program logics, such as Concurrent DL and first-order DL, both fragments of GL. Existing applications of games and Concurrent DL are also suitable as applications of CGL, e.g., security games (Pauly & Parikh, 2003) and concurrent programs with Demonic schedulers. In principle, CGL and its fragments can be beneficial to any user of GL and DL, even if they wish to use existing classical proof calculi or software tools. Every constructive proof is also a classical proof, so the theorems we show about CGL could be applied in existing classical workflows and software tools by identifying which proofs were constructive post-hoc.

Both Chapter 5 and Part III build on this chapter. Chapter 5 will add hybrid game support and a new type-theoretic semantics which simplifies technical typing constraints from the realizer semantics, but is presented with less emphasis on generality. The synthesis algorithm of Chapter 8 will base its correctness argument on Theorem 4.20 and Theorem 4.21, whose constructive proofs are an on-paper synthesis algorithm.

Chapter 5

Constructive Differential Game Logic

5.1 Introduction

The language of hybrid games is a compelling formalism which extends hybrid systems with 2-player adversarial dynamics. Hybrid games provide an elegant framework for modeling and verifying CPSs which must operate correctly in adversarial environments. Many real environments are adversarial, and adversarial models are a natural modeling abstraction for worst-case analysis even of systems that are not truly adversarial. Even when hybrid systems models are possible, hybrid games can provide more concise models that are easier to get right, because games allow existential specifications of controllers which are more concise than the universal specifications (such as safe control envelope specifications) available in hybrid systems. As shown by the 2D driving case study for VeriPhy (Chapter 3), hybrid system models of safe control envelopes can grow surprisingly complex.

Hybrid systems and games are also well-positioned to benefit from constructive game proofs, especially for the purposes of correct code synthesis. Whereas classical VeriPhy enforced safety of blackbox controllers at the potential expense of liveness, in CdGL we wish to also support whitebox controllers whose simultaneous safety and liveness have proofs. Synthesis of whitebox controllers is more challenging than synthesis of blackbox controllers in the sense that a whitebox controller must *construct* a correct control decision rather than simply *check* whether a proposed decision is safe. In addition to safety, a proof of liveness is a natural goal because liveness enables functional guarantees that show a system eventually reaches some goal, but only a whitebox approach could hope to preserve liveness guarantees: a blackbox controller uses a non-live fallback controller whenever a monitor is violated, which could happen arbitrarily often. Hybrid game proofs excel at expressing control algorithms and monitors in the same proof, making them a promising input format for synthesis of whitebox controllers in Chapter 8. Constructivity ensures that the content of a hybrid game proof corresponds to executable code, i.e., it ensures that code can be extracted from *every* proof. Practical experience with the classical implementation of VeriPhy (Chapter 3) suggests that it is both important and nontrivial to develop an approach which can extract code from every (correct) proof. Constructivity for hybrid games in particular assists synthesis by providing constructive reasoning about

real numbers, which are a key ingredient of games. Constructive reasoning about reals is subtle because exact comparisons on reals are undecidable; by requiring constructive arguments for reals, we ensure that games' strategies only employ operations on reals for which implementation code can be correctly synthesized (see Section 5.4.1 for discussion of comparisons and see Chapter 8 for synthesis, including synthesis of comparisons). While other approaches are possible, such as attempting to automatically reconstruct executable content from a classical proof, the constructive approach promises it will never give us the unpleasant surprise that code reconstruction failed for a proof which we believed would correspond to code. By avoiding such surprises, we help the Logician fulfill his promise to the Engineer of providing a robust implementation of synthesis.

Because constructive games are a promising foundation for CPS verification and synthesis, this chapter extends CGL to Constructive Differential Game Logic (CdGL) for hybrid games. Compared to the rules and axioms of CGL, this chapter contributes constructive counterparts for the (differential equation) axioms of Differential Game Logic (dGL), which proves the *classical* existence of winning strategies hybrid game strategies (Platzer, 2015a).

This chapter also provides a new type-theoretic semantics for CdGL, as opposed to the realizability semantics of Chapter 4. Whereas realizers describe strategies as imperative programs, type-theoretic terms describe strategies as functional programs. In keeping with the desire for a general presentation, the realizer semantics of Chapter 4 are useful because imperative implementations are likely possible in a wide array of languages. A functional implementation of CdGL's type-theoretic semantics may not be natural in languages which lack functional features. By studying both styles of semantics, the thesis demonstrates how strategies might be implemented in each paradigm. Implementation aside, the type-theoretic semantics reuse typing rules from the host type theory in order to eliminate the subtle well-formedness conditions which were necessary in CGL's realizability semantics. A functional style also allows us to easily give a big-step operational semantics for CdGL strategies, which in turn serves as a clear outline for the functional implementation of strategy execution.

We study 1D driving control as an example, which demonstrates the strengths of both games and constructivity in a simple setting. Games and constructivity both introduce uncertainties: A player is uncertain how their opponent will play, while constructive real-number comparisons are never sure of exact equality. These uncertainties demand nuanced proof arguments, but these nuances improve our fidelity to real systems and improve synthesizability of code from strategies. Our example theorem is a reach-avoid property for hybrid games. Reach-avoid specifications are an expressive class of properties and have been studied in many contexts (Quesel & Platzer, 2012; Tomlin et al., 2000; Fan, Mathur, Mitra, & Viswanathan, 2018; Ding & Tomlin, 2010; Fisac, Chen, Tomlin, & Sastry, 2015; Z. Huang, Wang, Mitra, Dullerud, & Chaudhuri, 2015), but their deductive verification is under-explored (Quesel & Platzer, 2012; Z. Huang et al., 2015).

While 1D driving is a simple and well-studied setting that is likely familiar to some readers, it provides an opportunity to showcase the unfamiliar nuances of constructive, reach-avoid game proofs. Because reach-avoid properties subsume both safety and liveness, they will also be a focus of our synthesis work (Chapter 8).

While the motivations for CdGL are practical, practical benefits can only be validated

once foundations have been established. This chapter establishes foundations, not practical implementations. Practical benefits for the Logic-User will be demonstrated by the Kaisar proof language in Chapter 7 and practical benefits for the Engineer will be demonstrated by constructive VeriPhy in Chapter 8. Kaisar will draw inspiration for its structured language design from the CdGL natural deduction calculus, while constructive VeriPhy makes essential use of the structured Kaisar format in order to access the computational content of a proof.

5.2 Related Work

The related works for this chapter are also discussed in Chapters 1, 3, and 4, so we restrict this section to a brief discussion on the relationship of hybrid games and constructivity to synthesis as well as a discussion of the relationship between CdGL and discrete CGL.

Synthesis approaches can be divided into fully automated approaches which infer monitors or controllers given only a model, vs. approaches which require a (usually interactive) proof before monitors or controllers can be created.

Among the former class, we know of no automatic approach which supports broad classes of hybrid games as inputs, only simple classes of games such as initialized rectangular[1] hybrid games with deterministic jumps (Henzinger et al., 1999). In our approach, hybrid games (and their proofs) are the *input* of synthesis, which should not be confused with the use of games as an *intermediate* language in an existing well-known approach (Tomlin et al., 2000). Their approach synthesizes a controller for a hybrid *system* rather than hybrid *game*, but crucially generates a finite-alternation hybrid game as an intermediate step based on a temporal-logic style specification for control of the hybrid system. Their temporal-logic style specifications appear similar to game logic formulas in that each logic features both box and diamond modalities, but the modalities have different meanings and support different notions of liveness: liveness in their approach represents the notion that *every* system execution eventually satisfies a given property, while a game-logical diamond modality says there *exists* a strategy which eventually satisfies the property for all behaviors of the opponent. A strength of their approach is that they target synthesis in the fully-automatic sense, i.e., their synthesis procedure takes only a system and its specification as inputs, rather than a proof as we do.

It is unsurprising that games have received less attention than systems (Taly & Tiwari, 2010; Kloetzer & Belta, 2008; Finucane et al., 2010; Filippidis et al., 2016), because even hybrid system synthesis has only been proved complete when restricted to relatively simple classes such as triangular hybrid systems (Shakernia et al., 2001) and state-controllable[2] linear systems (Shakernia et al., 2000).

VeriPhy (Chapter 3) falls into the latter class of tools which exploit (interactive) proofs in addition to models. While synthesis and proof are both undecidable, interactive proof

[1]Rectangular games use constant bounds in guards and differential inequalities. An *initialized* rectangular hybrid game is one where every discrete transition binds every variable which is bound by the continuous dynamics of the destination.

[2]Meaning that the controller is able to reach each state as its output, starting from each initial state.

for undecidable logics is well-understood, and interactive proofs can be provided in practice for classes of systems where automated synthesis would be difficult. The underlying expressiveness of proofs is the unique strength of proof-driven synthesis: in principle, every provable system or game can be synthesized. However, both the classical implementation of VeriPhy and the ModelPlex (Mitsch & Platzer, 2016b, 2018; Bohrer et al., 2018) tactic used therein demonstrate a more difficult reality: synthesis tools can only make full use of proof insights if every proof corresponds to executable code and if the computational content of a proof is easily accessible to tools.

As Chapter 4 did for discrete CGL, this chapter shows that CdGL provides a constructive foundation that ensures proofs correspond to code. It does so for an expressive (source) language of first-order regular hybrid games, a language much broader than those addressed by previous hybrid synthesis approaches.

CdGL shares most of its rules with CGL or dGL, but that does not trivialize our developments. An entirely new semantics requires entirely new soundness proofs, where our soundness proofs for ODE rules mean that familiar proof rules work constructively, not just classically. We adapt dGL rules to use constructive reals (Bishop, 1967; Bridges & Vita, 2007) by appealing (on paper) to constructive formalizations of ODEs (Cruz-Filipe, Geuvers, & Wiedijk, 2004; Makarov & Spitters, 2013). Our proofs are done on paper rather than in a proof assistant because the inductive definitions in our semantics are outside the fragment supported by definition well-foundedness checks in prominent proof assistants like Coq (Section 5.3.2).

Because Coq rejects the definition we wish to write, it is natural to ask whether the rejection of the definitions suggests any unsoundness in our definition. It does not. There even exist Coq formalizations of the Knaster-Tarski theorem (e.g., as part of (Sterling & Harper, 2018)), which could be used to manually prove that our definition is well-founded. Thus, it is less accurate to say Coq cannot formalize our semantics, and more accurate to say that a Coq formalization would rely heavily on (non-standard) libraries, which is one reason the formalization is left as future work.

5.2.1 Syntax of CdGL

As in CGL, CdGL has three classes of expressions e: terms f, g, games α, β, and formulas ϕ, ψ, ρ. In contrast to Chapter 4, we do not exhaustively define a closed language of terms, but allow any constructive function from states to reals as a (scalar) term, that is we inherit the term language of the host type theory in which we will define our semantics. The choice to use an open-ended term language is not fundamental: the approach used here allows for a flexible term language while necessitating fewer proofs about term language semantics, while the approach of Chapter 4 provides a more exhaustive semantic development. The choice to treat terms abstractly is also a practical one: while our models and theorem statements only use well-known term constructs, our proofs occasionally (Section B.1.2) use conditional terms, whose foundations can only be done justice by a detailed treatment of so-called (strong) collection axioms and choice functions in constructive set theory (Aczel & Rathjen, 2001, §2.3, §7). Rather than reproduce prior work on constructive set theory, we have written the main text of the chapter to be understood without prior knowledge of

constructive set theory, but refer the reader to the literature (Aczel & Rathjen, 2001, §2.3, §7) if they seek a deeper understanding of proof steps involving conditional terms.

The terminology for players is the same as in Chapter 4. The players of the game are called Angel and Demon, where Angel always refers to the player we control and Demon always refers to the player we do not control. The diamond modality $\langle\alpha\rangle\phi$ considers the case when Angel is next-to-move and the box modality $[\alpha]\phi$ considers the case where Demon is next-to-move, but both modalities ask whether Angel can win game α with postcondition ϕ. We say "player" for the player who is next-to-move and "opponent" for the other. We use the adjective forms Angelic and Demonic to describe anything which has an existential or universal nature respectively, despite the fact that all strategies discussed are strategies for the Angel player.

Reals and real-valued functions are understood constructively *à la* Bishop (Bishop, 1967; Bridges & Vita, 2007). We use Bishop-style real analysis because it preserves many classical intuitions (e.g., uncountability) about $\mathbb{R}$ while ensuring computability. Our Type-2 (Weihrauch, 2000) computability requirement requires that all functions on real numbers are computable to arbitrary precision given the capacity to sample the input with arbitrary precision, yet the set of reals is also uncountable (Bishop & Bridges, 2012, Thm. 2.19). It is a theorem (Weihrauch, 2000) that all such computable functions are continuous, but not all computable functions need be differentiable nor satisfy stronger notions of continuity such as (any of the several existing notions of) Lipschitz continuity. Thus, we explicitly mention when terms need to be differentiable or satisfy additional, stronger notions of continuity. Whenever we write that a function is continuous without specifying some specific, stronger notion of continuity, we mean that it is continuous in the standard sense that all computable real functions are continuous. Below, we mention commonly-used terms, many of which are real-valued counterparts of operators in CGL.

Definition 5.1 (Terms)**.** A *term* f, g is any computable function over the game state. We list example term constructs that appear in this chapter; we use dots $(\cdots)$ to emphasize that we are listing examples rather than defining the syntax of a term language:

$$f, g \ ::= \ \cdots \mid c \mid x \mid f + g \mid f \cdot g \mid f/g \mid \min(f, g) \mid \max(f, g) \mid (f)'$$

where $c \in \mathbb{R}$ is a real literal, x a program variable, $f + g$ a sum, $f \cdot g$ a product, and f/g is real division of f by g. Divisors g are assumed to be nonzero. We assume the set $\mathcal{V}$ of variables is finite[3]. For every base program variable x there is a primed counterpart x' whose purpose within an ODE is to track the time derivative of x. Minimum and maximum of terms f and g are written $\min(f, g)$ and $\max(f, g)$. Any differentiable term f has a definable (Section 5.3.2) spatial differential term $(f)'$, which agrees with the time derivative within an ODE. The differential term $(f)'$ can *only* be used for differentiable f.

Real-valued terms f, g are simply type-2 computable functions, usually from states to reals. It is occasionally useful for f to return a tuple of reals, which is computable when

[3]In contrast to CGL, some of the notations used in our proofs assume there are finitely many variables. Notation aside, the actual proofs should generalize easily to the countable variable sets supported in CGL.

every component is computable. Since terms are functions, operators are combinators: $f + g$ is a function which sums the results of f and g.

In CdGL, we extend the game language of CGL with differential equations (ODEs).

Definition 5.2 (Games)**.** The language of *hybrid games* α, β contains all operators of first-order regular games (Chapter 4) and additionally ODEs:

$$\alpha, \beta \ ::= \ ?\phi \mid x := f \mid x := * \mid \alpha \cup \beta \mid \alpha; \beta \mid \alpha^* \mid \alpha^d \mid x' = f \,\&\, \psi$$

The ODE game $x' = f \,\&\, \psi$ evolves ODE $x' = f$ for duration $d \geq 0$ chosen by the player, who must prove that the domain constraint formula ψ is true throughout. We require that term f is *effectively-locally-Lipschitz* on domain ψ, meaning that at every state satisfying ψ, a neighborhood and coefficient L can be constructed such that it is constructively provable that L is a (local) Lipschitz constant of f in the neighborhood. Effective local Lipschitz continuity guarantees unique solutions constructively exist[4] by Constructive Picard-Lindelöf (Makarov & Spitters, 2013). ODEs are explicit-form, so no primed variable y' for $y \in \mathcal{V}$ is mentioned in f or ψ. Systems of ODEs are supported, but we present single equations for readability, except in situations where multiple equations are essential, such as the example model (Section 5.2.2) and rule DG. We parenthesize games with braces $\{\alpha\}$ when necessary. For convenience, we also write derived operators where the opponent is given control of a single choice before returning control to the player. The *dual choice* $\alpha \cap \beta$, defined $\{\alpha^d \cup \beta^d\}^d$, says the opponent chooses which branch to take, before returning control to the player in the subgame. *Dual repetition* $\alpha^\times$ is defined likewise by $\{\{\alpha^d\}^*\}^d$.

The formula connectives of CdGL are those of CGL. A CdGL context is a list of formulas $\Gamma = \psi_1, \ldots, \psi_n$. In contrast to Chapter 4, we do not give names to formulas in contexts, because we do not explicitly write down proof terms in which those names would be needed.

We write $\phi\frac{y}{x}$ (likewise for α and f) for the *renaming* of variable x for y and vice versa in formula ϕ, and write ϕ_x^f for the result of *substitution* of term f for program variable x in ϕ, if the substitution is admissible (Def. 5.7 in Section 5.5).

[4]The cited formalization assumes effective Lipschitz continuity (with a uniform effective Lipschitz constant across the given set, as in effective *global* Lipschitz continuity) on a compact domain, whereas we use effective *local* Lipschitz continuity on a domain that need not be bounded. In our setting, however, effective global Lipschitz continuity on all compact subsets of the state space, potentially with a different Lipschitz constant on each subset, is equivalent to effective local Lipschitz continuity on the state space because the state space is Euclidean, and thus a locally-compact metric space. It is intuitive that the two notions would agree because effective local Lipschitz continuity is a quantified statement over neighborhoods of states, and the closure of each such neighborhood is a compact subset of the state space. On the other hand, it is perhaps surprising to some readers that the two notions of continuity agree, because there exist ODEs, such as $x' = x^2$, which are effectively-locally-Lipschitz continuous, but do not have a solution on every compact subset of the state space, i.e., the existence interval is finite. There is no contradiction here: the cited formalization assumes, in addition to its continuity assumption, that the diameter of the compact set can be bounded below as a function of the effective Lipschitz constant, a condition which fails for many ODEs, such as those with finite existence intervals. Indeed, the main gap between the formalization and our usage of it is that our notion of effective local Lipschitz continuity drops the requirement of a lower bound on diameter.

5.2.2 Example Game

We give an example game and example theorem statements, proven in Appendix B.1. Automotive systems are a major class of CPS. As an example, we consider time-triggered 1-dimensional driving with adversarial timing. In general, 1-dimensional driving is a simple introductory topic, but the reach-avoid proof (Appendix B.1) for our 1-dimensional model illustrates the unique challenges and subtleties that are common among constructive reach-avoid proofs for games. Because the theorems in this section are stated as diamond modalities $\langle\alpha\rangle\phi$, Angel will be the first player to move and Demon will be the second player. In a box modality, Demon would move first and Angel would move second.

In this example, our adversarial model allows Demon to choose the duration of each loop iteration from an interval $[0, T]$, where T is an upper bound on the time between consecutive control cycles. We sometimes need to prohibit pathological "Zeno" behaviors where infinitely many iterations occur in finite time; in that case we restrict durations to $t \in [T/2, T]$, providing a lower bound that rules out pathological behaviors while still allowing adversarial dynamics. We write x for the current position of the car, v for its velocity, a for the acceleration, $A > 0$ for the maximum positive acceleration, and $B > 0$ for the maximum braking rate. We assume $x = v = 0$ initially to simplify arithmetic. Our model is time-triggered, with continuous notions of time and space. That is, the controller is guaranteed to run at least once every $T > 0$ time units, but safety is shown at all times as the system evolves continuously. Local clock t marks the current time within the current timestep, then resets at each step. The control game (ctrl) says Angel can pick any acceleration a that is physically achievable ($-B \leq a \leq A$). The clock t is then reinitialized to 0. The plant game (plant) says Demon can evolve physics for duration $t \in [0, T]$ such that $v \geq 0$ holds throughout, after which he returns control to Angel.

Typical theorems in DLs and GLs are *safety* and *liveness*: are unsafe states always avoided and are desirable goals eventually reached? Safety and liveness of a 1D driving *system* have been proved previously: safe driving (safety) never goes past goal g, while live driving eventually reaches g (liveness).

$$\begin{aligned}
\mathsf{pre} &\equiv T > 0 \land A > 0 \land B > 0 \land v = 0 \land x = 0 \qquad \mathsf{post} \equiv (g = x \land v = 0)\\
\mathsf{ctrl} &\equiv a := *;\ ?-B \leq a \leq A;\ t := 0\\
\mathsf{plant} &\equiv \{t' = 1, x' = v, v' = a \,\&\, t \leq T \land v \geq 0\}^d\\
\mathsf{safety} &\equiv \mathsf{pre} \to \langle(\mathsf{ctrl}; \mathsf{plant})^\times\rangle\, x \leq g\\
\mathsf{liveness} &\equiv \mathsf{pre} \to \langle(\mathsf{ctrl}; \mathsf{plant}; \{?t \geq T/2\}^d)^*\rangle\, x \geq g
\end{aligned}$$

The liveness theorem (liveness) requires a lower time bound ($\{?t \geq T/2\}^d$) to rule out Zeno strategies: Zeno strategies allow Demon to (for example) choose exponentially decreasing durations for each loop iteration, which would, in effect, stop time because the total elapsed duration after arbitrarily many iterations would have a finite upper bound. Such strategies typically invalidate liveness because typical liveness guarantees only hold if Angel is given sufficient time to reach the goal: after all, there exists some duration by which Angel has not yet reached the goal. If Demon forced time to stop by that point, liveness would not hold. Because Zeno behaviors are highly unrealistic, the standard (and justified) response

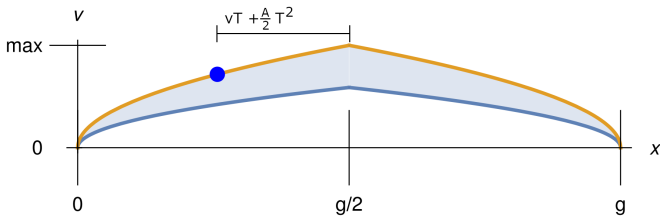


Figure 5.1: Safe driving envelope.

is to make them impossible rather than attempt to prove that they are live. The limit $t \geq T/2$ is chosen for simplicity. The safety theorem (safety) omits this limit because even Zeno behaviors satisfy the safety theorem, and a safety theorem which happens to include Zeno behaviors subsumes a safety theorem for the non-Zeno case.

Safety and liveness theorems, if designed carelessly, can have trivial solutions including but not limited to Zeno behaviors. It is safe to remain at $x = 0$ and is live to maintain $a = A$, but not vice-versa. In contrast to DLs, GLs easily express the requirement that *the same* strategy is both safe and live: we must remain safe *while* reaching the goal. We use this *reach-avoid* specification because it is immune to trivial solutions. We give a new reach-avoid result for 1D driving.

Example 5.1 (Reach-avoid). The following is provable in (dGL and) CdGL:

$$\mathsf{reachAvoid} \equiv \mathsf{pre} \rightarrow \langle \{\mathsf{ctrl}; \mathsf{plant}; ?x \leq g; \{?t \geq T/2\}^d\}^* \rangle \mathsf{post}$$

Angel *reaches* $g = x \wedge v = 0$ while safely *avoiding* states where $x \leq g$ does not hold. Angel is safe at *every* iteration for *every* time $t \in [0, T]$, thus safe *throughout* the game. The (dual) test $?t \geq T/2$ appears second, allowing Demon to win if Angel violates safety during $t < T/2$.

1D driving is well-studied for classical systems, but the constructive reach-avoid proof (Appendix B.1) is subtle because it must ensure progress in the presence of adversarial and constructive uncertainty, without compromising safety. The proof constructs an envelope of safe upper bounds and live lower bounds on velocity as a function of position (Fig. 5.1). The blue point indicates where Angel must begin to brake to ensure time-triggered safety, i.e., ensure she will remain safe even in the worst case where she must wait for T time before regaining control from Demon. It is surprising that Angel can achieve postcondition $g = x \wedge v = 0$, given that trichotomy ($f < g \vee f = g \vee f > g$) is constructively invalid. The key (Appendix B.1) is that comparison *terms* $\min(f, g)$ and $\max(f, g)$ *are* exact in Type 2 computability where it suffices to approximate min and max to arbitrary precision given the capacity to approximate the arguments to arbitrary precision. Our exact result encourages us that constructivity is not overly burdensome in practice. When decidable comparisons (formulas of shape $f \leq g + \varepsilon \vee f > g$ for some $\varepsilon > 0$, which can be proved by rule splitReal in Section 5.4) *are* needed, the alternative is a weaker guarantee $g - \varepsilon \leq x \leq g$ for some precision $\varepsilon > 0$. This relaxation is often enough to make the theorem provable, and reflects the fact that real agents only expect to reach their goal within finite precision.

While inexact comparisons are not needed in the acceleration controller for 1D driving, they are necessary in the loop convergence proof (rule $\langle * \rangle$I) to determine when the loop

should stop. In contrast to the exact rational comparisons of CGL, inexact comparisons introduce unique subtleties in termination checking, especially because the final iteration of a correct controller typically makes a small amount of progress as the system approaches a stop. The full proof overcomes (Section B.1.2) these subtleties with a non-trivial construction for the loop metric:

- Distance remaining to the goal is used as a termination metric, but a slight generalization of the scalar terms used in Chapter 5 is necessary. The termination bound $\mathbf{0}$ is now a distinguished value which is disequal from all real numbers, including the real number 0. Thus, the type of $\mathcal{M}$ a disjoint sum of a scalar type and unit type containing the single distinguished value $\mathbf{0}$. Intuitively, distinguished value $\mathbf{0}$ means that the remaining distance is *provably* zero, while real-number value 0 arises when the distance happens to be zero but has not be proven equal to 0. The system only terminates when the distance is provably 0, so it might (safely) run for an additional iteration when the metric evaluates to 0 rather than $\mathbf{0}$.

  The controller for the loop body must be designed with the possibility of an extra iteration in mind, i.e., an iteration starting at distance 0. In our example proof, the acceleration controller will not accelerate during the last iteration if the initial distance is 0, so that safety is preserved. While the final iteration will not physically progress toward the goal (because the system is already there), it will crucially make progress by *observing* that the goal has been reached, so that terminating the loop is a *provably* live decision.
- Because the special value $\mathbf{0}$ is only used when the distance is *provably* zero, the exact implementation of $\mathcal{M}$ relies on a case split. Case splits on constructive reals are nondeterministic in the sense that they do not uniquely specify the branch taken as a deterministic function of the state. As discussed in the proof, case-analysis terms are constructed using choice functions from constructive set theory (Aczel & Rathjen, 2001, §2.3, §7), whose strong collection axiom allows construction of piecewise, underspecified terms.

5.3 Type-theoretic Semantics

In this section, we define the semantics of hybrid games and game formulas in type theory. The type-theoretic semantics used in this chapter have the advantage that by reusing standard typing rules from the host theory, we can avoid the need to redevelop subtle well-formedness conditions from scratch. Both realizer semantics and type-theoretic semantics provide useful notions of strategy execution, but unlike the realizer semantics, the type-theoretic semantics give a functional notion of strategy execution which simplifies the development of a big-step operational semantics.

We start with assumptions on the underlying type theory.

5.3.1 Type Theory Assumptions

We assume a type theory with polymorphism and dependency in the style of the Calculus of Inductive and Coinductive Constructions (CIC) (Coquand & Huet, 1988; Coquand & Paulin, 1988; *Coq Proof Assistant*, 1989). We write M for terms and $\Delta \vdash M : \tau$ to say M has type τ in CIC context Δ. We assume first-class (indexed (Dybjer, 1994)) inductive and coinductive types. We write τ for type families and κ for kinds, which are type families inhabited by other type families. Inductive type families are written $\mu t : \kappa. \tau$, which denotes the *smallest* solution $\mathtt{ty}$ of kind κ to the fixed-point equation $\mathtt{ty} = \tau_t^{\mathtt{ty}}$ where $\tau_t^{\mathtt{ty}}$ denotes the result of substituting type (family) $\mathtt{ty}$ for type variable t in type (family) τ. Coinductive type families are written $\rho t : \kappa. \tau$, which denotes the *largest* solution $\mathtt{ty}$ of kind κ to the fixed-point equation $\mathtt{ty} = \tau_t^{\mathtt{ty}}$. Type-expression τ must be monotone in t so smallest and largest solutions exist by Knaster-Tarski (Harel et al., 2000, Thm. 1.12). Proof assistants like Coq reject definitions where monotonicity requires nontrivial proof; we did not mechanize our proofs because they use such definitions.

We use one predicative[5] universe which we write $\mathbb{T}$ and Coq writes `Type 0`. We use a predicative universe because consistency and computability are more obvious in a predicative setting, especially considering that we must also assume large elimination in our semantic definitions. We write $\Pi x : \tau_1. \tau_2$ for a dependent[6] function type with argument named x of type τ_1 and where return type τ_2 may mention x. We write $\Sigma x : \tau_1. \tau_2$ for a dependent pair type with left component named x of type τ_1 and right component of type τ_2, possibly mentioning x. These specialize to the simple function $\tau_1 \Rightarrow \tau_2$ and product types $\tau_1 * \tau_2$ respectively when x is not mentioned in τ_2. Lambdas $(\lambda x : \tau. M)$ inhabit dependent function types. Pairs (M, N) inhabit dependent pair types. Application is $M\ N$. Let-binding unpacks pairs, whose left and right projection are $\pi_0 M$ and $\pi_1 M$. We write $\tau_1 + \tau_2$ for a disjoint union inhabited by $\ell \cdot M$ and $r \cdot M$, and write $\mathsf{case}\ A\ \mathsf{of}\ \ell \Rightarrow B \mid r \Rightarrow C$ for its case analysis.

We assume a real number type $\mathbb{R}$ and a Euclidean state type $\mathfrak{S}$. The positive real numbers are written $\mathbb{R}_{>0}$, nonnegative reals $\mathbb{R}_{\geq 0}$. We assume scalar and vector sums, products, inverses, and units. States s, t support operations $s\ x$ and $\mathsf{set}\ s\ x\ v$ which respectively retrieve the value of variable x in $s : \mathfrak{S}$ or update it to v[7]. The usual axioms of setters and getters (J. N. Foster, 2010) are satisfied. We write $\mathfrak{s}$ for the distinguished variable of type $\mathfrak{S}$ representing the current state. We will find it useful to consider the semantics of an expression both at current state $\mathfrak{s}$ and at states s, t defined in terms of $\mathfrak{s}$ (e.g., $\mathsf{set}\ \mathfrak{s}\ x\ 5$).

[5] A universe τ is *impredicative* if it contains type families which quantify over τ, else it is *predicative*.

[6] Meaning that the return type can depend on a variable which stands for the value of the argument

[7] This chapter uses the Latin state names s and t rather than the Greek ω and ν of Chapter 4 for purely stylistic reasons: just as Chapter 2 uses Latin characters for object-logic rigid symbols, the names s and t remind us that our type-theoretic semantics definitions cannot just be read as semantic meta-logical statements but as syntactic object-logic statements in a type theory.

5.3.2 Semantics of CdGL

Terms f and g are type-theoretic functions of type $\mathfrak{S} \Rightarrow \mathbb{R}$. We will need differential terms $(f)'$, a definable term construct when f is differentiable. Not every term f need be differentiable, so we give a *virtual* definition, defining when $(f)'$ is equal to some term g. If $(f)'$ does not exist, then $(f)' = g$ is not provable. We define the (total) differential as the Euclidean dot product $(\cdot)$ of the gradient (variable name: ∇) with s', which is the vector of values $s\ x'$ assigned to primed variables x'. To show that ∇ is the gradient, we define the gradient as a limit, which we express in (ε, δ) style. In this definition, f and g are scalar-valued, and the minus symbol is used for both scalar and vector difference.

Definition 5.3 (Differential term semantics)**.** We virtually define the differential term $(f)'$ by defining when it equals a given term g in a given state s:

$$\begin{aligned}((f)'\ s = g\ s) \equiv\ & \Sigma\nabla : \mathbb{R}^{|s'|}.\, (g\ s = \nabla \cdot s') \star \Pi\varepsilon : \mathbb{R}_{>0}.\, \Sigma\delta : \mathbb{R}_{>0}.\, \Pi r : \mathfrak{S}. \\ & (\|r - s\| < \delta) \Rightarrow |f\ r - f\ s - \nabla \cdot (r - s)| \leq \varepsilon \|r - s\|\end{aligned}$$

For practical proofs, a library of standard rules for the automatic, syntactic differentiation of common arithmetic operations (Bohrer, Fernández, & Platzer, 2019) could be developed and their soundness could be proved.

The interpretation $\ulcorner \phi \urcorner : \mathfrak{S} \Rightarrow \mathbb{T}$ of formula ϕ is a predicate over states. A predicate of kind $\mathfrak{S} \Rightarrow \mathbb{T}$ is also understood as a *region*, e.g., $\ulcorner \phi \urcorner$ is the region containing states where ϕ is provable. A CdGL context Γ is interpreted over a uniform state term $s : \mathfrak{S}$ where $\mathfrak{s} : \mathfrak{S} \vdash s : \mathfrak{S}$, i.e., s usually mentions the distinguished state variable $\mathfrak{s}$. We define the notation $\ulcorner \Gamma \urcorner(s)$ to denote the CIC context containing $\mathfrak{s} : \mathfrak{S}$ and $\ulcorner \phi \urcorner\ s$ for each $\phi \in \Gamma$. The argument s in $\ulcorner \Gamma \urcorner(s)$ provides a convenient notation for describing the truth of the same context Γ across different states, in the same way that the notation $\ulcorner \phi \urcorner\ s$ allows describing the truth of formula ϕ in various states s. The sequent $(\Gamma \vdash \phi)$ is *valid* if there exists M where $\ulcorner \Gamma \urcorner(\mathfrak{s}) \vdash M : (\ulcorner \phi \urcorner\ \mathfrak{s})$. Formula ϕ is *valid* iff sequent $(\cdot \vdash \phi)$ is valid. That is, a valid formula is provable in every state with a *common* proof term M. While a proof term is allowed to inspect the state, the requirement for one common proof term implies that the proof term may only inspect the state constructively. For example, when proving a disjunctive postcondition $\phi \vee \psi$, a different approach may be used to prove each disjunct depending on the state, but the proof term would start by using a constructive case-analysis principle to decide which branch should be taken based on the state. Formula semantics employ the Angelic and Demonic semantics of games, which determine how to win a game α whose postcondition is ϕ. We write $\langle\!\langle \alpha \rangle\!\rangle : (\mathfrak{S} \Rightarrow \mathbb{T}) \Rightarrow (\mathfrak{S} \Rightarrow \mathbb{T})$ for the Angelic semantics of α and $[[\alpha]] : (\mathfrak{S} \Rightarrow \mathbb{T}) \Rightarrow (\mathfrak{S} \Rightarrow \mathbb{T})$ for its Demonic semantics. Angel's strategies for a hybrid game α with goal region $P : \mathfrak{S} \Rightarrow \mathbb{T}$ are inhabitants of $(\langle\!\langle \alpha \rangle\!\rangle\ P) : (\mathfrak{S} \Rightarrow \mathbb{T})$ and $([[\alpha]]\ P) : (\mathfrak{S} \Rightarrow \mathbb{T})$, respectively when Angel or Demon is next-to-move.

Definition 5.4 (Formula semantics)**.** We define the formula semantics by appealing to game and term semantics.

$$\begin{aligned}\ulcorner [\alpha]\phi \urcorner\ s &= [[\alpha]]\ \ulcorner \phi \urcorner\ s \\ \ulcorner \langle\alpha\rangle\phi \urcorner\ s &= \langle\!\langle \alpha \rangle\!\rangle\ \ulcorner \phi \urcorner\ s \\ \ulcorner f \sim g \urcorner\ s &= ((f\ s) \sim (g\ s))\end{aligned}$$

We write $[\alpha]$ or $\langle\alpha\rangle$ when we wish to discuss a game modality without specifying a postcondition. This notation is useful because Angelic strategies for diamond modalities differ from Demonic strategies for box modalities, regardless of the postcondition. Modal formula $\langle\alpha\rangle\phi$ is provable in s when $\langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\phi\urcorner\ s$ is inhabited so Angel has a $\langle\alpha\rangle$ strategy from s to reach region $\ulcorner\phi\urcorner$ on which ϕ is provable. Modality $[\alpha]\phi$ is provable in s when $[[\alpha]]\ \ulcorner\phi\urcorner\ s$ is inhabited so Angel has a $[\alpha]$ strategy from s to reach region $\ulcorner\phi\urcorner$ on which ϕ is provable. For $\sim \in \{\leq, <, =, \neq, >, \geq\}$, the values of f and g are compared at state s in $f \sim g$. The game and formula semantics are simultaneously inductive. In each case, the connectives which define $\langle\!\langle\alpha\rangle\!\rangle$ and $[[\alpha]]$ are duals, because $[\alpha]\phi$ and $\langle\alpha\rangle\phi$ are dual. Below, $P : (\mathfrak{S} \Rightarrow \mathbb{T})$ refers to the goal region of the game and $s : \mathfrak{S}$ to the initial state.

Definition 5.5 (Angelic semantics)**.** We define $\langle\!\langle\alpha\rangle\!\rangle : (\mathcal{S} \Rightarrow \mathbb{T}) \Rightarrow (\mathcal{S} \Rightarrow \mathbb{T})$ inductively (by a large elimination) on α:

$$\begin{aligned}
\langle\!\langle ?\psi\rangle\!\rangle\ P\ s &= \ulcorner\psi\urcorner\ s \star P\ s \\
\langle\!\langle x := f\rangle\!\rangle\ P\ s &= P\ (\mathsf{set}\ s\ x\ (f\ s)) \\
\langle\!\langle x := *\rangle\!\rangle\ P\ s &= \Sigma v : \mathbb{R}.\ P\ (\mathsf{set}\ s\ x\ v) \\
\langle\!\langle \alpha \cup \beta\rangle\!\rangle\ P\ s &= \langle\!\langle\alpha\rangle\!\rangle\ P\ s + \langle\!\langle\beta\rangle\!\rangle\ P\ s \\
\langle\!\langle \alpha;\beta\rangle\!\rangle\ P\ s &= \langle\!\langle\alpha\rangle\!\rangle\ (\langle\!\langle\beta\rangle\!\rangle\ P)\ s
\end{aligned}$$

$$\begin{aligned}
\langle\!\langle \alpha^d\rangle\!\rangle\ P\ s &= [[\alpha]]\ P\ s \\
\langle\!\langle x' = f\,\&\,\psi\rangle\!\rangle\ P\ s &= \Sigma d : \mathbb{R}_{\geq 0}.\, \Sigma sol : [0,d] \Rightarrow \mathbb{R}. \\
&\qquad (sol, s, d \vDash x' = f) \\
&\quad \star\ (\Pi t : [0,d].\ \ulcorner\psi\urcorner\ (\mathsf{set}\ s\ x\ (sol\ t))) \\
&\quad \star\ P\ (\mathsf{set}\ s\ (x, x') \\
&\qquad\qquad (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d))))
\end{aligned}$$

$$\langle\!\langle \alpha^* \rangle\!\rangle\ P\ s = \big(\mu\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\, \lambda t : \mathfrak{S}.\, (P\ t \Rightarrow \tau'\ t) + (\langle\!\langle\alpha\rangle\!\rangle\ \tau'\ t \Rightarrow \tau'\ t)\big)\ s$$

The Angelic semantics consider how Angel wins a game where she is currently in control. Angel wins $\langle ?\psi\rangle$ with goal region P by proving both ψ and P at s. Angel wins the deterministic assignment $\langle x := f\rangle$ by performing the assignment, then proving P. Angel wins nondeterministic assignment $\langle x := *\rangle$ by constructively choosing a value v to assign, then proving P. Angel wins $\langle \alpha \cup \beta\rangle$ by choosing between playing α or β, then winning that game. Angel wins $\langle \alpha;\beta\rangle$ if she wins α with the postcondition of winning β. Angel wins $\langle \alpha^d\rangle$ if she wins α when Demon is first-to-move in α.

We discuss the Angelic semantics of the ODE game $\langle x' = f\,\&\,\psi\rangle$ line-by-line. The first line of the Angelic ODE semantics ($\Sigma d : \mathbb{R}_{\geq 0}.\, \Sigma sol : [0,d] \Rightarrow \mathbb{R}.$) says that Angel must construct a duration $d : \mathbb{R}$ which she constructively proves to be nonnegative as well as function sol of type $[0,d] \Rightarrow \mathbb{R}$, which she will proceed to constructively prove to be a solution. Note that the variable sol stands for a function of the host type theory, all of which are a-priori computable and therefore continuous. In the next line, Angel constructively proves that sol is a solution by constructively proving the predicate $(sol, s, d \vDash x' = f)$ which we define[8] as:

$$(sol, s, d \vDash x' = f) \equiv \big((s\ x = sol\ 0) \star \Pi r : [0,d].\, ((sol)'\ r = f\ (\mathsf{set}\ s\ x\ (sol\ r)))\big)$$

The first conjunct of the definition of $(sol, s, d \vDash x' = f)$ says that the solution must agree with the initial state at time 0. The second conjunct says that at all times r in the domain of the solution function, the time-derivative $(sol)'$ of the solution function agrees

[8]For the sake of readability, we present the case where the ODE system is a singleton equation $x' = f$. It is implied that the definition generalizes to systems.

with the value of the right-hand side f in the state ($\mathsf{set}\ s\ x\ (sol\ r)$) which is the state of the system at time r. This is what it means to be the solution of an initial-value problem: the solution should have the required initial value and should make the ODE $x' = f$ a true equation. As in classical analysis, when $(sol, s, d \vDash x' = f)$ holds, then sol is also continuously differentiable. Note that $(sol, s, d \vDash x' = f)$ does not include our requirement that f is effectively-locally-Lipschitz, which is crucial for the constructive existence of unique solutions. Our syntax definition demands that we only ever write down an ODE if we already know the effectively-locally-Lipschitz requirement is satisfied, thus we do not repeat the requirement in the semantics.

The next line of the ODE semantics, $(\Pi t : [0, d].\ulcorner\psi\urcorner\ (\mathsf{set}\ s\ x\ (sol\ t)))$, says that Angel must be able to constructively prove the domain constraint ψ in each intermediate state ($\mathsf{set}\ s\ x\ (sol\ t)$) when given any time $t \in [0, d]$ by Demon. In the remaining lines, $P\ (\mathsf{set}\ s\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d))))$ says that Angel's proof of postcondition P in the final state ($\mathsf{set}\ s\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d))))$ must be constructive.

Note that the final state modifies both x and x'. While the postconditions of top-level theorem statements typically do not mention differentials, invariant-style proofs typically contain intermediate proof steps where postconditions do mention differentials, and for which it is essential that the semantics of ODEs reflect the differential of each variable in the final state.

The constructive Picard-Lindelöf (Makarov & Spitters, 2013) theorem constructs solutions for effectively-Lipschitz[9] ODEs, but the solution need not have a closed form. The proof calculus we introduce in Section 5.4 includes both solution-based proof rules and invariant-based rules. The solution-based proof rules syntactically replace an ODE with its solution in the special case where syntactic closed form does exist, while the invariant-based rules, including those which rely on Constructive Picard-Lindelöf, do not require the existence of *closed-form* solutions, but their soundness proofs do rely on the existence of some computable (not necessarily closed-form) solution. Solution-based rules can provide a particularly simple proof approach for simple ODEs with closed-form solutions, while invariant-style reasoning is crucial to support non-trivial ODEs which do not have simple closed forms.

[9]The cited paper formalizes a slightly different version of Constructive Picard-Lindelöf from what we assume. We assume effective *local* Lipschitz continuity over a domain which need not be bounded, whereas they assume effective Lipschitz continuity (with a uniform effective Lipschitz constant across the given set, as in effective *global* Lipschitz continuity) on some compact set. In our setting, however, effective global Lipschitz continuity on all compact subsets of the state space is equivalent to effective local Lipschitz continuity on the state space because the state space is Euclidean, and thus a locally-compact metric space. It is intuitive that the two notions would agree because effective local Lipschitz continuity is a quantified statement over neighborhoods of states, and the closure of each such neighborhood is a compact subset of the state space. On the other hand, some readers may be surprised that the two notions of continuity agree, because there exist ODEs, such as $x' = x^2$, which are effectively-locally-Lipschitz continuous, but do not have a solution on every compact subset of the state space, i.e., the existence interval is finite. There is no contradiction here: the cited formalization assumes, in addition to its continuity assumption, that the diameter of the compact set can be bounded below as a function of the effective Lipschitz constant, a condition which fails for many ODEs, such as those with finite existence intervals. Indeed, the main gap remaining between the formalization and our usage of it is that our notion of effective local Lipschitz continuity drops the requirement of a lower bound on diameter.

Angel strategies for $\langle\alpha^*\rangle$ are inductively defined: either choose to stop the loop and prove P now, else play a round of α before repeating inductively. By Knaster-Tarski (Harel et al., 2000, Thm. 1.12), this least fixed point exists because the interpretation of a game is monotone in its postcondition (Lemma 5.1). The existence of least fixed points is a fundamental tool for semantic proofs about loops (Section 5.5).

Lemma 5.1 (Monotonicity)**.** *Let $P, Q : \mathfrak{S} \to \mathbb{T}$ and $s : \mathfrak{S}$. Note that in this lemma, P and Q are* any *inhabitants of $\mathfrak{S} \to \mathbb{T}$, as opposed to only those of form $\ulcorner\phi\urcorner$. If $\Gamma, P\ s \vdash Q\ s$ is inhabited, then $\Gamma, \langle\!\langle\alpha\rangle\!\rangle\ P\ s \vdash \langle\!\langle\alpha\rangle\!\rangle\ Q\ s$ and $\Gamma, [[\alpha]]\ P\ s \vdash [[\alpha]]\ Q\ s$ are inhabited.*

Definition 5.6 (Demonic semantics)**.** We define $[[\alpha]] : (\mathcal{S} \Rightarrow \mathbb{T}) \Rightarrow (\mathcal{S} \Rightarrow \mathbb{T})$ inductively (by a large elimination) on α:

$$
\begin{aligned}
[[?\psi]]\ P\ s &= \ulcorner\psi\urcorner\ s \Rightarrow P\ s \\
[[x := f]]\ P\ s &= P\ (\mathsf{set}\ s\ x\ (f\ s)) \\
[[x := *]]\ P\ s &= \Pi v : \mathbb{R}.\ P\ (\mathsf{set}\ s\ x\ v) \\
[[\alpha \cup \beta]]\ P\ s &= [[\alpha]]\ P\ s\ \star\ [[\beta]]\ P\ s \\
[[\alpha; \beta]]\ P\ s &= [[\alpha]]\ ([[\beta]]\ P)\ s
\end{aligned}
$$

$$
\begin{aligned}
[[\alpha^d]]\ P\ s &= \langle\!\langle\alpha\rangle\!\rangle\ P\ s \\
[[x' = f\,\&\,\psi]]\ P\ s &= \Pi d : \mathbb{R}_{\geq 0}.\ \Pi sol : [0, d] \Rightarrow \mathbb{R}. \\
&\qquad (sol, s, d \vDash x' = f) \\
&\Rightarrow (\Pi t : [0, d].\ \ulcorner\psi\urcorner\ (\mathsf{set}\ s\ x\ (sol\ t))) \\
&\Rightarrow P\ (\mathsf{set}\ s\ (x, x') \\
&\qquad\qquad (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d))))
\end{aligned}
$$

$$
[[\alpha^*]]\ P\ s = \big(\rho\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}.\ (\tau'\ t \Rightarrow [[\alpha]]\ \tau'\ t)\ \star\ (\tau'\ t \Rightarrow P\ t)\big)\ s
$$

The Demonic semantics determine how Angel wins a game where Demon is currently in control. Angel wins $[?\psi]$ by proving postcondition P under assumption ψ, which Demon must provide (Section 5.6). Demonic deterministic assignment is identical to Angelic. Angel wins $[x := *]$ by proving P for *every* choice of x. Angel wins $[\alpha \cup \beta]$ with a pair of winning strategies for $[\alpha]$ and $[\beta]$ because Demon, not Angel, gets to choose which branch is played. Angel wins $[\alpha; \beta]$ by winning α with a goal region from which β is winnable. Angel wins $[\alpha^d]$ if she can win α when she is first-to-move in α.

We discuss the Demonic semantics of the ODE game $[x' = f\,\&\,\psi]$ line-by-line. The first line ($\Pi d : \mathbb{R}_{\geq 0}.\ \Pi sol : [0, d] \Rightarrow \mathbb{R}.$) says that Angel assumes she is given a duration $d \in \mathbb{R}_{\geq 0}$ and function ($sol : [0, d] \Rightarrow \mathbb{R}$) as arguments by Demon. As in the Angelic semantics, *sol* stands for a function of the host type theory, all of which are a-priori computable and therefore continuous. On the second line, Angel assumes that predicate $(sol, s, d \vDash x' = f)$ holds *constructively*. Also, because f is guaranteed to be effectively-locally-Lipschitz continuous by the syntactic constraints of ODEs, Constructive Picard-Lindelöf (Makarov & Spitters, 2013) can be applied, on any compact subset of the state space for which a uniform[10] (i.e., "global") effective Lipschitz constant exists constructively,

[10]In our setting, effective global Lipschitz continuity on all compact subsets of the state space is equivalent to effective local Lipschitz continuity on the state space because the state space is Euclidean, and thus a locally-compact metric space. It is intuitive that the two notions would agree because effective local Lipschitz continuity is a quantified statement over neighborhoods of states, and the closure of each such neighborhood is a compact subset of the state space. On the other hand, some readers are perhaps surprised that the two notions of continuity agree, because there exist ODEs, such as $x' = x^2$, which are effectively-locally-Lipschitz continuous, but do not have a solution on every compact subset of the state space, i.e., the existence interval is finite. There is no contradiction here: the cited formalization assumes, in addition to its continuity assumption, that the diameter of the compact set can be bounded below as

to show that the solution *sol* is unique and that Angel could construct it herself if she wished. Rather, we write *sol* as an assumption for symmetry with all the other assumptions and arguments which fundamentally must be provided by Demon. The third line ($\Pi t : [0,d].\ulcorner\psi\urcorner$ ($\mathsf{set}\ s\ x\ (sol\ t)$)) says that Angel receives a (constructive) assumption that for all $t : [0,d]$ the domain constraint ψ holds at time t (i.e, $\ulcorner\psi\urcorner$ ($\mathsf{set}\ s\ x\ (sol\ t)$)). That is, Demon is responsible for constructively proving the domain constraint for times t chosen by Angel up to time d chosen by Demon. The final lines (P ($\mathsf{set}\ s\ (x,x')\ (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d))))$) say that Angel is responsible for constructively proving the postcondition P in the final state $\mathsf{set}\ s\ (x,x')\ (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d)))$.

Angel wins $[\alpha^*]P$ if she can prove P no matter how many times Demon makes her play α. Angel's strategies for Demonic loops are coinductive using some invariant τ'. When Demon decides to stop the loop, Angel responds by proving P from τ'. Whenever Demon chooses to continue, Angel proves that τ' is preserved. Greatest fixed points exist by Knaster-Tarski (Harel et al., 2000, Thm. 1.12) using Lemma 5.1. The existence of greatest fixed points is a fundamental tool for semantic proofs about loops (Section 5.5).

It is worthwhile to discuss the relationship between the Angelic and Demonic semantics of each construct. We take the semantics of $x := *$ as an example. An Angelic strategy says how to compute x, a computation which is represented explicitly as an arithmetic term. A Demonic strategy simply accepts $x \in \mathbb{R}$ as its input where the set $\mathbb{R}$ is the uncountable set of all Bishop reals. That is, Demon can classically choose any real number when choosing x, even a number which is not computable by any term. In the Demonic case, Angel receives x as an input from classical Demon, but can only use computable operations on x in determining her own strategy for any following game statements.

The other connectives likewise make Angel resolve choices constructively but let Demon resolve choices classically. In choices and loops, for example, Angel's branching decisions must be computable, but Demon's branching choices may employ classical reasoning. An Angelic ODE allows Angel to constructively choose the duration of the ODE but requires her in return to constructively prove the ODE at all times up to the duration. A Demonic ODE lets Demon classically choose a duration up to which Demon *constructively* proves the domain constraint. As ODEs show, even Demon must use constructive reasoning at times, however. With domain constraints as well as tests, truth of a formula is always considered constructively regardless of player despite Demon's freedom to resolve *strategic choices* classically. Constructive truth is important because of proof-relevance: Angel's own strategy is allowed to branch on *how* Demon proved a given formula.

Because all strategies discussed in this chapter are for Angel, each strategy is constructive but permits Demon to resolve strategic choices classically. In the cyber-physical setting, the opponent is indeed rarely just a computer.

a function of the effective Lipschitz constant, a condition which fails for many ODEs, such as those with finite existence intervals. Indeed, the main gap remaining between the formalization and our usage of it is that our notion of effective local Lipschitz continuity drops the requirement of a lower bound on diameter.

5.3.3 Connecting CdGL to dGL

Having introduced the semantics of CdGL, the relationship between the meanings of CdGL and (classical) dGL formulas is a natural topic of discussion. In particular, we are now equipped to assess when a valid CdGL or dGL formula is, or is not, a valid formula of the other logic.

As a general rule, constructive logics have fewer valid formulas than their classical counterparts. For a canonical example, the law of the excluded middle ($\phi \vee \neg\phi$) is typically a sound axiom schema for classical logics, but not constructive ones. To find valid formulas of dGL which are invalid in CdGL, one need not even consider game modalities, because the arithmetic fragment of dGL already contains such formulas. For example, the disjunction $x \leq 0 \vee x > 0$ is classically valid but constructively invalid. For a fragment of arithmetic where classical and constructive truth *do* agree, see Section 7.4.1 in Chapter 7. Because disjunctions in CdGL are a special case of Angelic choices, there automatically exist examples of Angelic choice formulas which are valid in dGL but not CdGL. The previous example is equivalent to the formula $\langle ?x \leq 0 \cup ?x > 0\rangle \mathit{true}$ which is valid in dGL and invalid in CdGL. As discussed in Section 5.4.1, the distinction between classical and constructive Angelic choices has practical impact: classical choices require that Angel writes a program to decide which branch she takes, whereas classical choice does not. In both theory and practice, it is common for strategies of Angelic choices in dGL to branch on undecidable arithmetic properties that cannot be implemented as programs, thus it is important that CdGL added a restriction against such strategies.

The natural next question is whether, conversely, every valid formula of CdGL is a valid formula of dGL. Typically, the valid formulas of a constructive logic are a subset of the valid formulas of the corresponding classical logic. Thus, we expect that every valid formula of CdGL is valid in dGL, though we have not attempted a proof. However, there a moment of special attention is merited by the subtle differences between CdGL and dGL, particularly their different notions of continuity. Because one could theoretically contrive[11] ODEs which are (classically) locally-Lipschitz continuous but not effectively so, there exist ODEs $x' = f$ such that (for example) $[x' = f]\mathit{true}$ is a valid formula of dGL and $[x' = f]\mathit{true}$. However, this does not invalidate our conjecture that all valid CdGL formulas are valid dGL formulas. Rather, $[x' = f]\mathit{true}$ would not be a (well-formed) formula of CdGL *at all*, because CdGL assumes effective local Lipschitz continuity in its *syntax*. We believe that this technical detail is sufficient to ensure valid CdGL formulas are also valid in dGL: formulas of CdGL only discuss effectively-locally-Lipschitz continuous ODEs, which already well-behaved enough that unique solutions exist constructively, thus they are (classically) locally-Lipschitz continuous and have unique solutions in a classical setting as well.

If we extended dGL and CdGL to respectively allow ODEs that are not (classically) locally-Lipschitz continuous and not effectively-locally-Lipschitz, the relationship between the hypothetical extensions of dGL and CdGL would become more subtle. Without assuming the respective notions of Lipschitz continuity, the uniqueness of ODE solutions cannot be

[11]For example, one could write an ODE which contains a conditional term that conditions on a property that is true, but undecidable, and which simplifies to a locally-Lipschitz ODE in the branch where the undecidable property is true.

guaranteed, thus the modality $[x' = f \,\&\, \psi]\phi$ takes on a particularly subtle meaning in each logic: postcondition ϕ must hold for *all* solutions of $x' = f \,\&\, \psi$. This presents a problem in combination with the fact that the semantics of ODEs in CdGL quantifies over *computable* solution functions, while the semantics of ODEs in dGL quantifies over *classical* functions. There exist [Thm. 2](Aberth, 1971) computable ODEs α whose only solutions are uncomputable functions. For such ODEs α and for any CdGL formula ϕ, the semantics of formula $[\alpha]\phi$ vacuously quantifiers over the empty set of computable solutions, whereas the corresponding dGL semantics of formula $[\alpha]\phi$ would non-vacuously quantify over the set of all solutions, which is non-empty because it contains the uncomputable solution. In this contrived, albeit interesting, case, we expect that validity in (an extension of) CdGL would not imply validity in (an extension of) dGL, because the CdGL semantics of $[\alpha]\phi$ are vacuous and the dGL semantics are not.

If one wished to translate CdGL results into dGL even when both logics have been extended with exotic ODEs, there are several possible approaches. One possible approach would be to redefine the semantics of Demonic ODEs to require the existence of a computable solution, so that $[\alpha]\phi$ is unprovable, rather than vacuous, in CdGL when α is an ODE whose solutions are all uncomputable. Just as dL-style ODE proof rules can be generalized to ODEs with non-unique solutions (Bohrer & Platzer, 2020b), we hypothesize that the ODE proof rules of CdGL could also be adapted to a Demonic ODE modality that requires the existence of at least one solution. An alternate approach would be to translate CdGL *proofs* into dGL instead of relating the semantics of CdGL and dGL. Because no CdGL proof rule for ODEs has weaker assumptions than its dGL counterpart, a translation of CdGL proofs into dGL should be possible even in the presence of exotic ODEs.

On a related note, one may wonder whether it is essential that CdGL requires *effective* local Lipschitz continuity when dGL does not require effectiveness. Surprisingly, the solution of every locally-Lipschitz function remains computable even without the effectiveness assumption, indeed solutions are computable whenever they are unique (P. Collins & Graça, 2008). Thus, if one is only interested in computing the solutions of ODEs, effectiveness is not required. However, we have still chosen to employ *effective* local Lipschitz continuity in the semantics of CdGL in order to maintain a closer connection with the formalized proof of Constructive Picard-Lindelöf (Makarov & Spitters, 2013). Because some proofs about the semantics of CdGL appeal to Constructive Picard-Lindelöf, we expect it would be difficult to adapt those proofs to any setting which lacks the assumption of effective local Lipschitz continuity. Notably, we are not aware of any prior attempt to formalize a correctness proof for the algorithm (P. Collins & Graça, 2008) that computes solutions in the more general setting where effective local Lipschitz continuity is not assumed.

Lastly, because the major difference between dGL and CdGL is the difference between classicality and constructivity, one might wonder whether classical and constructive reasoning can be combined in CdGL, as well as which uses of constructivity in CdGL are truly essential. In constructive logics generally, classical truth of a formula ϕ can be expressed using standard double-negation translations (Avigad & Feferman, 1998, pg. 342) which compute for each ϕ a corresponding formula $\hat{\phi}$ which is constructively true exactly when ϕ is classically true. While it is not clear whether such a translation exists for modal formulas of ODEs, the standard translations for first-order logic could certainly be applied to the

first-order arithmetic fragment of CdGL.

One reason we did not pursue the translation of classical formulas into CdGL is because such translations may present additional conceptual complexity to the Logic-User: double-negation translations increase formula complexity. Syntactic sugar might partially alleviate the complexity of using the translation: just as CdGL makes widespread use of modal operators, classicality could be displayed to a Logic-User as if it were simply another operator on formulas. However, syntactic sugar would still not eliminate the mental burden of remembering which formulas are classical, which are constructive, and how the two kinds of formula interact. Thus, we have not pursued an approach which mixes classical and constructive reasoning.

Nonetheless, an approach which mixes classical and constructive reasoning would have its own benefits. Constructivity in CdGL serves to ensure that synthesis is possible. It stands to reason that constructivity is inessential in formulas whose proofs never influence execution, i.e., those for which proof relevance is never used. In practice, such formulas occur frequently. For example, it is common to prove that an arithmetic formula holds as an invariant of a game or assume its truth in a test without ever branching on a proof of the invariant. In such cases, it would be useful to make classical formulas available and immediately exploit the availability of classical decision procedures (G. E. Collins & Hong, 1991). As we will discuss in Section 7.4.1, our use of constructive arithmetic will require a more careful and limited use of such procedures.

5.4 Proof Calculus

We use a natural-deduction system for syntactic CdGL proofs. The CdGL proof system includes all rules of CGL (Chapter 4) except its arithmetic rules (e.g., rule split), so this section focuses on the differences between the two, such as the addition of constructive counterparts to the differential equation proof rules of dGL. We also discuss the arithmetic CGL rule, (split), which CdGL must change for the sake of soundness. In contrast to rational comparisons, exact real-number comparisons are not decidable, so CdGL provides an inexact comparison rule (splitReal) instead of the exact rational comparison rule split of CGL (Chapter 4).

The high degree of textual similarity between the CdGL proof calculus and those for CGL or dGL does not mean that the development of the CdGL proof calculus is trivial. The soundness proof for CdGL (Theorem 5.9) uses its new type-theoretic semantics, meaning the soundness proofs are entirely new compared to Chapter 4. Likewise, the similarity between the CdGL and dGL rules for ODEs is a testament to the fact that constructive analysis supports familiar reasoning principles despite a difference in mathematical foundations. The fact that familiar principles are available constructively is a fact worth knowing, because it implies that a Logic-User familiar with classical hybrid systems proofs will have a gentle learning curve in Chapter 7 when we develop a Logic-User-facing CdGL proof tool.

The difference between the ODE fragments of CdGL and dGL is so minor as to be easily forgotten: we require *effective* local Lipschitz continuity rather than (classical) local Lipschitz continuity. We have yet to encounter any ODE of practical interest which is

locally-Lipschitz continuous (in the classical sense) without being effectively so, yet the effectiveness assumption is of theoretical importance as a precondition to Constructive Picard-Lindelöf, which needs explicit knowledge of a function's Lipschitz constant in order to provide an explicit bound on how quickly the construction of an ODE's solution converges to the true solution.

The main difference between dGL and CdGL (Section 5.3.3) is the same as the difference between discrete GL and CGL: case analysis must be decidable and well-foundedness arguments for loops use effective well-foundedness. In practical use, the new challenges presented by CdGL vs. CGL are those of constructive real *arithmetic* rather than constructive real *analysis*: for example, exact comparisons on constructive reals are not decidable, whereas exact comparisons on the rational numbers of CGL are.

When Γ is a CdGL context, we write $(\Gamma \vdash \phi)$ for the natural-deduction sequent with conclusion ϕ and context Γ. We write $\Gamma^{\frac{y}{x}}$ for the renaming of program variable x to y and vice versa in context Γ. Likewise Γ^f_x is the substitution of term f for program variable x. As usual, we write $\mathrm{FV}(e)$, $\mathrm{BV}(\alpha)$, and $\mathrm{MBV}(\alpha)$ for the free variables of expression e, bound variables of game α, and must-bound variables of game α respectively, i.e., variables which *might* influence the meaning of an expression, might be modified during game execution, or are written during *every* execution. They are defined as in CGL, with the definitions for ODEs following those from classical dGL.

5.4.1 First-order Arithmetic Proofs

Like any first-order program logic, CdGL proofs contain first-order reasoning at the leaves. Decidability of constructive first-order real arithmetic is an open problem (Lombardi, 2020) in stark contrast to its decidable classical counterpart (Tarski, 1951), so first-order facts are proven manually in practice. Our semantics embed CdGL into type theory; we defer first-order arithmetic proving to the host theory for the sake of this chapter. In Chapter 7, we identify a fragment of first-order arithmetic where constructive and classical truth coincide, allowing classical tools to be applied, thus reducing the arithmetic proof burden for the Logic-User when compared to paper CdGL proofs. A major difference between classical and constructive real arithmetic is that constructive arithmetic only allows inexact comparisons, not exact comparisons, because exact comparisons over reals are not decidable (Bishop, 1967), but inexact comparisons up to some $\varepsilon > 0$ with formula shape $f \leq g + \varepsilon \vee f > g$ are decidable (Bridges & Vita, 2007), captured in rule splitReal. Note that it is also sound to conclude the seemingly-stronger condition $f < g + \varepsilon \vee f > g$. The condition $f < g + \varepsilon \vee f > g$ is not fundamentally stronger in the sense that a rule which concludes it could be derived from splitReal by varying the choice of ε in the application of splitReal. Undecidability of exact comparisons leads to subtle liveness arguments in the proof (Appendix B.1) of the example game (Section 5.2.2) because loop termination cannot be guarded by exact Boolean comparisons.

$$\text{(splitReal)} \quad \frac{\varepsilon > 0}{\Gamma \vdash f \leq g + \varepsilon \vee f > g}$$

Though CdGL synthesis is not developed in detail until Chapter 8, we take a brief de-

tour to discuss the implications of rule splitReal on synthesis because those implications are subtle, yet crucial. Due to the performance limitations of exact representation of computable reals, synthesis will ultimately employ a sound approximation of real numbers as rational intervals. Intervals and constructive reals share the crucial property that they require comparisons to be inexact. The subtle difference between the two is that the constructive real comparisons implemented with rule splitReal specify their own comparison precision ε which is independent of the compared terms f and g, whereas interval comparisons compute intervals for the values of f and g, upon which the amount of inexactness in the comparison is simply the combined interval width of f and g, that is, the amount of inexactness in the terms being compared. The takeaway of these similarities and differences is that by using constructive reals instead of classical reals, we provide hope that conditionals over intervals can be synthesized from conditionals over constructive reals. Whenever interval code assigns f and g intervals whose combined with is bounded by δ, it is guaranteed that at least one disjunct of $f \leq g + \varepsilon \vee f > g$ holds, which is crucial for the *liveness* of generated code, because code which branches on $f \leq g + \varepsilon \vee f > g$ would get stuck if it cannot determine which branch is true. While stuck branches are still possible for intervals whose combined widths exceed ε, this is greatly preferably to the use of exact comparisons wherein branches can get stuck even for arbitrarily-small (proper) intervals. In Chapter 8, we will leave it to the user to ensure that intervals are sufficiently small, though one could also attempt to prove intervals are sufficiently small to ensure liveness.

$$\text{(DC)}\quad \frac{\Gamma \vdash [x'=f \,\&\, \psi]\rho \quad \Gamma \vdash [x'=f \,\&\, \psi \wedge \rho]\phi}{\Gamma \vdash [x'=f \,\&\, \psi]\phi} \qquad \text{(DI)}\quad \frac{\Gamma \vdash \phi \quad \Gamma \vdash \forall x\,(\psi \rightarrow [x' := f](\phi)')}{\Gamma \vdash [x'=f \,\&\, \psi]\phi}$$

$$\text{(DG)}\quad \frac{\Gamma, y = f_0 \vdash [x'=f, y' = a(x)y + b(x) \,\&\, \psi]\phi}{\Gamma \vdash [\{x'=f \,\&\, \psi\}; \{y := *; y' := *\}^d]\phi}\ {}^{1} \qquad \text{(DW)}\quad \frac{\Gamma \frac{y}{x}, \psi \vdash \phi}{\Gamma \vdash [x'=f \,\&\, \psi]\phi}$$

$$\text{(DV)}\quad \frac{\begin{array}{c}\psi, h \geq g \vdash \phi \quad \Gamma \vdash d > 0 \wedge \varepsilon > 0 \wedge h - g \geq -d\varepsilon \\ \Gamma \vdash \langle t := 0; \{t'=1, x'=f \,\&\, \psi\}\rangle t \geq d \quad \Gamma \vdash [x'=f]((h)' - (g)') \geq \varepsilon\end{array}}{\Gamma \vdash \langle x'=f \,\&\, \psi\rangle\phi}$$

$$\text{(bsolve)}\quad \frac{\Gamma \frac{y}{x}, t \geq 0, \hat{\psi}, x = sln\frac{y}{x}, x' = f \vdash \phi}{\Gamma \vdash [t := 0; \{t' = 1, x' = f \,\&\, \psi\}]\phi}\ {}^{2}$$

$$\text{(dsolve)}\quad \frac{\begin{array}{c}\Gamma \vdash d \geq 0 \quad \Gamma \frac{y}{x}, 0 \leq t \leq d, x = sln\frac{y}{x}, x' = f \vdash \psi \\ \Gamma \frac{y}{x}, 0 \leq t = d, x = sln\frac{y}{x}, x' = f \vdash \phi\end{array}}{\Gamma \vdash \langle t := 0; \{t' = 1, x'=f \,\&\, \psi\}\rangle\phi}\ {}^{3}$$

[1] $y \notin \mathrm{FV}(\Gamma) \cup \mathrm{FV}(f_0) \cup \mathrm{FV}(f) \cup \mathrm{FV}(a) \cup \mathrm{FV}(b) \cup \mathrm{FV}(\psi) \cup \{x\}$
[2] y fresh, $x' \notin \mathrm{FV}(\phi)$, $\{t, t'\} \cap \mathrm{FV}(\Gamma) = \emptyset$, and sln solves $\{t' = 1, x' = f \,\&\, \psi\}$
[3] y fresh, $x' \notin \mathrm{FV}(\phi)$, $\{t, t', x, x'\} \cap \mathrm{FV}(d) = \emptyset$, $\{t, t\}' \cap \mathrm{FV}(\Gamma) = \emptyset$, and sln solves $\{t' = 1, x' = f \,\&\, \psi\}$

Figure 5.2: CdGL proof calculus: ODEs.

5.4.2 ODE Proofs

We resume discussion of the CdGL proof calculus and turn our focus to ODE reasoning. Figure 5.2 gives the ODE rules, which are constructive versions of dGL (Platzer, 2015a) rules for ODEs. In each rule, y is fresh. In rule schemata bsolve and dsolve, it is a side condition that sln solves $x' = f$ for *all* time $t \geq 0$ such that the domain constraint ψ holds. Rule schemata bsolve and dsolve also assume as side conditions that $x' \notin \mathrm{FV}(\phi)$, t is fresh in Γ, and they treat d as a term metavariable for an argument of the rule schemata, not a program variable. Abbreviation $\hat{\psi} \equiv \forall 0 \leq s \leq t\, [t := s; x := sln]\psi$ says the domain constraint ψ holds through time t where s is fresh. For nilpotent[12] ODEs such as the plant of Example 5.1, reasoning via solutions is possible. Since CdGL supports nonlinear ODEs which often do not have closed-form solutions[13], we provide invariant-based rules, whose classical counterparts have been shown complete (Platzer & Tan, 2020) for semianalytic invariants of polynomial ODEs. Invariant reasoning can even be easier for nilpotent ODEs compared to solution-based reasoning in the case that the solution of an ODE is complicated. *Differential induction* rule DI (Platzer, 2010a) says ϕ is an invariant of an ODE if it holds initially and if its *differential formula* (Platzer, 2010a) $(\phi)'$ holds throughout, for example $(f \geq g)' \equiv ((f)' \geq (g)')$. The definition of differential formulas $(\phi)'$ in CdGL is identical to the definition used in paper presentations of dL (Platzer, 2010a), which is a definition by induction on ϕ. This is in contrast to the presentation of dL in Chapter 2, which separately implements special cases of DI in order to remove $(\phi)'$ from the language and thus avoid the subtleties of defining a formal semantics for $(\phi)'$ that works for all formulas ϕ of the dL uniform substitution calculus.

Soundness of rule DI requires differentiability of terms mentioned in ϕ, thus $(\phi)'$ is constructed in such a way as to be unprovable when ϕ mentions nondifferentiable terms. Just as soundness of classical differential induction relies on the mean-value theorem, soundness of DI relies on the constructive mean-value theorem. For a statement of (the relevant corollaries of) the constructive mean-value theorem, see Theorem B.5 in Appendix B.2. Aside from the fact that it operates over constructive reals rather than classical reals, the constructive mean-value theorem has the same assumptions as its classical counterpart, so that our rule DI does not require additional assumptions compared to its classical counterpart.

Differential cut rule DC (Platzer, 2017a, 2008b) proves ρ invariant, then adds it to the domain constraint. As in dL, soundness of differential cuts relies on the fact that the set of trajectories for any ODE is prefix-closed, a property which is easily proved without appealing to real analysis. *Differential weakening* rule DW says that if ϕ follows from the domain constraint, it holds throughout the ODE. Soundness of differential weakening follows directly from the semantics of an ODE game without any appeal to real analysis. *Differential ghost* rule DG (Platzer, 2017a, 2012d) permits us to augment an ODE system

[12]An ODE is nilpotent if its matrix becomes zero when raised to a sufficient power. The solutions of nilpotent ODEs are polynomial, which makes them amenable to first-order reasoning.

[13]Existence of constructive solutions means that the final value of each value can be computed to arbitrary precision given a single initial state which is computable to arbitrary precision. It does not mean the output for even an individual state has a finite closed form, let alone a closed form which describes the output for every state.

with a new dimension y, which enables (Platzer & Tan, 2020) proofs of otherwise unprovable properties. In discussing why rule DG is sound, we again encounter one of the only differences between constructive and classical ODE reasoning: new ODE dimensions must be *effectively*-locally-Lipschitz[14], meaning that for every point in the domain, a neighborhood and a (local) Lipschitz constant L can be constructed such that it is constructively provable that L is a (local) Lipschitz-constant on the neighborhood. Just like its classical counterpart, however, the rule DG already requires the right-hand-side for the new dimension y to be linear in y and continuous in x because under that assumption Constructive Picard-Lindelöf (Makarov & Spitters, 2013) (correspondingly, classical Picard-Lindelöf in dGL) ensures an unchanged duration, which is essential to soundness. Because every linear function is effectively-locally-Lipschitz (even effectively-globally-Lipschitz), our rule DG does not need an extra assumption to ensure effective local Lipschitz continuity. We would only need to explicitly prove extra assumptions on effective local Lipschitz continuity if we were to generalize rule DG to allow non-linear differential ghosts. Even then, we are aware of no ODE of practical interest that only satisfies the classical notion of local Lipschitz continuity without satisfying our effective local Lipschitz continuity requirement.

In rule DG, the assignments $\{y := *; y' := *\}^d$ say that the postcondition is only known to hold for some unknown y and y'. We present DG with the assignments $\{y := *; y' := *\}^d$ in order to match the intended use in Chapter 6. The more common presentation of rule DG (Bohrer & Platzer, 2020b) elides the assignments and instead assumes that y is fresh. The two presentations are equivalent. *Differential variant* (Platzer, 2010a; Tan & Platzer, 2019) rule DV is an Angelic counterpart to rule DI. The schema parameters d and ε must not mention x, x', t, t'. To show that f eventually exceeds g, first choose a duration d and a sufficiently high minimum rate ε at which $h - g$ will change. Then prove that $h - g$ decreases at rate at least ε and that the ODE has a solution of duration d satisfying constraint ψ. Thus, at time d, both $h \geq g$ and its provable consequences hold. Rule DV differs from its classical counterpart in a notable way: a constructive proof must provide explicit witnesses for duration d and rate ε, while a classical proof need only show their existence. In a practical classical proof, existence of d and ε might be discharged by an automated solver, whereas the author of a constructive proof would choose d and ε explicitly. Rules bsolve and dsolve assume as a side condition that *sln* is the unique

[14]In the proofs, we will also use the fact that ODEs are effectively-globally-Lipschitz continuous on compact domains. Effective Lipschitz continuity with a uniform Lipschitz constant (as in effective global Lipschitz continuity) on each compact subsets of the state space is equivalent to effective local Lipschitz continuity on the state space because the state space is Euclidean, and thus a locally-compact metric space. It is intuitive that the two notions would agree because effective local Lipschitz continuity is a quantified statement over neighborhoods of states, and the closure of each such neighborhood is a compact subset of the state space. On the other hand, some readers are perhaps surprised that the two notions of continuity agree, because there exist ODEs, such as $x' = x^2$, which are effectively-locally-Lipschitz continuous, but do not have a solution on every compact subset of the state space, i.e., the existence interval is finite. There is no contradiction here: the cited formalization assumes, in addition to its continuity assumption, that the diameter of the compact set can be bounded below as a function of the effective Lipschitz constant, a condition which fails for many ODEs, such as those with finite existence intervals. Indeed, the main gap remaining between the formalization and our usage of it is that our notion of effective local Lipschitz continuity drops the requirement of a lower bound on diameter.

solution of $x' = f$ on domain ψ. They are convenient for ODEs with simple solutions, while invariant reasoning supports complicated ODEs. The rules bsolve and dsolve are presented here with an initial assignment $t := 0$ for the time variable. This presentation is used because it will simplify computations over derivations developed in Chapter 6. The conference version of this work (Bohrer & Platzer, 2020b) presents the rules without initial assignments because that presentation is more compositional, albeit less appropriate for use in Chapter 6. We suspect that our rule is interderivable with the more compositional rule from the conference version (Bohrer & Platzer, 2020b), but have not attempted a proof of interderivability. Practically speaking, the reader should use whichever rule fits their preference.

5.5 Theory: Soundness

Following constructive counterparts of classical soundness proofs for dGL, we prove that the CdGL proof calculus (Section 5.4) is sound: provable formulas are true in the CIC semantics. For the sake of readability, we give statements here and as well as proof outlines for some of the stated properties. All proofs and lemmas statements are reported in Appendix B.2. Similar lemmas have been used to prove soundness of dGL (Platzer, 2018b), but our new semantics lead to simpler statements for Lemmas 5.4, 5.5, and 5.6. The coincidence property for terms is not proved but assumed, since we inherit a semantic treatment of terms from the host theory. Let $s\frac{y}{x}$ be the state which is equal to state s except that the values of x and y are swapped. Let s^f_x be $\mathsf{set}\ s\ x\ (f\ s)$ where $f\ s$ is the value of term f in state s since terms in CdGL are just computable functions over the state. The respective notations $M\frac{y}{x}, e\frac{y}{x}$, and $\Gamma\frac{y}{x}$ for renaming in proof terms M, expressions e, and contexts Γ are analogous to the renaming notation for states. For finite[15] variable sets V, we definite a CIC proposition $s \overset{V}{=} t \leftrightarrow \star_{x\in V}(s\ x = t\ x)$ which says the states s and t agree on all $x \in V$. The notation for $\star_{x\in V}$ is shorthand for a nested pair type which conjoins the types $(s\ x = t\ x)$ for each of the finitely many $x \in V$. Recall as always that $\mathrm{FV}(e), \mathrm{BV}(\alpha)$, and $\mathrm{MBV}(\alpha)$ stand for free variables of expression e, may-bound variables of game α, and must-bound variables of game α, respectively.

Note that because the proofs of uniform renaming are constructive, they implicitly define an algorithm for computing the renaming $M\frac{y}{x}$ of a proof term M.

Lemma 5.2 (Formula uniform renaming)**.** *If* $\ulcorner\Gamma\urcorner(s) \vdash M : \ulcorner\phi\urcorner\ s$ *then there exists some proof term (call it* $M\frac{y}{x}$*) such that* $\ulcorner\Gamma\urcorner(s\frac{y}{x}) \vdash M\frac{y}{x} : \ulcorner\phi\frac{y}{x}\urcorner\ s\frac{y}{x}$*. Also,* $(sol, s, d \vDash z' = f) \Leftrightarrow (sol, s\frac{y}{x}, d \vDash (z\frac{y}{x})' = f\frac{y}{x})$*. The function sol requires no renaming as it is just a univariate function from time to the value of a single scalar variable, as opposed to a function which accepts or returns a state.*

Lemma 5.3 (Game uniform renaming)**.** *If* $\ulcorner\Gamma\urcorner(s) \vdash M : \langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\phi\urcorner\ s$ *then there exists* $M\frac{y}{x}$ *such that* $\ulcorner\Gamma\urcorner(s\frac{y}{x}) \vdash M\frac{y}{x} : \langle\!\langle\alpha\frac{y}{x}\rangle\!\rangle\ \ulcorner\phi\frac{y}{x}\urcorner\ s\frac{y}{x}$*. If* $\ulcorner\Gamma\urcorner(s) \vdash M : [[\alpha]]\ \ulcorner\phi\urcorner\ s$ *then there exists* $M\frac{y}{x}$ *such that* $\ulcorner\Gamma\urcorner(s\frac{y}{x}) \vdash M\frac{y}{x} : [[\alpha\frac{y}{x}]]\ \ulcorner\phi\frac{y}{x}\urcorner\ s\frac{y}{x}$.

[15] If we wish to support infinite variable sets, we could use a dependent function over V rather than a nested pair.

Lemma 5.4 (Formula Coincidence)**.** *Let $s,t : \mathfrak{S}$ and let $V \supseteq \mathrm{FV}(\phi)$ be a set of variables. Assume $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash s \overset{V}{=} t$, meaning s and t assign equal values to each variable in V. Assume $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash (\ulcorner\phi\urcorner\ s)$, then $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash (\ulcorner\phi\urcorner\ t)$.*

Coincidence for contexts also holds. Note that the states s and t at which we consider the conclusion context Γ_2 need not be the distinguished state $\mathfrak{s}$ of the context Γ_1. The claim can also be generalized to allow an arbitrary state argument for Γ_1 as a consequence by applying it again with $\Gamma_2 = \Gamma_1$.

If $\ulcorner\Gamma_1\urcorner(\mathfrak{s}) \vdash s \overset{V}{=} t$ for $V \supseteq \mathrm{FV}(\Gamma_2)$ then $\ulcorner\Gamma_1\urcorner(\mathfrak{s}) \vdash \ulcorner\Gamma_2\urcorner(s)$ is inhabited iff $\ulcorner\Gamma_1\urcorner(\mathfrak{s}) \vdash \ulcorner\Gamma_2\urcorner(t)$ is inhabited.

Coincidence for the construct $(sol, s, d \vDash x' = f)$ also holds: If $s \overset{\mathrm{FV}(f)\cup\{x\}}{=} t$ then $(sol, s, d \vDash x' = f) \Leftrightarrow (sol, t, d \vDash x' = f)$.

Lemma 5.5 (Game coincidence)**.** *If $s \overset{V}{=} t$ for $V \supseteq \mathrm{FV}(\alpha) \cup (\mathrm{FV}(\phi) \setminus \mathrm{MBV}(\alpha))$ then:*

- *if $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash (\langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\phi\urcorner\ s)$ is inhabited then $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash (\langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\phi\urcorner\ t)$ is.*
- *if $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash ([[\alpha]]\ \ulcorner\phi\urcorner\ s)$ is inhabited then $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash ([[\alpha]]\ \ulcorner\phi\urcorner\ t)$ is.*

Because the proof is constructive, it amounts to an algorithm which computes an inhabitant of the conclusion.

Lemma 5.6 (Bound effect)**.** *Let $P : \mathfrak{S} \Rightarrow \mathbb{T}$ and $V \subseteq \mathrm{BV}(\alpha)^{\complement}$, a subset of the complement of bound variables of α. For simplicity, we state the case where truth of all formulas in Γ is evaluated at designated state $\mathfrak{s}$. To apply bound effect at a context which is evaluated at some other state, additionally apply the context coincidence claim from Lemma 5.4.*

- *There exists M such that $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash M : \langle\!\langle\alpha\rangle\!\rangle\ P\ s$ iff there exists N such that $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash N : \langle\!\langle\alpha\rangle\!\rangle\ (\lambda t.\ P\ t \star (s \overset{V}{=} t))\ s$.*
- *There exists M such that $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash M : [[\alpha]]\ P\ s$ iff there exists N such that $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash N : [[\alpha]]\ (\lambda t.\ P\ t \star (s \overset{V}{=} t))\ s$.*

Because the proof of each claim is constructive, each proof amounts to an algorithm for computing a proof term N satisfying the respective claim.

The term substitution admissibility definition follows that of Chapter 4 (Def. 4.14) but it is repeated here for convenience.

Definition 5.7 (Term substitution admissibility)**.** A substitution σ is admissible for expression e (likewise for contexts Γ and CIC proof terms M) if e does not bind any variable x replaced by σ (any $x \in Dom(\sigma)$), nor mention that x free under a binder of any free variable of $\sigma(x)$.

Recall that while there exist sound admissibility notions that loosen the restriction on binding x in e, our restriction is not prohibitive in practice because any such bindings could be eliminated before substitution through the use of α-renaming and the introduction of stuttering assignments $x := x$.

Let σ be a substitution replacing program variables with terms. Let $\sigma(e)$ and $\sigma(\Gamma)$ respectively be the application of substitution σ to e or Γ. In the special case that σ is a singleton substitution $\cdot^f_x$, we also alternatively write e^f_x and Γ^f_x for the result of substitution on (respectively) expressions and contexts. Let $\sigma^*_s(t)$ be the adjoint state of t at s, which agrees with t except on $x \in Dom(\sigma)$ which are assigned the value of $\sigma(x)$ at state s.

Lemma 5.7 (Formula substitution). *If σ is admissible for Γ, then $\ulcorner\sigma(\Gamma)\urcorner(s) \Leftrightarrow \ulcorner\Gamma\urcorner(\sigma^*_s(s))$. If σ is additionally admissible for ϕ then $\ulcorner\Gamma\urcorner(\sigma^*_s(s)) \vdash M : \ulcorner\phi\urcorner\ \sigma^*_s(s)$ iff there exists a CIC proof term (call it $\sigma(M)$) such that $\ulcorner\sigma(\Gamma)\urcorner(s) \vdash \sigma(M) : \ulcorner\sigma(\phi)\urcorner\ s$. Likewise for predicate $(sol, s, d \vDash y' = g)$. In the converse direction, the witness is not necessarily M.*

Because the proof (in Appendix B.2) is constructive, it amounts to an algorithm for computing the substitution result $\sigma(M)$ for CIC proof terms M of the formula and context semantics. From here onward when M is a CIC proof term for the formula or context semantics, notation $\sigma(M)$ refers specifically to the proof term constructed in accordance to the proof.

Lemma 5.8 (Game substitution). *If σ is admissible for $\langle\alpha\rangle\phi$ or respectively $[\alpha]\phi$, then:*

- *$\ulcorner\Gamma\urcorner(\sigma^*_s(s)) \vdash M : \langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\phi\urcorner\ \sigma^*_s(s)$ iff there exists a CIC proof term (call it $\sigma(M)$) such that $\ulcorner\sigma(\Gamma)\urcorner(s) \vdash \sigma(M) : \langle\!\langle\sigma(\alpha)\rangle\!\rangle\ \ulcorner\sigma(\phi)\urcorner\ s$.*
- *$\ulcorner\Gamma\urcorner(\sigma^*_s(s)) \vdash M : [[\alpha]]\ \ulcorner\phi\urcorner\ \sigma^*_s(s)$ iff there exists a CIC proof term (call it $\sigma(M)$) such that $\ulcorner\sigma(\Gamma)\urcorner(s) \vdash \sigma(M) : [[\sigma(\alpha)]]\ \ulcorner\sigma(\phi)\urcorner\ s$.*

In the converse direction, the witness is not necessarily M.

Because the proof (in Appendix B.2) is constructive, it amounts to an algorithm for computing the substitution result $\sigma(M)$ for CIC proof terms M of the semantics of games. From here onward when M is a CIC proof term for the semantics of games, notation $\sigma(M)$ refers specifically to the proof term constructed in accordance to the proof.

Soundness of the proof calculus follows from the lemmas, with soundness proofs of the ODE rules employing several known results from constructive analysis. Because the proof of soundness is constructive, it amounts to an algorithm which consumes a CdGL proof and constructs an inhabitant of the type corresponding to the conclusion. The full proof of soundness is in Appendix B.2.

Theorem 5.9 (Soundness). *Recall that the CdGL sequent $(\Gamma \vdash \phi)$ is* valid *iff there exists M where $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash M : (\ulcorner\phi\urcorner\ \mathfrak{s})$.*

If sequent $(\Gamma \vdash \phi)$ has a proof in CdGL, then sequent $(\Gamma \vdash \phi)$ is valid in CdGL. As a special case, if sequent $(\cdot \vdash \phi)$ has a CdGL proof, then formula ϕ is valid in CdGL.

Proof Sketch. By induction on the derivation. The case for rule [:∗]E is proven by appealing to Lemma 5.7. Lemma 5.5, Lemma 5.4, and Lemma 5.6 are applied when maintaining truth of a formula across changing state. The equality and inequality cases of rules DI and DV employ the constructive mean-value theorem (Theorem B.5 in Appendix B.2), which has been formalized, e.g., in Coq (Cruz-Filipe et al., 2004). Rules DW, bsolve, and dsolve follow from the semantics of ODEs. Rule DC uses the fact that prefixes of solutions are themselves solutions. Rule DG uses Constructive Picard-Lindelöf (Makarov & Spitters, 2013), which constitutes an algorithm for arbitrarily approximating the solution of any ODE which is effectively-globally-Lipschitz on some compact domain. Effective global Lipschitz continuity of the original ODE on compact domains follows from our assumption of effective local Lipschitz continuity because the state space is a locally-compact metric space. The full soundness proof of DG constructs the (uniform, as in global effective Lipschitz continuity) Lipschitz constant on each compact domain (corresponding to each compact time interval $[0, t]$ for which the ODE evolves) for the ghost variable using the fact that it is the composition of a linear function with the (continuous) solution of x. The convergence

rate of the solution approximation is a function of each global Lipschitz constant and the size of each domain. It is crucial that our (local and global) notions of Lipschitz continuity be effective because effectiveness provides explicit constructions for Lipschitz constants, which enable explicitly calculating how quickly the solution approximation converges, and thus calculating how many times the approximation must be iterated to provide a correct solution within any desired precision. □

We have shown that every provable formula is true in the type-theoretic semantics. Because the soundness proof is constructive, it amounts to an extraction algorithm from CdGL into type theory: from each CdGL proof, we can compute a program in type theory which inhabits the corresponding type of the semantics. Soundness holds for our constructive semantics despite our ODE rules having no major differences from their dGL predecessors except the use of constructive reals and effective local Lipschitz-continuity. In contrast to the discrete fragment of CdGL, the ODE fragment of CdGL does not introduce any major new proof burdens on the author of proof, meaning the user of the proof calculus gets constructivity of ODE proofs "for free".

5.6 Theory: Extraction and Execution

Another perspective on constructivity is that provable properties must have witnesses. We show Existence and Disjunction properties providing witnesses for existentials and disjunctions. The Existence and Disjunction properties are commonly used as tests of whether a logic is constructive and are crucial because their proofs amount to on-paper synthesis algorithms for existential quantifiers and disjunctions. As in Chapter 4, we prove weak versions of the Existence and Disjunction property which show that *semantic* rather than *syntactic* witnesses exist. The semantics are computational, so our proofs still amount to the extraction of executable witnesses from proofs of existentials and disjunctions.

Lemma 5.10 (Existence Property)**.** *Let $s : \mathfrak{S}$. If $\ulcorner\Gamma\urcorner(s) \vdash M : (\ulcorner\exists x\, \phi\urcorner\ s)$ then there constructively exists number $v : \mathbb{R}$ and CIC proof term N such that $\ulcorner\Gamma\urcorner(s) \vdash N : (\ulcorner\phi_x^v\urcorner\ s)$.*

Lemma 5.11 (Disjunction Property)**.** *If $\ulcorner\Gamma\urcorner(s) \vdash M : (\ulcorner\phi \vee \psi\urcorner\ s)$ then there constructively exists a proof term N such that $\ulcorner\Gamma\urcorner(s) \vdash N : (\ulcorner\phi\urcorner\ s)$ or $\ulcorner\Gamma\urcorner(s) \vdash N : (\ulcorner\psi\urcorner\ s)$.*

The proofs (Appendix B.2) follow their counterparts in type theory. In contrast to standard statements of the Disjunction Property, our weak Disjunction Property considers truth at a *specific state*. Its strong counterpart is simply false in dynamic logic in general: validity of $\phi \vee \psi$ does *not* imply validity of either ϕ or ψ. For example, $x < 1 \vee x > 0$ is valid, but its disjuncts are not. Rather, at least one disjunct is provable in each state, not necessarily the same disjunct. Because CdGL allows arbitrary computable functions as terms, the standard, syntactic Existence Property is likely to hold in CdGL whereas it did not in CGL. Regardless, we have proved the semantic version instead for the sake of symmetry and because its proof is particularly direct.

Recall that CGL provided a Strategy Property as a counterpart to the Existence and Disjunction properties for modal formulas $\langle\alpha\rangle\phi$ and $[\alpha]\phi$, as did CGL. The Strategy Property showed that each provable modality has a realizer which, according to the realizability

semantics of CGL, would win the game regardless of the behavior of the opponent Demon. The Strategy Property was an immediate consequence of soundness, and because CdGL is also sound, it is also the case that every provable CdGL modality has a corresponding CIC term which implements a winning strategy. However, a major difference between the CGL semantics and CdGL semantics is that the type-theoretic CdGL semantics describe how strategies are built rather whereas the CGL semantics describe how games are played. To complete the story of CdGL, we wish to show that the winning strategy corresponding to a proof can actually win against its opponent. To do so, we develop a big-step operational semantics play which allows playing two strategies (both constructive!)[16] against each other to extract a proof that both of their goals hold in some final state t. Combined with soundness, the big-step operational semantics show that the CIC term extracted from each proof can actually be played against any (constructive) opponent and achieve its postcondition when doing so. By reusing CIC as the language of strategies, we avoid the technical expense of explicitly defining and analyzing a new language of strategies, while providing the key result that winning strategies corresponding to CdGL proofs can play and win games.

Function play below gives a big-step semantics where a constructive Angel and constructive Demon play a common game α with respective goal regions P and Q. The CIC terms as and ds belong to the CIC type families prescribed by the semantics of α (i.e., they respectively belong to $\langle\!\langle\alpha\rangle\!\rangle\ P\ s$ and $[[\alpha]]\ Q\ s$) and are understood as CIC terms which implement the strategies for Angel and Demon starting from state s with the respective goal regions. The semantics play show that given the game, goal regions, and strategies, a final state t can be constructed which satisfies both of the goals. Note that the type of play suggests play can (as desired) only construct the final state t by actually playing the strategies as and ds. Arbitrary regions P and Q are given as arguments to play and play could not hope to construct an element of an arbitrary region out of thin air.

Strategies as and ds are both constructive, of which as is initially in control. Since Angel is the constructive player's name in this thesis, one intuition for play is that Angel wrote two strategies which she plays against each other. The type-theoretic semantics are slightly more general in that they allow a classical Demon. Two constructive strategies are used in play mainly to avoid explicitly defining classical Demon strategies and their semantics. Later implementation work (Chapter 8) will ultimately use a distinct representation for Demon strategies, but we found two constructive strategies useful for our theoretical presentation in the play operator. We give the type of play:

$$\mathsf{play}:\ \Pi\alpha : \mathsf{Game}.\,\Pi P, Q : (\mathfrak{S} \Rightarrow \mathbb{T}).\,\Pi s : \mathfrak{S}.\,\big(\langle\!\langle\alpha\rangle\!\rangle\ P\ s \Rightarrow [[\alpha]]\ Q\ s \Rightarrow \Sigma t : \mathfrak{S}.\,(P\ t \star Q\ t)\big)$$

In the type of play, the first argument α is the game to be played, the next two arguments P and Q are CIC regions which are the semantic representation of the respective goal regions of Angel and Demon, the following argument s is the state in which gameplay starts, and the final two arguments (called as and ds) are CIC terms which implement the respective strategies of Angel and Demon for respective goal regions assuming the initial state is s. The result of play is a dependent pair containing a state t (the final state of the game) and CIC proof terms that t belongs to *both* goal regions.

[16]When we implement synthesis (Chapter 8), we will let the opponent make decisions classically and announce them to Angel. We use constructive strategies for both players here for theoretical simplicity.

To avoid heavyweight notation, we only write down the arguments of play which actually change throughout recursive calls or are actually mentioned in its body. That is, applications of play are written $\mathsf{play}_\alpha\ s\ \mathsf{as}\ \mathsf{ds}$ and we do not write the arguments P or Q explicitly, to save space. Because P and Q also appear in the types of as and ds, writing them down would be redundant, anyway. Note that α^d is played by swapping the Angelic and Demonic strategies in α. The recursive call in the α^d case typechecks precisely because both strategies are constructive and by the α^d cases of the type-theoretic semantics.

Recall that in CdGL (and CGL), the diamond modality gives a constructive strategy for a player who is currently in control, while the box modality gives a constructive strategy for a player who is *not* currently in control. As expected under that interpretation, the diamond strategy is the driving force in the control flow of play: it resolves all strategic conditions, which are passively consumed by the box strategy. If a duality is encountered, the players (or strategies) switch turns. When an atomic program is encountered, each player proves their postcondition, though in the case of tests, the proof of the test condition must be passed to the box strategy as an assumption before it is able to prove its postcondition.

$$
\begin{aligned}
\mathsf{play}_{x:=f}\ s\ \mathsf{as}\ \mathsf{ds} &\equiv (\mathsf{let}\, t\ =\ \mathsf{set}\ s\ x\ (f\ s)\, \mathsf{in}\, (t, (\mathsf{as}\ t, \mathsf{ds}\ t)))\\
\mathsf{play}_{x:=*}\ s\ \mathsf{as}\ \mathsf{ds} &\equiv \mathsf{let}\, t\ =\ \mathsf{set}\ s\ x\ \pi_0\mathsf{as}\, \mathsf{in}\, (t, (\pi_1\mathsf{as}, \mathsf{ds}\ \pi_0\mathsf{as}))\\
\mathsf{play}_{x'=f\,\&\,\psi}\ s\ \mathsf{as}\ \mathsf{ds} &\equiv \mathsf{let}\, (d, \mathrm{sol}, \mathrm{solves}, c, p)\ =\ \mathsf{as}\ s\, \mathsf{in}\\
&\qquad\qquad (\mathsf{set}\ s\ x\ (\mathrm{sol}\ d), (p, \mathsf{ds}\ d\ \mathrm{sol}\ \mathrm{solves}\ c))\\
\mathsf{play}_{?\phi}\ s\ \mathsf{as}\ \mathsf{ds} &\equiv (s, (\pi_1\mathsf{as}, \mathsf{ds}\ (\pi_0\mathsf{as})))\\
\mathsf{play}_{\alpha\cup\beta}\ s\ \mathsf{as}\ \mathsf{ds} &\equiv \mathsf{case}\ (\mathsf{as}\ s)\ \mathsf{of}\\
&\qquad\qquad \mathsf{as}_\ell \Rightarrow \mathsf{play}_\alpha\ s\ \mathsf{as}_\ell\ (\pi_0\mathsf{ds})\\
&\qquad\qquad \mid \mathsf{as}_r \Rightarrow \mathsf{play}_\beta\ s\ \mathsf{as}_r\ (\pi_1\mathsf{ds})\\
\mathsf{play}_{\alpha;\beta}\ s\ \mathsf{as}\ \mathsf{ds} &\equiv (\mathsf{let}\, (t, (\mathsf{as}', \mathsf{ds}'))\ =\ \mathsf{play}_\alpha\ s\ \mathsf{as}\ \mathsf{ds}\, \mathsf{in}\, \mathsf{play}_\beta\ t\ \mathsf{as}'\ \mathsf{ds}')\\
\mathsf{play}_{\alpha^*}\ s\ \mathsf{as}\ \mathsf{ds} &\equiv \mathsf{case}\ (\mathsf{as}\ s)\ \mathsf{of}\\
&\qquad\qquad \mathsf{as}_\ell \Rightarrow (s, (\mathsf{as}_\ell, \pi_0\mathsf{ds}))\\
&\qquad\qquad \mid \mathsf{as}_r \Rightarrow \mathsf{let}\, (t, (\mathsf{as}', \mathsf{ds}'))\ =\ \mathsf{play}_\alpha\ s\ \mathsf{as}_r\ (\pi_1\mathsf{ds})\, \mathsf{in}\, \mathsf{play}_{\alpha^*}\ t\ \mathsf{as}'\ \mathsf{ds}'\\
\mathsf{play}_{\alpha^d}\ s\ \mathsf{as}\ \mathsf{ds} &\equiv \mathsf{play}_\alpha\ s\ \mathsf{ds}\ \mathsf{as}
\end{aligned}
$$

As an aside, we observe that a game consistency property for two constructive players (Corollary 5.12) follows from the fact that the type of play is inhabited (i.e., the fact that play typechecks) and from the consistency of type theory.

Corollary 5.12 (Consistency)**.** *It is never the case that both the type* $\ulcorner\langle\alpha\rangle\phi\urcorner\ s$ *and the type* $\ulcorner[\alpha]\neg\phi\urcorner\ s$ *are inhabited.*

Proof. Suppose $\mathsf{as} : \ulcorner\langle\alpha\rangle\phi\urcorner\ s$ and $\mathsf{ds} : \ulcorner[\alpha]\neg\phi\urcorner\ s$, then $\pi_1(\mathsf{play}_\alpha\ s\ \mathsf{as}\ \mathsf{ds}) : \bot$, contradicting consistency of type theory. □

We proved consistency because it is a commonly-studied property of game logics, but the consistency property will also contribute to the following discussion of some subtleties in the play semantics.

One may wonder how play achieves both players' goals when the players are in competition or wonder how to reconcile the big-step nature of play with the fact that one player may win a game early by forcing the opponent to fail a test. Because zero-sum games are consistent, one does not expect both players to win. However, an important aspect of understanding the type of play is that consistency says only one player wins when the players have opposite goals, whereas play allows different goals P and Q. In the extreme case where we wish to use play to execute a game where Angel has total control over the state, we would set Demon's goal region to the set of all states (corresponding to a tautology such as $1 > 0$ in the postcondition of a formula). However, even the tautological goal region does not hold for every game, rather a Demon strategy for the tautological goal region is a strategy to *reach the end of the game without failing a test*. Thus, the play semantics only apply to games where both players have a strategy to reach the end of the game, and neither has a strategy that forces the other to fail a test. Just as the CGL semantics (Chapter 4) explicitly handle early test failures, one could develop a big-step semantics which allows early failure, but it would be more complicated than ours. To do so, one would have to generalize the return type to include the possibility of early failure and also develop a weak notion of Demonic strategies which do not necessarily win the game with any goal region (just as the set of all realizers in CGL includes realizers which never win). Such a semantics could be developed, but early termination has already been explored in CGL and we prefer to present the big-step semantics play without a treatment of early termination because of its greater simplicity.

5.7 Summary and Discussion

We extended Constructive Game Logic CGL to CdGL for constructive *hybrid* games. We introduced a type-theoretic semantics for CdGL which, compared to realizers, can more easily be described precisely and has a direct correspondence with well-known functional languages. Because Chapter 4 has already developed a detailed proof term language for the discrete fragment, we did not develop an explicit language of proof terms in this chapter; however, soundness (Theorem 5.9) implies that every proof in CdGL has a corresponding proof term in the host type theory. We extended the CGL natural deduction calculus to support differential equations and constructive reals. Major theorems of Chapter 4 were proved again for the new semantics and proof rules. We proved an example reach-avoid correctness theorem for 1D driving with adversarial timing. Reach-avoid correctness proofs, in contrast to simple safety proofs, contain enough information to extract both control and monitoring code, and will be a focus of synthesis work in Chapter 8. The Logic-User is pleased that reach-avoid properties, a broad class of properties, are well-supported in CdGL, and the Engineer will be pleased with the code that is ultimately synthesized.

In addition to a type-theoretic static semantics, we developed a big-step operational semantics which demonstrates how two constructive strategies may be played against one another. The purpose of our operational semantics is similar to the purpose of Theorem 4.20 in Chapter 4: it shows how strategies might be synthesized or executed, thus enabling the robust reimplementation of VeriPhy that the Engineer desires. The big-step operational

semantics further have the advantage that they are written directly in a type theory whose conversion to executable code is well-understood. While well-understood, it is also true that our use of constructive reals and constructive ODE could make a naïve computational interpretation expensive: the particular formalization of Constructive Picard-Lindelöf cited in our proofs (Makarov & Spitters, 2013) reported that the code they extracted for their experiments was only efficient up to 2 digits of precision, both since the Picard iteration which underlies their computation may take many iterations to converge for high (global) Lipschitz constants and since naïve implementations of computable reals can be very slow.

As with many programming languages, strategies could be executed either by interpretation or compilation. The big-step semantics play are best understood as a high-level interpreter for competing constructive strategies, whereas synthesis can be understood as a compilation step that commits to one strategy for Angel and allows classical blackbox Demon implementations for an external environment. When we do extract and execute strategies in Chapter 8, we will use ideas from both the play semantics and the realizability semantics of Chapter 4: the play semantics inspire the interface and recursive structure of the interpreter function, while realizability semantics inspire the treatment of game state.

In the type-theoretic sense, strategies for Angelic game operators are positive and thus can be synthesized by extracting witnesses from each introduction rule. While our ODE proof rules focus on Demonic proofs, the Angelic ODE solution rule contains a witness for the duration[17] of the ODE, just as other Angelic introduction rules contain witnesses. In Demonic game operators, invariants and test conditions simply describe the *range* of allowed observable behaviors, thus Demonic strategies can be understood as negative: it suffices to monitor whether Demon's behavior complies with invariants and test conditions extracted from the proof. Thus, while our play function plays two constructive strategies against each other for the sake of simplicity, the negative nature of Demon would make a synthesis approach equally justified in allowing a blackbox, untrusted implementation of an environment controlled by Demon, while concrete verified control code is synthesized for the Angel player's strategy.

[17]Because the solution rule is only applicable under the side condition that a closed-form solution exists, the side condition implies existence of a witness for the solution as well.

Chapter 6

Refining Constructive Hybrid Games

In Chapter 5, we developed a solid foundation for the verification of constructive hybrid games. Advantages of constructive hybrid games include concise controller models, the ability to extract control code from liveness proofs, and the ability to elegantly express reach-avoid properties that capture both safety and liveness. However, the state of the art for systems verification and synthesis is more mature than for games, classical VeriPhy included (Chapter 3). To lay a foundation that will help the state of the art for games catch up with systems, this chapter studies the relationship between hybrid games and systems. While the language of games is provably more powerful than the language of systems (Platzer, 2015a), we will show how *proofs* of game properties bridge the expressiveness gap: games are more powerful than systems in that game theorem statements are quantified over the existence of a strategy, but a CdGL proof is such a strategy, so that playing a game in accordance with a given strategy behaves like a system.

Connecting games to systems via their proofs suggests an approach for the implementation of synthesis for games in Chapter 8: a first pass of our synthesis tool will consume a game proof to produce a system where Angel plays the specific strategy represented in the proof while Demon plays adversarially. Such an approach provides a clear separation of concerns between processing of game proofs and code generation from systems. The fact that systems can be generated from game proofs does not, however, make game logic redundant with the logic of hybrid systems. The system corresponding to a strategy of a game can be unboundedly more complex than the game model, so games retain an important advantage in providing concise formal models that are easier to read and trust. Games also retain their advantage in expressing reach-avoid properties: while the work of this chapter does transfer guarantees from games to the systems implementing their strategies, the transferred guarantee is a *safety* guarantee showing that the final state arising from each Demon strategy, and thus the final state of the system, satisfies a given postcondition which was proved for the given game (likewise for Angel strategies). Regarding liveness, the system itself can be read as a witness of the game's liveness theorems (if any): it represents the Angel strategy[1], which exists, which achieves a postcondition regardless of Demon's strategy. We show (Theorem 6.4) that all safety properties of the system are

[1]Because Angel's strategy can, for example, control the duration of Angelic loops and ODEs, strategy existence subsumes standard notions of liveness such as "there exists a duration by which a goal is reached."

safety properties of the game, implying that the system reaches all states reached when playing the strategy. Thus, liveness guarantees transfer from the game to the system, a fact that is useful for synthesis of concrete controllers (Chapter 8).

In short, games provide a very direct treatment of reach-avoid properties whereas a careful reading is required to understand representations of reach-avoid properties in the context of systems. For the Logic-User, it is preferable to work in the world of games where such properties can be treated directly.

To study the gap between games and systems, this chapter develops a CdGL refinement calculus so we can show refinement properties between games and the systems we will extract from their proofs. The CdGL refinement calculus is of independent interest because refinement and equivalence reasoning have repeatedly proven fruitful for programs, including CPS models:

- Equivalences of programs are the fundamental building block for KAT (Kozen, 1997).
- Differential refinement logic dRL (Loos & Platzer, 2016) has reduced the human labor required for verification of classical hybrid systems by mixing use of refinement properties and dL modalities, i.e., refinement can reduce a theorem about a complex system to a theorem about a system whose proof is easier.
- Differential game refinements in dGL (Platzer, 2017b) provide relational reasoning for *differential* games with continuous competition (as opposed to hybrid games).

Compared to refinement of systems, the main added challenge of game refinement is that games are subnormal and subregular (Hughes & Cresswell, 1996)[2] and a refinement calculus for games should, where possible, seek refinement rules which are sound for all games in the absence of systems-specific axioms. Likewise, identifying the right semantics for refinement of games requires careful attention because standard definitions of refinement for systems (Loos & Platzer, 2016) rely on Kripke semantics, but subnormality and subregularity imply that vanilla Kripke semantics are insufficient for games. Because this chapter builds on the type-theoretic region-based semantics of CdGL, it will identify a novel (constructive) region-based semantics for refinement. A realizability semantics per CGL could allow the addition of a refinement operator too, but even then refinement would require subtleties not present in the case of systems, because a realizability semantics for refinement would have to characterize the fact that refinement relations can hold between two syntactically distinct games whose realizers thus have entirely different shapes.

Once equipped with CdGL refinements, we bridge the gap between hybrid games and systems by defining an operation we call *reification*. The reification operation takes as its input a hybrid game α, its correctness condition ϕ, and a CdGL proof that α satisfies ϕ. The reified output is a hybrid *system* which implements the strategy expressed in the constructive correctness proof. Refinement allows us to prove the relationship between a game and its reified system: the output system refines the input game so that *every* safety theorem of the output is a theorem of the input. Conversely, the output system also satisfies ϕ. To our knowledge, prior works (Back & von Wright, 1998) only feature ad-hoc discussions of reification; we give the first rigorous algorithm and correctness theorems.

[2]Meaning for example that they do not support Kripke's axiom K or Gödel's generalization rule

It may be surprising that game strategies can be reduced to systems, because games are known (Platzer, 2015a) to be more expressive than systems. Our result does not contradict this fact: only once a winning strategy is known can we bridge this expressiveness gap. Reification can be applied in several ways:

i) When synthesizing controllers and monitors for games, we will first extract a system from a game, then synthesize the system (Chapter 8).

ii) When we develop structured proofs for hybrid games (Chapter 7), each proof will be structured as a *strategy*, or *program*, which is annotated with proof steps. To recover a theorem for a (properly two-player) game model, a refinement is established between the game and its strategy.

iii) Experience with dRL (Loos & Platzer, 2016) suggests that refinement-based proofs, in combination with proofs of modal formulas, may improve productivity in general.

iv) Reification and refinement give an intensional view of strategy equality: two strategies are "the same" if their reification produces equivalent systems.

v) Refinement may enable comparing the efficacy and conservativity of two controllers: can one controller reach a broader range of states in the same time? For the sake of explaining one proposed approach, consider two games α and β which apply different controllers under the same physical model of 1D driving up to the same time limit t. If α and β both refine each other (i.e., they are equivalent), then neither is more efficient. But if one strictly refines the other, it means that one can reach strictly more states, meaning it either reaches more distant states (it goes faster) or it reaches states that are nearby but conservatively left out of the other model (less conservative, even if not strictly faster).

In Section 6.1, we discuss related work specific to refinement. In Section 6.2, we add a refinement connective to CdGL. In Section 6.3, we generalize the CdGL semantics to support refinement. In Section 6.4, we give a calculus for CdGL refinements. In Section 6.5, we discuss theoretical results about soundness and reification.

6.1 Related Work

Refinement calculi have been studied extensively. Prior work (Back & von Wright, 1998) has given examples of how programs can be refined from games using a formal calculus of refinement rules, but has not given an explicit reification algorithm let alone its correctness theorems. We give an explicit *reification* algorithm and prove that it captures the winning strategy of a game in a system. As with Chapter 5, our proofs are not formalized because the most obvious formalization would rely on features which are unsupported or experimental in prominent proof assistants such as Coq. Beyond the features used in Chapter 5, we make significant use of universe polymorphism and even universe-polymorphic recursion. The following paragraph contains technical discussion of universes; see Section 6.3 for more information.

We believe that a Coq formalization of our semantics and our theorems is possible, but that it is best left as future work because working around the limits of Coq would require

significant work, work which would distract from the story this chapter seeks to tell. For a given universe $\mathbb{T}_i$, Coq is capable[3] of formalizing and reasoning about the fragment of our semantics that belongs to $\mathbb{T}_i$. To provide a comprehensive treatment of our refinement calculus in Coq, one would write a code generator which determines a sufficiently large universe for a given refinement proof and then generates a Coq formalization which uses the appropriate universe. In short, a Coq formalization is not absent from this chapter because Coq disagrees with us regarding the soundness of our proof principles, but rather it is absent because such a formalization would require investing significant development effort into a code-generation approach. That requirement arises from the experimental nature of Coq's implementation of universe polymorphism.

We build directly on classical refinement reasoning for hybrid systems from dRL (Loos & Platzer, 2016) and also on (propositional, discrete) game algebra (Goranko, 2003). The game equivalence judgment and rules used in game algebra only consider global equivalences of games as opposed to equivalences which only hold conditionally under a particular context. Contextual reasoning is uniquely challenging for games, which are subnormal. The logic dRL supports contextual reasoning but not games. Our refinement calculus subsumes both game algebra and dRL by mixing games rules with contextual rules for systems. Even for rules which look the same as prior work, our constructive semantics demand novel soundness proofs.

Event-B (Abrial, 2010) uses refinement for practical verification by verifying simpler models, then refining them to more complex models. Their definition of refinement includes a more aggressive treatment of ghost variables than our system does: relational arguments between ghost and non-ghost state allow refinement properties to hold between systems with differing sets of state variables. Because our system fixes the dimension of the state, our proofs sometimes require dummy assignments that are not necessary in Event-B's treatment. The tradeoff is that our definition of refinement formulas is simple as a result.

Extensions of Event-B have been proposed for hybrid systems (Banach et al., 2015; Su et al., 2014; Banach, 2013; Dupont et al., 2018), of which at least one has been implemented (Dupont et al., 2018), but the extensions do not support hybrid games. Overall, Event-B holds promise as a platform for development of new refinement reasoning technology for CPS, but that fact does not contradict the importance of ensuring that the refinement of hybrid systems (and games) is soundly supported by solid semantic foundations, nor the viability of other implementation approaches. Notably, the dL family of logics and their KeYmaera X implementation have provided extensive experience in the use of real-arithmetic decision procedures and the automated foundational verification of ODE invariants. Any implementation in a new platform, whether the existing Event-B implementation (Dupont et al., 2018) or the Kaisar language developed later in this thesis (Chapter 7), has to confront the practical implementation challenges of arithmetic reasoning and invariant reasoning for itself. Because our rules are descendants of the dL rules, our treatment of ODEs is perhaps more easily developed than in Event-B, while our reliance on constructive arithmetic requires greater care in using arithmetic solvers than in any classical implementation.

[3]Once the additional formalization challenges discussed in Chapter 5 have been addressed

6.2 Constructive Differential Game Logic

We extend the language of CdGL with refinement formulas and give an example game which we will refine to a system which plays a winning strategy of it.

6.2.1 Syntax

We conservatively extend CdGL as presented in Chapter 5: the term and game languages are unchanged, while we add one formula connective for refinement.

Definition 6.1 (CdGL Formulas)**.** The language of CdGL *formulas* ϕ, ψ, φ is extended by:

$$\phi \ ::= \ \cdots \mid \alpha \leq^i_{[]} \beta$$

Game refinements come in two standard (Goranko, 2003) kinds: Angelic and Demonic. *Demonic refinement* $\alpha \leq^i_{[]} \beta$ of *rank* i holds if for every ϕ with $\mathfrak{R}(\phi) \leq i$, Demonic winning strategies of $[\alpha]\phi$ can be mapped into winning strategies of $[\beta]\phi$. The mapping from $[\alpha]\phi$ into $[\beta]\phi$ is constructive, modeled by a constructive implication (semantically, by a function type): a refinement computes a strategy for β and is allowed to invoke[4] the α strategy in doing so. *Angelic refinement* $\alpha \leq^i_{\langle\rangle} \beta$ effectively maps Angelic winning strategies of $\langle\alpha\rangle\phi$ into winning strategies of $\langle\beta\rangle\phi$. Note the difference between Demonic and Angelic refinement carefully: Angelic refinement may be more familiar to readers, especially those which are familiar with dRL (Loos & Platzer, 2016), but we take the Demonic presentation as primary, in large part because the theorems we wish to prove in this chapter are Demonic. Angelic and Demonic refinement are interdefinable: we define $\alpha \leq^i_{\langle\rangle} \beta \equiv \alpha^d \leq^i_{[]} \beta^d$. If we had instead treated Angelic refinement as primary, we would define $\alpha \leq^i_{[]} \beta \equiv \alpha^d \leq^i_{\langle\rangle} \beta^d$.

To define refinements, we introduce the rank $\mathfrak{R}(\alpha$ or $\phi)$ of a game or formula, a technical device which represents the smallest predicative universe in which α has a semantics, see Section 6.3. You may wish to ignore rank on the first reading: it can be inferred automatically, and we write $\alpha \leq_{[]} \beta$ when rank is unimportant. The defined game skip is the trivial test $?true$.

We use the same terminology for players used in Chapter 4 and Chapter 5: Angel refers to a constructive player "we" control while Demon refers to a classical opponent. The difference between the diamond modality $\langle\alpha\rangle\phi$ and box modality $[\alpha]\phi$ is that a diamond modality indicates that Angel is next-to-move (we are currently in control) while the box indicates that Demon is next-to-move (we are currently not in control).

6.2.2 Example Game

We define another example game because it is especially suitable for a gentle introduction to refinements. Consider a *push-pull cart* (Platzer, 2018a) (PP) on a 1-dimensional playing

[4]More generally, all elimination principles corresponding to the type of the strategy can be used, such as case analysis and induction in the case of strategies defined by sum types or inductive type families.

field with boundaries $x_\ell \leq x \leq x_r$ where x is the position of the cart and $x_\ell < x_r$ holds strictly. The CdGL theorem statement (6.1) for PP will be a CdGL *box* modality, meaning that the first player to move is *Demon*, the classical player whom we do not control. The initial position is written x_0. The preconditions are in formula pre. Demon is at the left of the cart and Angel at its right. Each player chooses to pull or push the cart, then the (oversimplified) physics say velocity is proportional to the sum of forces. Physics can evolve so long as the boundary $x_\ell \leq x \leq x_r$ is respected, with duration chosen by Demon. The oversimplified physics of this example were chosen for the sake of exposition.

$$\begin{aligned}\mathsf{pre} &\equiv x_\ell < x_r \land x_\ell \leq x_0 = x \leq x_r\\ \mathsf{PP} &\equiv \{\{L := -1 \cup L := 1\};\\ &\qquad \{R := -1 \cup R := 1\}^d;\\ &\qquad \{x' = L + R \,\&\, x_\ell \leq x \leq x_r\}\}^*\end{aligned}$$

A simple safety theorem for the push-pull game says that Angel has a strategy to ensure position x remains constant ($x = x_0$) no matter how Demon plays:

$$\mathsf{pre} \to [\mathsf{PP}]x = x_0 \tag{6.1}$$

The winning strategy that proves (6.1) is a simple mirroring strategy: Angel observes Demon's choice of L and plays the opposite value of R so that $L + R = 0$. Because $L + R = 0$, the ODE simplifies to $x' = 0 \,\&\, x_\ell \leq x \leq x_r$, which has the trivial solution $x(t) = x(0)$ for all times $t \in \mathbb{R}_{\geq 0}$. Angel shows the safety theorem by replacing the ODE with its solution and observing that $x = x_0$ holds for all possible durations.

While the example can be verified using solutions, we give a refinement calculus (Section 6.4) for the full range of invariant reasoning provided in CdGL (Chapter 5), including *differential invariants* (Platzer, 2017a, 2010b, 2018a) and *differential ghosts* (Platzer, 2018a) Invariant reasoning is essential for verification of games whose ODEs have non-polynomial, even non-elementary solutions (such as the model presented in Section 3.7). Ghost reasoning allows proving differential invariants which are not inductive (Platzer & Tan, 2020), which cannot be proved otherwise (Platzer, 2012d).

In contrast to a safety theorem, a liveness theorem would be shown by a progress argument. Suppose that Angel could set $L = 2$ but Demon can only choose $R \in \{-1, 1\}$. Then Angel's liveness theorem might say she can achieve $x = x_r$ if she is allowed to choose the ODE duration, because the choice $L = 2$ ensures at least 1 unit of progress in x for each unit of time.

6.3 Type-Theoretic Semantics

We generalize the type-theoretic semantics of CdGL (Chapter 5). We define the semantics of the new refinement formulas $\alpha \leq^i_{[]} \beta$ and employ an infinite tower of type universes, in support of refinements. We first give our assumptions on the underlying type theory.

As before, we assume a Calculus of Inductive and Coinductive Constructions (CIC)-like type theory (Coquand & Huet, 1988; Coquand & Paulin, 1988; *Coq Proof Assistant*, 1989),

but we now use an infinite tower of cumulative predicative universes. We use a predicative universe because consistency and computability are more obvious in a predicative setting, especially considering that we must also assume large elimination in our semantic definitions. We write $\mathbb{T}_i$ for the ith predicative universe, and regions P, Q can now belong to any family $\mathcal{S} \Rightarrow \mathbb{T}_i$. As usual in infinite towers of (cumulative) predicative universes, the different universes $\mathbb{T}_i$ are distinguished from each other by the fact that $\mathbb{T}_i : \mathbb{T}_i$ does not hold but $\mathbb{T}_i : \mathbb{T}_{i+1}$ does hold for all i and, because the universes are cumulative, so does $\mathbb{T}_i : \mathbb{T}_j$ for all $j > i$.

The formula and game semantics are now indexed by universes:

$$\begin{aligned}
\ulcorner \phi \urcorner &: \mathcal{S} \Rightarrow \mathbb{T}_{\mathfrak{R}(\phi)} \\
\langle\!\langle \alpha \rangle\!\rangle &: (\mathcal{S} \Rightarrow \mathbb{T}_{\mathfrak{R}(\alpha)}) \Rightarrow (\mathcal{S} \Rightarrow \mathbb{T}_{\mathfrak{R}(\alpha)}) \\
[[\alpha]] &: (\mathcal{S} \Rightarrow \mathbb{T}_{\mathfrak{R}(\alpha)}) \Rightarrow (\mathcal{S} \Rightarrow \mathbb{T}_{\mathfrak{R}(\alpha)})
\end{aligned}$$

The semantic functions for vanilla CdGL formulas and games are defined as before, because the generality of type universes is only used for the semantics of refinement formulas:

$$\ulcorner \alpha \leq^i_{[]} \beta \urcorner\ s = \big(\Pi P : (\mathfrak{S} \Rightarrow \mathbb{T}_i).\ \big([[\alpha]]\ P\ s \Rightarrow [[\beta]]\ P\ s\big)\big)$$

Game α demonically refines β ($\alpha \leq_{[]} \beta$) from a state s if for *all* goal regions P there exists an effective mapping from Demonic strategies $[[\alpha]]\ P\ s$ to Demonic strategies $[[\beta]]\ P\ s$. That is, refinements may depend on the state (they are *local* or *contextual*), but must hold for all goal regions P, as refinements consider the general game form itself, not a game fixed to a particular postcondition. It is worth noting that our semantics for refinement differs from definitions that have been used, for example, in classical refinement calculi for hybrid systems (Loos & Platzer, 2016). Compared to systems calculi, the main semantic challenge is accounting for the richness of game logic, which is subnormal and subregular (Hughes & Cresswell, 1996), meaning for example that for a given game α, formula $[\alpha](\phi \wedge \psi)$ need not hold when both $[\alpha]\phi$ and $[\alpha]\psi$ do. Compared to classical calculi, the main semantic challenge is making refinement constructive. One way to understand our semantics of refinement is to read ahead to the refinement elimination rule R[·] in Section 6.4. As usual, (e.g., box) refinement elimination assumes a modal formula $[\alpha]\phi$ and refinement relation $\alpha \leq_{[]} \beta$ from which it concludes $[\beta]\phi$. In this light, we define refinement as the weakest (i.e., provable-most-often) connective for which the refinement elimination rule is sound.

Whenever α and β both have rank at most i, then $\ulcorner \alpha \leq^i_{[]} \beta \urcorner\ s : (\mathcal{S} \Rightarrow \mathbb{T}_{i+1})$. Because refinement formulas are first-class, type-theoretic quantifiers may appear nested and in arbitrary positions within a formula's interpretation as a type family, not necessarily prenex position. We ensure predicativity by requiring that refinements quantify only over postconditions of lower rank. Rank can be inferred in practice by inspecting a proof: each rank annotation need only be as large as the rank of every postcondition in every application of rules R⟨·⟩ and R[·] from Section 6.4.

The Angelic refinement operator $\alpha \leq_{\langle\rangle} \beta$ is definable from $\alpha \leq_{[]} \beta$ by the equivalence:

$$\alpha \leq_{\langle\rangle} \beta \equiv \alpha^d \leq_{[]} \beta^d$$

which is equivalent to the direct semantic definition

$$\ulcorner \alpha \leq^i_{\langle\rangle} \beta \urcorner\ s = \big(\Pi P : (\mathfrak{S} \Rightarrow \mathbb{T}_i).\ \big(\langle\!\langle \alpha \rangle\!\rangle\ P\ s \Rightarrow \langle\!\langle \beta \rangle\!\rangle\ P\ s\big)\big)$$

of Angelic refinements as functions which map Angelic strategies for α of type $\langle\!\langle \alpha \rangle\!\rangle\ P\ s$ constructively into $\langle\!\langle \beta \rangle\!\rangle\ P\ s$ for game β, for every goal region P and for given state s.

6.4 Refinement Proof Calculus

We give a natural deduction calculus for hybrid game refinements. As in vanilla CdGL, the rules feature a context Γ of CdGL formulas, which may now include refinement formulas. We reiterate that because we treat Demonic refinements $\alpha \leq^i_{[]} \beta$ as primary, we only present Demonic refinement rules here. Readers who are more familiar with Angelic refinement connectives such as the connective from dRL (Loos & Platzer, 2016) should pay special attention to the distinction between Demonic and Angelic refinements: the rules we present are dual to Angelic rules such as those of dRL. At the same time, no generality is lost by presenting the Demonic connective. As stated in Section 6.3, Angelic refinement $\alpha \leq^i_{\langle\rangle} \beta$ is definable from Demonic refinement: $\alpha \leq^i_{\langle\rangle} \beta \equiv \alpha^d \leq^i_{[]} \beta^d$. In particular, because our set of rules includes cases where α and β contain the duality operator, Angelic refinement proofs are readily supported by applying the Demonic refinement rules to the dual games.

Recall that because Demon moves first in a box modality, the Angel player in our Demonic refinements only has control (which is existential in nature) over connectives that appear under (an odd number of) dualities α^d. We write $\alpha \cong \beta$ for $\alpha \leq_{[]} \beta \wedge \beta \leq_{[]} \alpha$. Recall that hybrid *systems* are hybrid games which do not contain the dual operator α^d.

The refinement elimination rules $\mathrm{R}\langle\cdot\rangle$ and $\mathrm{R}[\cdot]$ say every true postcondition ϕ of a game α is a true postcondition of every β which α refines. The side condition for $\mathrm{R}\langle\cdot\rangle$ and $\mathrm{R}[\cdot]$ is that $\mathfrak{R}(\phi) \leq i$ where i is the rank annotation of the refinement. These are the only rules which care about rank, so ranks can be inferred from proofs by inspecting the uses of these rules. While rank is of little practical importance, our use of rank crucially enables our predicative formal foundations.

The refinement rules for discrete connectives are given in Fig. 6.1. Soundness of game refinement rules is subtle because games are subnormal; in particular, it is subtle to reason about games which contain, or behave like, sequential compositions. Sound game refinement rules for sequential composition reasoning can be divided into two classes: rules which refine games globally by requiring an empty context (rule ;G), and rules which restrict some subgames to be systems (rule ;S) to enable contextual reasoning. The former approach (rule ;G) follows game algebra (Goranko, 2003). Though it requires an empty context, rule ;G is crucial for sound reasoning about proper games, i.e., games which contain dualities. The latter approach (rule ;S) generalizes the dRL (Loos & Platzer, 2016) rule for contextual refinement of sequential compositions by allowing systems to refine proper games. By combining both approaches, our calculus proves all statements that are provable in either of game algebra or dRL while enabling proofs of additional statements by overcoming the challenges of sound reasoning about proper games.

$$(\mathrm{R}\langle\cdot\rangle)\quad \frac{\Gamma \vdash \langle\alpha\rangle\phi \quad \Gamma \vdash \alpha \leq^{i}_{\langle\rangle} \beta}{\Gamma \vdash \langle\beta\rangle\phi}\ {}^{1} \qquad (\langle?\rangle)\quad \frac{\Gamma \vdash \phi \to \psi}{\Gamma \vdash ?\phi^d \leq_{[]} ?\psi^d}$$

$$(\mathrm{R}[\cdot])\quad \frac{\Gamma \vdash [\alpha]\phi \quad \Gamma \vdash \alpha \leq^{i}_{[]} \beta}{\Gamma \vdash [\beta]\phi}\ {}^{1} \qquad ([?])\quad \frac{\Gamma \vdash \psi \to \phi}{\Gamma \vdash ?\phi \leq_{[]} ?\psi}$$

$$(;\mathrm{S})\quad \frac{\Gamma \vdash \boldsymbol{\alpha_1} \leq_{[]} \alpha_2 \quad \Gamma \vdash [\boldsymbol{\alpha_1}]\beta_1 \leq_{[]} \beta_2}{\Gamma \vdash \boldsymbol{\alpha_1}; \beta_1 \leq_{[]} \alpha_2; \beta_2}\ {}^{2} \qquad (\langle\cup\rangle\mathrm{L})\quad \frac{\Gamma \vdash \alpha^d \leq_{[]} \gamma \quad \Gamma \vdash \beta^d \leq_{[]} \gamma}{\Gamma \vdash \{\alpha \cup \beta\}^d \leq_{[]} \gamma}$$

$$(;\mathrm{G})\quad \frac{\Gamma \vdash \alpha_1 \leq_{[]} \alpha_2 \quad \cdot \vdash \beta_1 \leq_{[]} \beta_2}{\Gamma \vdash \alpha_1; \beta_1 \leq_{[]} \alpha_2; \beta_2}\ {}^{3} \qquad ([\cup]\mathrm{R})\quad \frac{\Gamma \vdash \alpha \leq_{[]} \beta \quad \Gamma \vdash \alpha \leq_{[]} \gamma}{\Gamma \vdash \alpha \leq_{[]} \beta \cup \gamma}$$

$$([\cup]\mathrm{L1})\quad \Gamma \vdash \alpha \cup \beta \leq_{[]} \alpha \qquad (\langle\cup\rangle\mathrm{R1})\quad \Gamma \vdash \alpha^d \leq_{[]} \{\alpha \cup \beta\}^d$$

$$([\cup]\mathrm{L2})\quad \Gamma \vdash \alpha \cup \beta \leq_{[]} \beta \qquad (\langle\cup\rangle\mathrm{R2})\quad \Gamma \vdash \beta^d \leq_{[]} \{\alpha \cup \beta\}^d$$

$$(\langle{:}*\rangle)\quad \Gamma \vdash \{x := f\}^d \leq_{[]} \{x := *\}^d \qquad (\mathrm{un}*)\quad \frac{\Gamma \vdash [\boldsymbol{\alpha}^*](\boldsymbol{\alpha} \leq_{[]} \beta)}{\Gamma \vdash \boldsymbol{\alpha}^* \leq_{[]} \beta^*}\ {}^{3}$$

$$([{:}*])\quad \Gamma \vdash x := * \leq_{[]} x := f \qquad (\mathrm{roll}_l)\quad \Gamma \vdash \mathsf{skip} \cup \{\alpha; \alpha^*\} \cong \alpha^*\ {}^{4}$$

$$(\mathrm{skip}^d)\quad \mathsf{skip}^d \cong \mathsf{skip} \qquad (;^d)\quad \{\alpha;\beta\}^d \cong \alpha^d; \beta^d \qquad (:=^d)\quad x := f^d \cong x := f \qquad (\mathrm{DDE})\quad \{\alpha^d\}^d \cong \alpha$$

[1] Assuming $\mathfrak{R}(\phi) \leq i$
[2] $\boldsymbol{\alpha_1}$ respectively $\boldsymbol{\alpha}$ is a hybrid system as indicated by bold font
[3] Where $\cdot$ indicates an empty context
[4] Recall that $\mathsf{skip} \equiv ?\mathit{true}$ by definition

Figure 6.1: Refinement of discrete connectives.

When expressing rules whose soundness require certain games to be systems, we write bold variables where systems are required, e.g., $\boldsymbol{\alpha}$ and $\boldsymbol{\alpha_1}$ in rules ;S and un$*$. One sequence refines another piecewise in the ;S rule, which is *contextual*: refinement of the second component exploits the fact that the first component has been executed. The ability to reason in context in rule ;S is useful, but the support for contexts is responsible for the restriction to systems. Because refining compositions of non-system games is also useful, we provide a second rule which instead sacrifices contexts. Rule ;G is a variant of rule ;S which says α_1 can be any game, but only if $\beta_1 \leq_{[]} \beta_2$ holds in the empty context, which is essential for soundness. System $\boldsymbol{\alpha_1}$ in the second premise of rule ;S could soundly be α_2, but $[\boldsymbol{\alpha_1}]$ is often more convenient in practice because it is a *system* modality, thus normal.

Rules $\langle ? \rangle$ and $[?]$ refine tests by respectively weakening or strengthening test conditions: if Angel passes a stronger test, she passes a weaker one, whereas if Demon provides Angel with a proof of a weaker condition, providing a stronger one would only make Angel's life easier. The left and right rules for choices are dual to one another. Rules $\langle \cup \rangle$R1 and $\langle \cup \rangle$R2 say each branch refines an Angelic choice (i.e., Angel has the freedom to choose which branch), while $[\cup]$R says a Demonic choice is refined by refining both branches (because Angel does not choose which branch Demon takes). Rules $\langle :* \rangle$ and $[:*]$ say that deterministic assignments refine nondeterministic ones: if assigning x the value of a particular term f makes some postcondition true, there certainly exists a value of x which makes it true. Dually, if all values of x make a postcondition true, so does the particular value yielded by f. Rule un$*$ compares loops by comparing their bodies, but the refinement of the bodies must hold no matter how many repetitions have already occurred. The other loop reasoning principle, rule roll_l, allows unrolling an execution of a loop in a refinement proof. Rules skip^d, $:=^d$, and $;^d$ say skip and $x := f$ are self-dual and the dual of a sequence is a sequence of duals. Double duals cancel by rule DDE. Because Angelic refinement is defined as a Demonic refinement of dual games, rules for the refinement of dual games can also be used for (non-dualized) Angelic refinement. For example, rule $\langle :* \rangle$ proves $x := f \leq_{\langle\rangle} x := *$ for all x and f, i.e., expanding the definition of $\cdot \leq_{\langle\rangle} \cdot$ in the judgement $\Gamma \vdash x := f \leq_{\langle\rangle} x := *$ yields the exact text of the rule $\langle :* \rangle$.

The rules in Fig. 6.2 are algebraic properties which will be used in the proof of Theorem 6.4. These rules generalize known game equalities (Goranko, 2003) to refinement. The calculus we present is not intended to be minimal (nor include every known equality (Goranko, 2003)). For example, rules $[\cup]$L1 and $[\cup]$L2 are interderivable using the commutativity rule $\cup$c, but all are presented because all are useful both in actual proofs and for the purpose of exposition. Our rules are primarily straightforward generalizations of those from dRL (Loos & Platzer, 2016) by relaxing the language from systems to games. The exceptions are the sequential composition and looping rules, whose soundness requires special attention in the setting of games, specifically requires restricting some arguments to *systems* as were used in dRL. Several rules of dRL, such as the DR and MDF rules (Loos & Platzer, 2016) for implicational domain constraint reasoning and time stretching, do not have counterparts in our calculus. We have no reason to doubt that sound constructive counterparts exist, but have simply not yet needed those rules because of our present focus on reification (Section 6.5.2).

Rules refl and trans say refinement is a partial order. Sequential composition has

$$\text{(trans)}\quad \frac{\Gamma \vdash \alpha \leq_{[]} \beta \quad \Gamma \vdash \beta \leq_{[]} \gamma}{\Gamma \vdash \alpha \leq_{[]} \gamma}$$

(refl)	$\Gamma \vdash \alpha \leq_{[]} \alpha$	($;\mathrm{d}_r$)	$\Gamma \vdash \{\alpha \cup \beta\}; \gamma \cong \{\alpha; \gamma\} \cup \{\beta; \gamma\}$
($;\mathrm{id}_l$)	$\Gamma \vdash \{\mathsf{skip}; \alpha\} \cong \alpha$	($;$A)	$\Gamma \vdash \{\alpha; \beta\}; \gamma \cong \alpha; \{\beta; \gamma\}$
($;\mathrm{id}_r$)	$\Gamma \vdash \{\alpha; \mathsf{skip}\} \cong \alpha$	($:=:=$)	$\Gamma \vdash x := f; x := g \cong x := g$ [2]
(annih_l)	$\Gamma \vdash ?\mathit{false}; \alpha \cong ?\mathit{false}$	($\cup$A)	$\Gamma \vdash \{\alpha \cup \beta\} \cup \gamma \cong \alpha \cup \{\beta \cup \gamma\}$
($:=$nop)	$\Gamma \vdash \{x := x\} \cong \mathsf{skip}$ [1]	($\cup$c)	$\Gamma \vdash \alpha \cup \beta \cong \beta \cup \alpha$
		($\cup$idem)	$\Gamma \vdash \alpha \cup \alpha \cong \alpha$

[1] Recall that $\mathsf{skip} \equiv ?\mathit{true}$ by definition
[2] for $x \notin \mathrm{FV}(g)$

Figure 6.2: Algebraic rules.

$$\text{(DC)}\quad \frac{\Gamma \vdash [x' = f \,\&\, \phi]\psi}{\Gamma \vdash \{x' = f \,\&\, \phi\} \cong \{x' = f \,\&\, \phi \wedge \psi\}}$$

$$\text{(DW)}\quad \Gamma \vdash \{x := *; x' := f; ?\psi\} \leq_{[]} \{x' = f \,\&\, \psi\}$$

$$\text{(solve)}\quad \frac{\Gamma \vdash [t := *; ?0 \leq t \leq d; x := sln]\psi}{\Gamma, t = 0, d \geq 0 \vdash \{t := d; x := sln; t' := 1; x' := f\} \leq_{[]} \{t' = 1, x' = f \,\&\, \psi\}^d}$$ [1]

$$\text{(DG)}\quad \Gamma \vdash \{y := f_0; x' = f, y' = a(x)y + b(x) \,\&\, \psi\} \leq_{[]} \{x' = f \,\&\, \psi; \{y := *; y' := *\}^d\}$$ [2]

[1] sln solves ODE on $[0, d]$ and $\{t, t', x, x'\} \cap FV(d) = \emptyset$. Note that d is a term argument to the rule schema, not a program variable.
[2] $y \notin \mathrm{FV}(\Gamma) \cup \mathrm{FV}(f_0) \cup \mathrm{FV}(f) \cup \mathrm{FV}(a) \cup \mathrm{FV}(b) \cup \mathrm{FV}(\psi) \cup \{x\}$

Figure 6.3: Differential equation refinements.

identities (rules $;\mathrm{id}_l$ and $;\mathrm{id}_r$). Rule $:=:=$ deduplicates a double assignment if the first assignment does not influence the second: $\mathrm{FV}(f)$ are the *free variables* mentioned in f. Choice (rules $\cup$A) and sequence (rule ;A) are associative, and choice is commutative (rule $\cup$c) and idempotent ($\cup$idem), while sequential composition is right-distributive (rule $;\mathrm{d}_r$). Impossible tests can annihilate any following program (rule annih_l). Assigning a variable to itself is a no-op (rule $:=$nop).

Fig. 6.3 gives the ODE refinement rules. *Differential cut* rule DC says the domain constraints ϕ and $\phi \wedge \psi$ are equivalent if ψ holds as a postcondition under domain constraint ϕ. *Differential weakening* rule DW says an ODE is overapproximated by the program which assumes only the domain constraint. *Differential solution* rule solve says that a solvable Angelic ODE $x' = f \,\&\, \psi$, whose syntactic solution term is sln, is refined by a deterministic program which assigns the solution to x after some duration throughout which the domain constraint holds, specified by a duration term d which is constant throughout the ODE.

Here $sln = (\lambda s : \mathcal{S}.\ (sol\ (s\ t)))$ is the term corresponding to the semantic solution sol at time t. In the definition of sln, recall that the notation $s\ t$ is used to retrieve the value of variable t in state s. *Differential ghost* rule DG soundly augments an ODE with a fresh dimension y so long as the solution for y exists as long as that of x, and is known (Platzer & Tan, 2020) to enable proofs of otherwise unprovable (Platzer, 2012d) properties. The right-hand side for y is required to be linear in y because this suffices to ensure duration preservation as required for soundness. Rule DG is not an equivalence because a nondeterministic assignment could choose to assign values of y and y' that are not in the trajectory of a given linear ODE (Platzer, 2008a).

The nondeterministic assignments introduced in DG are necessary for a subtle reason: a CdGL refinement $\alpha \leq_{[]} \beta$ holds if *every* provable box property of α is a provable box property of β, *including* properties which mention variables y or y' that are not mentioned in β. Because α *does* mention and modify y and y', we can only ensure refinement by allowing β to set arbitrary values for y and y', so that refinement holds even for properties which depend on the value of y and y' in α. The subtle interaction between ghosts and refinement is not unique to CdGL but must be addressed in any refinement calculus which features ghosts. The subtle interaction between ghosts and refinement will also inform the design of the Kaisar proof language in Chapter 7: by separating a game proof into a hybrid system proof and a refinement proof, ghost reasoning can be confined to the hybrid system portion of the proof, so that refinements and ghosts never need to interact.

6.5 Theory

We develop theoretical results about CdGL refinements: soundness and the relationship between games and systems. Proofs are in Appendix C.1.

6.5.1 Soundness

The *sine qua non* condition of any logic is soundness. We show that every formula provable in the CdGL refinement calculus is true in the type-theoretic semantics.

Theorem 6.1 (Soundness of Proof Calculus)**.** *If sequent $\Gamma \vdash \phi$ is provable in CdGL with refinement then $\Gamma \vdash \phi$ is valid in CdGL with refinement. As a special case, if $(\cdot \vdash \phi)$ is provable, then ϕ is valid.*

The proof of soundness relies on standard language properties such as coincidence, bound effect, renaming, and substitution. Those language properties were all proved inductively for vanilla CdGL (Chapter 5). For constructs introduced in Chapter 5, the semantics of Chapter 5 and Chapter 6 differ only by a universal quantification over type universes, so the proof cases from Chapter 5 transfer trivially. Thus, the proofs of each language property for CdGL with refinement (Appendix C.1) merely add cases for refinement formulas within the inductive proofs of Chapter 5.

The main soundness proof then shows soundness of each refinement rule. Because many of the refinement rules capture reasoning principles analogous to those of some vanilla

CdGL rule, many cases of the soundness proof are analogous to cases from the vanilla CdGL soundness proof.

6.5.2 Reification

A game α describes what actions are allowed for each player but not how Angel selects among them given an adversarial Demon. Every game modality proof, whether of $[\alpha]\phi$ or $\langle\alpha\rangle\phi$, describes Angel's strategy to achieve a given postcondition ϕ. Once Angel's strategy has been fixed, Demon is allowed to make arbitrary (universally-quantified) moves so long as he obeys the rules of the game. Whereas a given game can contain both Angelic and Demonic choices, control of all choices in a *system* is always uniformly given to *one* of Angel or Demon: modality $[\boldsymbol{\alpha}]\phi$ treats a system $\boldsymbol{\alpha}$ as Demonic while $\langle\boldsymbol{\alpha}\rangle\phi$ treats a system as Angelic.

A folklore theorem describes the relation between hybrid games and hybrid systems: given a proof (winning strategy) for a hybrid game, one can *reify* Angel's strategy to produce a hybrid *system* which implements that strategy. Constructivity of CdGL helps us make the folklore theorem a reality because it makes the computational content of Angel's strategy readily accessible for reification. Since Demonic choices survive reification but Angelic ones do not, we will find it simplest to work with Demonic game modalities $[\alpha]\phi$ here, but every Angelic game modality $\langle\alpha\rangle\phi$ could equivalently be expressed as $[\alpha^d]\phi$. Additionally, CdGL rules for diamond modalities $\langle\alpha\rangle\phi$ are written equivalently with $[\alpha^d]\phi$ since our theorems in this section are phrased in terms of boxes.

As discussed in the introduction of this chapter, a formal reification operator connecting (proved) games and systems could both lay the foundation for tooling for CdGL and formalize the folklore concept that proving a game is like finding a strategy and proving that the strategy wins the game. In this section, we formally define the reification operation and prove its relation to the source game using refinements and a derivation $\mathcal{A}$ in the CdGL (non-refinement) proof calculus. CdGL (non-refinement) proofs are as in Chapter 5; the proof calculus that was presented in Chapter 5 was already optimized to work well in combination with reification.

There are several technical challenges in developing the reification operator and its metatheory. A wide range of refinement reasoning is exercised in the metatheoretic proofs because reasoning about reification requires reasoning in context, reasoning algebraically, and not only reasoning about both systems and games but relating games controlled by Angel to systems controlled by Demon which implement a specific Angel strategy. Another technical subtlety is posed by the existence of Demonic tests of game modality formulas: the CdGL proof calculus allows proofs of formulas such as $[?(\langle\alpha\rangle\phi);\{\alpha\}^d]\phi$ where Angel plays a strategy which was dynamically provided to her by Demon. There is no obvious reification operation which would remove the test that mentions game α, so our reification operation imposes a restriction on tests and contexts instead, which we will call the *system-test* requirement (Def. 6.2).

Let $\mathcal{A}$ be a CdGL proof of some CdGL formula $[\alpha]\phi$ in context Γ, i.e., let $\Gamma \vdash \mathcal{A}\colon [\alpha]\phi$. We then write $\mathcal{A} \rightsquigarrow_\alpha \boldsymbol{\alpha}$ to say the (unique) result of reifying the strategy given by $\mathcal{A}$ into hybrid game α is the hybrid *system* $\boldsymbol{\alpha}$. The system $\boldsymbol{\alpha}$ needs to commit to Angel's strategy

according to $\mathcal{A}$ while retaining *all* available choices of Demon, else the output system could not be successfully "played" against all Demon opponents permitted by the input game. What properties ought $\boldsymbol{\alpha}$ satisfy?

Committing to a safe Angel strategy should never make the system less safe. The safety postcondition ϕ should *transfer* to $\boldsymbol{\alpha}$, i.e., the following property should hold:

$$\text{If } \Gamma \vdash \mathcal{A} \colon [\alpha]\phi \text{ and } \mathcal{A} \rightsquigarrow_\alpha \boldsymbol{\alpha} \text{ then } (\Gamma \vdash [\boldsymbol{\alpha}]\phi) \text{ is provable.}$$

Transfer alone does not capture reification, e.g., defining $\boldsymbol{\alpha} = \ ?\mathit{false}$ for all α and $\mathcal{A}$ would vacuously satisfy the transfer property but would certainly not capture the strategic meaning of the proof $\mathcal{A}$.

We, thus, guarantee a converse direction. The reified hybrid system $\boldsymbol{\alpha}$ is a *safety refinement* of hybrid game α, so *every* postcondition ψ satisfying $[\boldsymbol{\alpha}]\psi$ also satisfies $[\alpha]\psi$:

$$\text{If } \Gamma \vdash \mathcal{A} \colon [\alpha]\phi \text{ and } \mathcal{A} \rightsquigarrow_\alpha \boldsymbol{\alpha} \text{ then } (\Gamma \vdash \boldsymbol{\alpha} \leq_{[]} \alpha) \text{ is provable.}$$

Intuitively, $[\boldsymbol{\alpha}]\psi$ says postcondition ψ holds for every Demon behavior of $\boldsymbol{\alpha}$, while $[\alpha]\psi$ holds if there *exists* an Angel strategy that ensures ψ for every Demon behavior of α. Since derivation $\mathcal{A}$ is designed to satisfy ψ, there certainly exists a strategy that satisfies ψ. Refinement captures the notion that Angelic choices in $\boldsymbol{\alpha}$ are made more strictly than in α, while Demonic choices are only made more loosely.

Even transfer *and* refinement do not fully validate the reification operation, since defining $\boldsymbol{\alpha} = \alpha$ suffices to ensure both. This leads to a third, most obvious property: $\boldsymbol{\alpha}$ must be a system when α is a game. Not only are systemhood, transfer, and refinement all desirable properties for reification, but their combination is an appealing specification because there is no trivial operation which satisfies all three. If the above three properties hold, they also imply a sound version of the normal modal logic axiom K that is elusive in games: If $\Gamma \vdash \mathcal{A} : [\alpha]\phi$ and $\Gamma \vdash [\boldsymbol{\alpha}]\psi$ is provable for $\mathcal{A} \rightsquigarrow_\alpha \boldsymbol{\alpha}$, then $\Gamma \vdash [\alpha](\phi \wedge \psi)$ is provable. Additionally, transfer and systemhood suggest that game synthesis can "export" a game proof to a systems proof, then apply synthesis to the resulting system. We discuss some technicalities first.

Technicalities. Reification $\cdot \rightsquigarrow \cdot$ accepts a CdGL proof and game, and returns a system. It is defined by induction on (normal, system-test) CdGL derivations. In each case, we write $\mathcal{A}, \mathcal{B}, \mathcal{C}$ for the CdGL derivation of each premise (from left to right) and $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}$ for corresponding output systems such that $\mathcal{A} \rightsquigarrow_\alpha \boldsymbol{\alpha}, \mathcal{B} \rightsquigarrow_\beta \boldsymbol{\beta}, \mathcal{C} \rightsquigarrow_\gamma \boldsymbol{\gamma}$ where α, β, γ are the games appearing in the modal formula proved by each premise.

The sequential composition case is one the most subtle; we preview main subtleties here and discuss them further in those cases and their proofs. In dynamic logic, modality $[\alpha;\beta]\phi$ is typically proved by reducing it to $[\alpha][\beta]\phi$, so care is needed if our caller wants us to reify only the strategy of α. Two features of our definition address sequential composition. Firstly, we identify $[\alpha;\beta]\phi$ with $[\alpha][\beta]\phi$, i.e., we apply $\langle\!|;|\!\rangle$I implicitly because we use it so often. Secondly, the game argument α in $\cdot \rightsquigarrow_\alpha \cdot$ tracks the game α whose strategy will be reified. In the sequential case, we update the game argument to remember to reify β once α is done. Regardless, the definition of $\mathcal{A} \rightsquigarrow_\alpha \boldsymbol{\alpha}$ is by induction on the derivation $\mathcal{A}$, not

game α. The argument α is only used for early termination: if $\mathcal{A}$ proves $[\alpha][\beta]\phi$ but α is passed for the game argument, only the reification for α is output. When $\mathcal{A}$ is a proof of a non-modal formula such as a comparison formula $f > g$, reification returns skip, which is a no-op statement ($?true$).

Angelic modalities $\langle\alpha\rangle\phi$ are represented by duality $[\alpha^d]\phi$. This style is interchangeable with normal-form CdGL proofs; we elide the duality ($[\alpha^d]\phi \leftrightarrow \langle\alpha\rangle\phi$) and skip ($[\mathsf{skip}]\phi \leftrightarrow \phi$) steps which convert between the two. Recall from Chapter 4 that case-analysis is sometimes *normal* (not reducible to a different proof term), despite not being *canonical* (an introduction rule) because case analyses which inspect the state cannot be resolved statically, thus are not reducible. Normal case analyses are analogous to case-tree normal forms in lambda calculi with coproducts (Altenkirch et al., 2001). Normal forms of (classical) ODE proofs have been characterized (Bohrer & Platzer, 2019). In our proofs, we rely on the fact that ODE proofs can be normalized so that rule DI never appears at the top level by moving applications of DI under DC. By analogy to its dual DI, the differential variant rule DV never appears at the top level[5] either. The normal form (Bohrer & Platzer, 2019) of an ODE proof optionally begins with a single block of ghost steps using rule DG. Any ghosts are followed by a block differential cuts using DC, each of which is proved by a differential invariant step with rule DI. The normal form proof ends with an application of differential weakening, i.e., it ends with an application of rule DW.

As alluded to in the introduction of the reification section (Section 6.5.2), reification of Angel's strategy is only possible[6] when Angel's strategy is statically knowable. Because the CdGL proof calculus allows *strong tests*, which can contain CdGL modalities, it allows Demon to pass a strategy to Angel as evidence for a test, which Angel could then play as part of her own strategy. We identify a fragment which rules out strategy-passing behavior and thus admits reification, which we call the *system-test* fragment. Our system-test fragment is weaker than strong-test CdGL but stronger than weak-test CdGL, i.e., the fragment of CdGL where tests contain no modalities.

Definition 6.2. System-test A formula is *system-test* if all modalities mentioned in it are box modalities $[\boldsymbol{\alpha}]\phi$ of systems $\boldsymbol{\alpha}$. The most restrictive *system-test* property is that for proofs. A proof is *system-test* if every context in its proof tree contains only system-test formulas. We call a game *system-test* if all the formulas appearing in its tests and domain constraints are system-test formulas.

In principle, any formula is permissible which is constructively equivalent to a box system modality, such as $\langle\boldsymbol{\alpha}^d\rangle\phi$. We exclude them from the definition solely to avoid confusion and reduce proof complexity. Note that negated box system modalities with negated postconditions such as $\neg[\boldsymbol{\alpha}]\neg\phi$ are acceptable despite their classical equivalence to the forbidden modality $\langle\alpha\rangle\phi$ because the equivalence does not hold constructively. A

[5]As of this writing, it will not appear in the normal fragment at all. In future work, reification of proofs using DV would be supported by developing the Angelic relative of DC in CdGL and requiring DV to appear under it. Angelic axioms analogous to DC have been developed for dL (Tan & Platzer, 2019, Axiom DR$\langle\cdot\rangle$), but have not been a priority in CdGL because the Demonic counterparts have historically received the most use.

[6]Assuming that our target language is the language of hybrid systems rather than a custom language with support for first-class strategies.

witness of $\neg[\boldsymbol{\alpha}]\neg\phi$ does not contain a witness of $\langle\alpha\rangle\phi$, but rather only provides the ability to derive contradiction from any proof of $[\boldsymbol{\alpha}]\neg\phi$. For the sake of defining the system-test fragment, we consider the propositional connectives distinct from game modalities despite the fact that we have treated the propositional connectives as defined operators, defined in terms of game modalities. Notably, we allow first-order arithmetic in tests.

Restricting reification to the system-test fragment ensures the reification of the hypothesis rule hyp is a system. In the proof rules that follow, $\cdot\frac{y}{x}$ is the transposition renaming of variable x to (usually fresh) variable y and vice-versa in a term, formula, game, or context.

Definitions. We define reification. Reification $\mathcal{A} \rightsquigarrow_\alpha \boldsymbol{\alpha}$ is defined inductively on the CdGL derivation $\mathcal{A}$, whose conclusion is the box CdGL modality $[\alpha]\phi$ for some CdGL formula ϕ. Within each case of the inductive definition, we let $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}$ respectively refer to the results of reifying the first, second, and third premises, read left-to-right, i.e., $\mathcal{A} \rightsquigarrow_\cdot \boldsymbol{\alpha}, \mathcal{B} \rightsquigarrow_\cdot \boldsymbol{\beta}$, and $\mathcal{C} \rightsquigarrow_\cdot \boldsymbol{\gamma}$. In each rule, the game argument $\cdot$ depends on[7] which game is proved by each premise, and is *not* always the corresponding game α, β, or γ, meaning there is also *not* always a direct relation between each game metavariable (e.g., α) and each bolded metavariable (e.g., $\boldsymbol{\alpha}$).

We find it helpful to assume, recursively, that the postcondition of the input proof is always of form $[L]\phi$ rather than simply ϕ. In the case for $\alpha;\beta$, game L will serve as a "stack" (mnemonically: "list") which remembers to reify β once the reification of α is done. In short[8], the modality L is understood as a stack (or "list") of remaining programs which are reified after reification of α completes.

In each case of reification, we write the proof rule with a postcondition $[L]\phi$ rather than writing the arbitrary postconditions ϕ that were presented in Chapter 5. The game argument $\cdot$ of the reification operation $\rightsquigarrow_\cdot$ will typically mention L as a subgame to indicate that L also has to be reified after reifying whatever game is currently in focus. By only writing the cases for postconditions of shape $[L]\phi$, we are only writing the inductive cases of the reification definition! In an abuse of notation, we note that the given cases subsume the base cases for non-modal formulas ϕ by letting $L = \mathsf{skip}$ so that $[L]\phi$ and ϕ are equivalent.

We first give the reifications of case-analysis and hypothesis proofs, the only two normal proofs which are not introduction forms. In the $\vee$E case, $\mathcal{A}$ proves a first-order disjunction $\varphi \vee \psi$, since proper choice game modalities $\langle\alpha \cup \beta\rangle$ are not permitted in system-test, normal-form proofs. Recall that normal-form proofs (Def. 4.15 in Section 4.9) are those which cannot be simplified further, while the definition of the system-test fragment only allows proper game choice modalities if they happen to get removed during normalization.

In rules with multiple premises, $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ stand for the reified systems for the second and third premises. For example, in $\vee$E, we have $\mathcal{B} \rightsquigarrow_{\alpha;L} \boldsymbol{\beta}$ where $\mathcal{B}$ is the proof of second premise $\Gamma, \varphi \vdash [\alpha][L]\phi$, likewise $\mathcal{C} \rightsquigarrow_{\alpha;L} \boldsymbol{\gamma}$ for the third premise $\Gamma, \psi \vdash [\alpha][L]\phi$. Note that the game argument $\cdot$ of the recursive calls to $\rightsquigarrow_\cdot$ is not written explicitly in our rule-based notation, but is discussed in the text in cases where it is particularly significant. In the

[7]The precise computation is given in the proofs (Appendix C.1) in the annotations on the recursive calls of reification, which are all written out in full.

[8]See the cases for $\alpha;\beta$ and the proofs (Appendix C.1) for more information.

case of $\vee$E, which is an elimination rule, the recursive calls happen to keep the original game argument $\alpha;L$. In most cases, which correspond to introduction rules, the recursive calls will use game arguments that are structurally simpler than in the outer call.

$$\text{hyp}\frac{(\text{if } [\alpha][L]\phi \in \Gamma)}{\Gamma \vdash [\alpha][L]\phi} \rightsquigarrow_{\alpha;L} \{\alpha;L\}$$

$$(\vee\text{E})\quad \frac{\Gamma \vdash \varphi \vee \psi \quad \Gamma,\varphi \vdash [\alpha][L]\phi \quad \Gamma,\psi \vdash [\alpha][L]\phi}{\Gamma \vdash [\alpha][L]\phi} \rightsquigarrow_{\{\alpha;L\}} \{?\varphi;\{\boldsymbol{\beta}\}\} \cup \{?\psi;\{\boldsymbol{\gamma}\}\}$$

Hypothesis proofs do not give concrete strategies for $\alpha;L$ and thus trivially refine $\alpha;L$ to itself. Case analysis allows Demon to choose either branch, so long as it is provable. The output is nondeterministic if φ and ψ are not mutually exclusive. Both φ and ψ are game-free in the system-test fragment and, in practical proofs, even belong to quantifier-free first-order arithmetic.

We now discuss the discrete *Angelic* cases, which plug in the specific Angel strategy from proof $\mathcal{A}$. The base cases follow from the presented atomic cases by letting $L = \mathsf{skip}$, for example rule $\langle{:}*\rangle$I can be applied at a leaf where ϕ is a non-modal formula, in which case the output is merely $x := f$ rather than $\{x := f;\boldsymbol{\alpha}\}$ where $\mathcal{A} \rightsquigarrow_L \boldsymbol{\alpha}$. Because the inductive call of rule $\langle{:}*\rangle$I is written $\mathcal{A} \rightsquigarrow_L \boldsymbol{\alpha}$, the rule $\langle{:}*\rangle$I is also an example of a rule where there is no particular relationship between the symbols α and $\boldsymbol{\alpha}$, indeed it does not even mention the symbol α.

$$\langle{:}*\rangle\text{I}\frac{\Gamma\frac{y}{x}, x = (f\frac{y}{x}) \vdash [L]\phi}{\Gamma \vdash [\{x := *\}^d][L]\phi} \rightsquigarrow_{\{x:=*\}^d;L} x := f;\boldsymbol{\alpha} \qquad \langle{:=}\rangle\text{I}\frac{\Gamma\frac{y}{x}, x = (f\frac{y}{x}) \vdash [L]\phi}{\Gamma \vdash [\{x := f\}^d][L]\phi} \rightsquigarrow_{\{x:=f\}^d;L} x := f;\boldsymbol{\alpha}$$

$$\langle?\rangle\text{I}\frac{\Gamma \vdash \psi \quad \Gamma \vdash [L]\phi}{\Gamma \vdash [\{?\psi\}^d][L]\phi} \rightsquigarrow_{\{?\psi\}^d;L} \boldsymbol{\beta} \qquad \langle;\rangle\text{I}\frac{\Gamma \vdash [\alpha^d][\beta^d][L]\phi}{\Gamma \vdash [\{\alpha;\beta\}^d][L]\phi} \rightsquigarrow_{\{\alpha;\beta\}^d;L} \boldsymbol{\alpha}$$

$$\langle\cup\rangle\text{I1}\frac{\Gamma \vdash [\alpha^d][L]\phi}{\Gamma \vdash [\{\alpha \cup \beta\}^d][L]\phi} \rightsquigarrow_{\{\alpha\cup\beta\}^d;L} \boldsymbol{\alpha} \qquad \langle\cup\rangle\text{I2}\frac{\Gamma \vdash [\beta^d][L]\phi}{\Gamma \vdash [\{\alpha \cup \beta\}^d][L]\phi} \rightsquigarrow_{\{\alpha\cup\beta\}^d;L} \boldsymbol{\alpha}$$

$$\langle*\rangle\text{I}\frac{\Gamma \vdash \varphi \quad \varphi, \mathcal{M}_0{=}\mathcal{M}\succ\mathbf{0} \vdash [\alpha^d](\varphi \wedge \mathcal{M}_0\succ\mathcal{M}) \quad \varphi,\mathbf{0}\succcurlyeq\mathcal{M} \vdash [L]\phi}{\Gamma \vdash [\{\alpha^*\}^d][L]\phi} \rightsquigarrow_{\{\alpha^*\}^d;L} \{?\mathcal{M}\succ\mathbf{0};\boldsymbol{\beta}\}^*;?\mathbf{0}\succcurlyeq\mathcal{M};\boldsymbol{\gamma}$$

Discrete assignments remain in the output. Nondeterministic assignments are proved by providing a witness term f (rule $\langle{:}*\rangle$) and proving the postcondition after assigning x the value of f. Subtly, Angelic tests can be eliminated by rule $\langle?\rangle$ because they are *proven to succeed* and because we wish only to keep tests which *Demon* is required to pass. We reiterate the meaning of bold Greek variables: in the $\langle?\rangle$ case, for example, $\boldsymbol{\beta}$ is the result of the recursive call on $\Gamma \vdash [L]\phi$, whose game argument (not written) is L.

The sequential composition case is the reason that we have structured the entire definition of reification with a "stack" game L and postcondition $[L]\phi$. The recursive call[9] of the sequential composition case is $\mathcal{A} \rightsquigarrow_{\alpha^d;\{\beta^d;L\}} \boldsymbol{\alpha}$, which implements the sequential composition $\{\alpha;\beta\}^d$ with stack L by reifying α^d first and pushing β^d onto the stack to

[9]Recall that the game arguments of recursive calls, here $\alpha^d;\{\beta^d;L\}$, are not explicitly written in the rule-based notation above.

get a new stack of $\beta^d; L$. Subtly, the programs β^d and L will be reified within the single recursive call on $\alpha^d; \{\beta^d; L\}$: the normal-form proof of $[\alpha^d][\beta^d]L$ will first prove away α^d, then present a proof of the new postcondition $[\beta^d][L]\phi$, which eventually gets reified by a nested recursive call with game argument $\beta^d; L$. While technically inconvenient, the stack-based treatment of sequential compositions is a crucial tool to overcome the fact that a proof about $\alpha; \beta^d$ is often not merely one proof for α^d and one proof for β^d: if α^d contains branching connectives, the proof about $\{\alpha; \beta\}^d$ often contains multiple different subproofs about β^d throughout different branches. Our definition seamlessly reifies proofs about $\{\alpha; \beta\}^d$ even when α^d has many branches and β^d has many proofs. For example, the interaction of this case with the Demonic choice case allows returning a branching program which uses the reification of each β^d subproof in the corresponding system branch.

Normal Angelic choice proofs are injections, so Angelic proofs reify by rule $\langle\cup\rangle$R1 or $\langle\cup\rangle$R2 according to one branch or the other. Normal Angelic repetition proofs are by convergence: some metric $\mathcal{M}$ decreases to terminal value $\mathbf{0}$ while maintaining invariant formula φ. Variable $\mathcal{M}_0$ remembers the value of $\mathcal{M}$ at the start of each loop iteration for comparison purposes. Hybrid systems loops are nondeterministic, so Demon chooses the loop duration, but the Demonic test $\mathcal{M} \succ \mathbf{0}$ must pass at each repetition (a loop only runs its body when its guard has not yet terminated) and $\mathbf{0} \succcurlyeq \mathcal{M}$ must pass at the end (a loop only terminates when its guard terminates), restricting Demon's choice of loop duration to those durations permitted by the Angelic loop.

We give discrete Demonic reification rules, then compare them to the Angelic ones.

$$\llbracket:=\rrbracket\mathrm{I}\frac{\Gamma\frac{y}{x}, x = f\frac{y}{x} \vdash [L]\phi}{\Gamma \vdash [x := f][L]\phi} \rightsquigarrow_{\{x:=f\};L} x := f; \boldsymbol{\alpha} \qquad [:*]\mathrm{I}\frac{\Gamma\frac{y}{x} \vdash [L]\phi}{\Gamma \vdash [x := *][L]\phi} \rightsquigarrow_{\{x:=*\};L} x := *; \boldsymbol{\alpha}$$

$$\llbracket;\rrbracket\mathrm{I}\frac{\Gamma \vdash [\alpha][\beta][L]\phi}{\Gamma \vdash [\alpha; \beta][L]\phi} \rightsquigarrow_{\{\alpha;\beta\};L} \boldsymbol{\alpha} \qquad [?]\mathrm{I}\frac{\Gamma, \psi \vdash [L]\phi}{\Gamma \vdash [?\psi][L]\phi} \rightsquigarrow_{?\psi;L} ?\psi; \boldsymbol{\alpha}$$

$$[\cup]\mathrm{I}\frac{\Gamma \vdash [\alpha][L]\phi \quad \Gamma \vdash [\beta][L]\phi}{\Gamma \vdash [\alpha \cup \beta][L]\phi} \rightsquigarrow_{\{\alpha\cup\beta\};L} \boldsymbol{\alpha} \cup \boldsymbol{\beta} \qquad [*]\mathrm{I}\frac{\Gamma \vdash \psi \quad \psi \vdash [\alpha]\psi \quad \psi \vdash [L]\phi}{\Gamma \vdash [\alpha^*][L]\phi} \rightsquigarrow_{\{\alpha^*\};L} \boldsymbol{\beta}^*; \boldsymbol{\gamma}$$

Reification of a discrete *Demonic* connective does not restrict Demon's capabilities, but recursively reifies any Angelic proof steps appearing later (in L or under dualities). Nondeterministic Demonic assignments (rule $[:*]$I), unlike Angelic ones (rule $\langle:*\rangle$I), are not modified during reification, because Demon retains the power to choose any value. Demonic tests introduce assumptions (rule $[?]$I), which the reified system must keep in order to avoid changing the acceptable behavior. In contrast, Angelic tests (rule $\langle?\rangle$I) are erased during reification because they were Angel's responsibility to prove, which she has already done, and impose no restriction on Demon. Demonic sequential compositions are like Angelic ones. Demonic choices reify each branch (rule $[\cup]$I) because Angel does not choose which branch of a Demonic choice is taken, in contrast to Angelic choices, which have two rules (rules $\langle\cup\rangle$I1 and $\langle\cup\rangle$I2) which refine to whatever branch Angel choses to take. Note that reification of games with form $\{\alpha \cup \beta\}; L$ follows distributive normal forms $\{\alpha; L\} \cup \{\beta; L\}$,

which are equivalent by rule ;d$_r$. Demonic repetitions keep the loop, recalling that the loop invariant ψ justifies the postcondition by premise $\mathcal{C}$.

We give the reification cases for ODEs.

$$\text{dsolve}\frac{\begin{array}{c}\Gamma \vdash d \geq 0\\ \Gamma\tfrac{y}{x}, 0{\leq}t{\leq}d, x = sln\tfrac{y}{x}, x'{=}f \vdash \psi\\ \Gamma\tfrac{y}{x}, 0{\leq}t{=}d, x = sln\tfrac{y}{x}, x'{=}f \vdash [L]\phi\end{array}}{\Gamma \vdash [t := 0; \{t'{=}1, x'{=}f \,\&\, \psi\}^d][L]\phi} \rightsquigarrow_{\{t:=0;\{x'=f\,\&\,\psi\}^d\};L} t := d; x := sln; x' := f; \boldsymbol{\gamma}$$

$$\text{bsolve}\frac{\Gamma\tfrac{y}{x}, t{\geq}0, \hat{\psi}, x{=}sln\tfrac{y}{x}, x'{=}f \vdash [L]\phi}{\Gamma \vdash [t := 0; \{t'{=}1, x'{=}f \,\&\, \psi\}][L]\phi} \rightsquigarrow_{\{t:=0;x'=f\,\&\,\psi\};L} t := 0; \{t'{=}1, x'{=}f \,\&\, \psi\}; \boldsymbol{\alpha}$$

$$\text{DW}\frac{\Gamma\tfrac{y}{x}, \psi \vdash [L]\phi}{\Gamma \vdash [x'{=}f \,\&\, \psi][L]\phi} \rightsquigarrow_{\{x'=f\,\&\,\psi\};L} x := *; x' := f; ?\psi; \boldsymbol{\alpha}$$

$$\text{DC}\frac{\Gamma \vdash [x'{=}f \,\&\, \psi]\varphi \quad \Gamma \vdash [x'{=}f \,\&\, \psi \wedge \varphi][L]\phi}{\Gamma \vdash [x'{=}f \,\&\, \psi][L]\phi} \rightsquigarrow_{\{x'=f\,\&\,\psi\};L} \boldsymbol{\beta}$$

$$\text{DG}\frac{\Gamma, y{=}f_0 \vdash [x'{=}f, y'{=}a(x)y + b(x) \,\&\, \psi][L]\phi}{\Gamma \vdash [x'{=}f \,\&\, \psi; \{y := *; y' := *\}^d][L]\phi} \rightsquigarrow_{\{x'=f\,\&\,\psi\};L} y := f_0; \boldsymbol{\alpha}$$

Variable y is fresh in rules dsolve, bsolve, and DG. In rule dsolve, $\boldsymbol{\gamma}$ refers to the reification result of the premise that proves the postcondition $[L]\phi$. Also in rule dsolve, the side condition requires that y is fresh, term sln is the unique solution of the ODE, and chosen duration term d is constant throughout the ODE.

The reification of an invariant-based Demonic proof (rules DC and DW) is a *relaxation* of the ODE: the reified system need not follow the precise behavior of the ODE so long as all invariants required for the proof are obeyed. Indeed, this is where proof-based synthesis in ModelPlex (Mitsch & Platzer, 2016b) gains much of its power: real implementations never follow an ODE with perfect precision, but usually do follow its invariant-based relaxation. This is the same power exploited in Chapter 3 and Chapter 8.

The Angelic domain constraint is comparable to an Angelic test: it is soundly omitted in the reification because it is proven to pass. In Demonic ODE solutions (rule bsolve), the duration and domain constraint are *assumptions*, and the defined formula $\hat{\psi} \equiv \forall 0 \leq s \leq t\, [t := s; x := sln]\psi$ says the domain constraint ψ holds through time t where s is fresh. Since our ODEs are effectively-locally-Lipschitz continuous, they have unique solutions. Thus, Demon could soundly reify the unique solution sln of the ODE that was specified by the input proof in the application of dsolve. The main benefit of doing so would be that the output falls within *discrete* dynamic logic. For that reason, the implementation of synthesis in constructive VeriPhy (Chapter 8) will use the discrete translation, whereas we use the hybrid translation here for purely stylistic reasons to emphasize that reification targets Angelic connectives and can leave Demonic connectives undisturbed, unless there is an overriding practical reason to translate them.

Of course, there often is a practical reason to simplify Demonic ODEs. We now discuss the invariant-style rules for Demonic ODEs, which do translate away the ODE for the practical reason that the translation results in monitors (Chapter 8) which are much more

permissive than monitoring of an exact ODE solution, thus more useful in practice, yet still provably safe. Differential Cut (rule DC) reification introduces an assumption in the domain constraint, and is sound by rule DC. By itself, rule DC *strengthens* a program, but in combination with rule DW enables relaxation of ODEs. Differential Weakening (rule DW) relaxes an ODE by allowing x and x' to change *arbitrarily* so long as the domain constraint ψ (and thus invariants introduced by rule DC) remain true. Note the order of operations for the case for rule DW: assumption of the domain constraint should occur in the final state of the ODE, so the test is placed after the assignments. The right-hand side f, which determines the differential variable x', can and often does mention x, so it is assigned after x to ensure that it captures the value of x' for the *final* state. Differential Ghost rule DG adds a dimension to the ODE and reifies according to the recursive call. The introduced dimension is linear in order to soundly preserve the duration of the ODE. Assignment $y := f_0$ sets the ghost variable's initial value to a chosen term. The dual assignments $\{y := *; y' := *\}^d$ capture the fact that $[L]\phi$ only holds for *some* values of y and y'. In practice, the assignments $\{y := *; y' := *\}^d$ can be eliminated by assuming that y and y' are fresh in $[L]\phi$, but our presentation of the rule simplifies proofs about reification.

Reification Example. Recall example PP and its safety property (6.1). Let $\mathcal{A}_{\mathsf{PP}}$ be the proof of (6.1) with a mirroring strategy (Section 6.2.2). The reified result $\boldsymbol{\alpha}_{\mathsf{PP}}$ is

$$\begin{aligned}\boldsymbol{\alpha}_{\mathsf{PP}} = \big\{&\{L := -1; R := 1; x' = L + R \,\&\, x_\ell \leq x \leq x_r\}\\ &\cup\{L := 1; R := -1; x' = L + R \,\&\, x_\ell \leq x \leq x_r\}\big\}^*\end{aligned}$$

which we discuss step-by-step. Demonic repetition reification just repeats the reified body. Reifying a Demonic choice follows the structure of the proof, not the source program, hence the ODE occurs for each branch. Each branch commits to a choice of L, and each branch of $\mathcal{A}_{\mathsf{PP}}$ resolves the Angelic choice R to balance out L. When reifying an Angelic choice, only the branch taken is emitted. In $\boldsymbol{\alpha}_{\mathsf{PP}}$, we assume that $\mathcal{A}_{\mathsf{PP}}$ proves the ODE $x' = L + R \,\&\, x_\ell \leq x \leq x_r$ by replacing it with its solution, which is why the ODE appears verbatim in the refined system. A differential invariant proof could also be used with a differential cut (DC) of $x = x_0$, in which case physics are represented by the program $x := *; x' := *; ?x_\ell \leq x \leq x_r \wedge x = x_0$ in the result of reification. Different proofs generally give rise to different systems, some of which are less restrictive than others. Differential invariants, especially inequational invariants, ($x \geq x_0$ vs. $x = x_0$) can be more easily monitored with finite-precision numbers.

Note that the system $\boldsymbol{\alpha}_{\mathsf{PP}}$ is a refinement of PP and satisfies the same safety theorem $\mathsf{pre} \to [\boldsymbol{\alpha}_{\mathsf{PP}}]x = x_0$. Next, we show that this is the case for all reified strategies.

Metatheoretic Results. We state theorems (proven in Appendix C.1) showing how the reification of a game α refines α. Recall that Γ, α, ϕ, and $\mathcal{A}$ all belong to the *system-test* fragment of CdGL.

Theorem 6.2 (Systemhood)**.** *If $\Gamma \vdash \mathcal{A} : [\alpha]\phi$ in CdGL without refinement for system-test $\Gamma, \mathcal{A}$, and hybrid game α and $\mathcal{A} \rightsquigarrow_\alpha \boldsymbol{\alpha}$ then $\boldsymbol{\alpha}$ is a system, i.e., it does not contain dualities.*

Theorem 6.3 (Reification transfer)**.** *If* $\Gamma \vdash \mathcal{A} : [\alpha]\phi$ *in* CdGL *without refinement for system-test* $\Gamma, \mathcal{A}$*, and hybrid game* α *and* $\mathcal{A} \rightsquigarrow_\alpha \boldsymbol{\alpha}$ *then* $\Gamma \vdash [\boldsymbol{\alpha}]\phi$ *is provable in* CdGL *without refinement.*

Theorem 6.4 (Reification refinement)**.** *If* $\Gamma \vdash \mathcal{A} : [\alpha]\phi$ *in* CdGL *without refinement for system-test* $\Gamma, \mathcal{A}$*, and hybrid game* α *and* $\mathcal{A} \rightsquigarrow_\alpha \boldsymbol{\alpha}$ *then* $\Gamma \vdash \boldsymbol{\alpha} \leq_{[]} \alpha$ *is provable in* CdGL *with refinement.*

Theorem 6.2 is proven by trivial induction on $\mathcal{A}$. Theorem 6.3 is proven by inducting on $\mathcal{A}$, reusing its contents in a proof for $\boldsymbol{\alpha}$. Theorem 6.4 inducts on $\mathcal{A}$ and in each case appeals to the corresponding refinement rule. The fact that Theorem 6.4 could be proved validates the strength of CdGL's refinement rules.

6.6 Discussion

Refinement reasoning is of general interest whenever we wish to prove a relationship between two programs or whenever we wish to verify a complex program by first verifying a simple one. In the domain of CPS, one might wish to prove that one controller is more efficient than another or prove that an entire line of cars are safe by comparing them to a common base model. In the context of this thesis, refinement has the additional advantage that it opens an avenue to support games in synthesis and verification tools by first supporting systems, then reducing verified games to systems implementing their proven winning strategies. Our reduction from game proofs to systems will be used in Part III as we develop new tooling for CdGL.

In developing a refinement calculus for Constructive Differential Game Logic (CdGL), we overcame several technical challenges. Refinement is made more subtle by the facts that game logic is subnormal and subregular, and that ours is a *constructive* logic. Subnormality and subregularity imply that game logic generally needs a semantics beyond vanilla Kripke semantics, thus our semantics is designed for the more general semantics of games rather than using the Kripke semantics of refinement which appears in the literature (Loos & Platzer, 2016). Constructivity means we cannot use the classical equivalence $[\alpha]\phi \leftrightarrow \neg\langle\alpha\rangle\neg\phi$ to make Angelic and Demonic CdGL modalities interdefinable; while Angelic and Demonic *refinement* are interdefinable, the constructive distinction between Angelic and Demonic *modalities* necessitates an increase in the number of rules.

We introduced a new constructive semantics for refinement and proved soundness. We formalized a reification operation and folklore theorem which reduce verified hybrid games to hybrid systems by specializing a game to the commitments made by its winning strategy. Our reification results provide several insights relevant to tool development. Theorem 6.2 and Theorem 6.3 support synthesis by ensuring that the reified system is a *system* which satisfies the *same* safety condition as the input game, which are respectively required in order to use systems-based synthesis algorithms and to ensure an end-to-end safety guarantee. These results suggest hybrid games synthesis can be implemented by reduction to hybrid systems synthesis, simplifying implementation while still providing the robustness and the control code synthesis feature which are desired by the Engineer. Theorem 6.4 supports refinement-based proof technology: because the reified system refines

the input game, game safety can be shown by choosing a strategy and showing the strategy (equivalently, the reified system) safe. This refinement-based approach will benefit the Logic-User by providing a gentle approach for learning hybrid games proofs once hybrid systems verification is understood. Theorem 6.4 is also useful for arguing that liveness guarantees transfer from games to systems (Chapter 8), because it captures the idea that the reified system can reach at least all the states reached when playing the given strategy in the game, potentially more.

We expect that, like most refinement calculi, CdGL refinements are of interest when comparing the efficacy (dominance) of two control strategies for a same game, or when verifying a complex model in stages. Additionally, by reducing games to systems and comparing equivalence of systems, one also gains a notion of when two strategies can be considered "the same".

Part III

Proof-Structured Synthesis

Chapter 7

Structured Proofs for Dynamic Logics

In Chapter 3 we identified the limitations of Bellerophon (Fulton et al., 2017), the current-generation proof script language for dL, which could be called an unstructured proof script language. Firstly, unstructured proofs are difficult to process automatically in synthesis tools, leading to an implementation of VeriPhy (Chapter 3) that left the Engineer desiring greater robustness. Secondly, unstructured proofs give the Logic-User several challenges regarding maintainability, traceability, and readability. In this chapter, we identify structuring principles for program-logic proofs, and design and implement a proof language Kaisar which applies those principles to CdGL (from Chapter 5). While Kaisar is necessarily specific to the dL family, our goal is to present design principles in enough generality that they could be applied to other languages.

At the highest level, Kaisar's novel design follows from two core design principles which respectively come from CdGL's constructivity and from the literature on structured proof languages (Section 7.1): (I) the *Curry-Howard* isomorphism for games says that a *constructive* game proof consists of a programmatic strategy, which behaves like a hybrid system, and a correctness proof for that strategy; (II) proofs should be *stable* in the sense that a revision to one model or proof statement will not require non-local changes to conceptually unrelated proof statements.

These principles manifest in features which support the Logic-User's aforementioned design priorities of readability, maintainability, and traceability. The same principles also support additional goals such as conciseness and flattening of the learning curve for proofs, of which the former is important because it saves effort for the Logic-User and the latter is important because it simplifies the training process for a new Logic-User. Given the current state of the art, such as Bellerophon, these priorities remain crucial to productivity and accessibility of new languages.

Principle I gives our workflow a gentle learning curve. Organizing proofs around strategy programs with inline annotations both allows easy visual tracing of the relationship between proof and code and also lets us exploit insights on readability, maintainability, and scalability from programming language design. First, the user simply writes a hybrid game strategy, analogous to a hybrid system. Next, specifications are inserted gracefully

and iteratively into the model, then Kaisar attempts automatic proofs. If models are simple, the hardest task, manual proof, can be completed avoided. More often, manual proofs are used, but only for crucial questions and only once easier tasks are completed. Lastly, the approach is extended from strategies to games with automatic refinement (Chapter 6) reasoning that checks whether a given strategy plays a given hybrid game. By building game verification on top of systems verification, we allow a new Kaisar user to build their knowledge of games on top of any existing knowledge of hybrid systems. More broadly, simplifying the learning curve of CPS verification is crucial because CPS correctness is of broad interest, which means there exist many potential Logic-Users, yet much of this broader audience does not already possess the Logic-User's desired proof expertise. While novel proof language designs will not fundamentally change the fact that interactive proofs require special expertise, they are a key tool for making that expertise easier to acquire by building on an existing understanding of programming.

Principle II is broad on purpose since stateful systems like hybrid games require notions of proof stability which are more general than those in common use. Before giving new notions of stability, we show two commonly-used principles which also appear in Kaisar:

III proof contexts should support named references with lexical, persistent scoping, even as game state changes. Destructive, imperative changes to contexts are disallowed;

IV by default, fact selection should rely on positive mention of relevant facts as opposed to negative mention of irrelevant facts.

Principle III is essential for maintainability: changing one line of a model should not invalidate proof steps which refer to unrelated facts and program statements. Named facts are essential to readability: asking the Logic-User to refer to assumptions by position is equivalent to asking the Engineer to refer to variables by memory address. While named, well-scoped facts may be taken for granted in some verification tools, program logics present a twist: game state changes frequently, and proofs must frequently use facts which are proved in other states. Because changes in the state can change which facts are true, it is both important for the proof language to carefully ensure soundness of fact transport across states and important to provide automated support for fact transport. Principle IV supplements Principle III: because backend solvers need assumptions to be minimized, maintenance of negative-style proofs often involves adding many non-local weakening steps to cancel out a single new assumption. Positive-style proofs avoid polluting the context of every proof step when a new assumption is added. Either negative-style or positive-style proofs can be more verbose in different situations: positive-style proofs can be verbose when most facts are used in most proof steps, while negative-style proofs can be verbose when most facts are *not* used in most proof steps. Not only do we wish to remember facts proven in previous states, but we must relate multiple states or even predict hypothetical future states in order to best support common proof idioms.

To best support proofs of programmatic models with rich, changing state, Kaisar must go beyond existing principles of proof stability. To this end, Kaisar introduces a new feature, *labeled reasoning*, which provides a uniform mechanism for stable references to past and future system *states*.

Labeled reasoning significantly simplifies various important CPS paradigms: *i)* model-

predictive controllers (Mitsch et al., 2017) that predict future states to make safe control choices, *ii)* ODE invariant (Mitsch et al., 2017) proofs that compare current and initial states, *iii)* stability proofs (Chan et al., 2016) that track change in distance, *iv)* liveness (Mitsch et al., 2017) and reach-avoid (Bohrer & Platzer, 2020b) proofs which require progress arguments, *v)* sandbox controllers (Bohrer et al., 2018) that allow the model-predictive approach to safely interact with external control code, which is key for synthesis (Chapter 8). Of these, we will present examples of (logical) model-predictive control, invariants, reach-avoid correctness, and sandboxing.

We prioritize presenting the Kaisar language and its uses over presenting implementation details, but the implementation of Kaisar's novel design also overcomes novel technical hurdles, as Section 7.4 will discuss. One of the major technical hurdles for the implementation is to provide a systematic treatment of state, which crucially supports Principle II, Principle III, and especially labeled reasoning in the presence of rich mutable state. That treatment is provided with the combination of rich data structures for organizing assumptions and definitions together with a notion of *static single-assignment* proof which is inspired by the standard notion (Cytron, Ferrante, Rosen, Wegman, & Zadeck, 1991) from compilers, but serves a distinct purpose of systematically naming and remembering historical program state for easy processing by high-level proof automation.

A key ingredient of the systematic treatment of state is discrete ghost state reasoning, which allows a proof context to remember facts about old states when the current state changes. For hybrid systems (and games) specifically, *differential* ghosts are also essential (Platzer & Tan, 2018, 2020; Platzer, 2012a) because there exist properties of ODEs which can be proven with ghost reasoning but cannot be proven without it. Thus, it is also important that the treatment of ghosts in Kaisar is general enough to capture both (automatic) discrete ghost steps and (manual) differential ghost steps. By enabling features such as labeled reasoning, these technical advances crucially help Kaisar achieve the Logic-User's language goals.

We outline the structure of the chapter. Related work, including a wide array of existing program verification languages and tools that inspired Kaisar, is in Section 7.1. The connection between games and systems is explained in Section 7.2. The heart of the chapter, Section 7.3, introduces Kaisar. It starts with toy examples but builds to fundamental paradigms such as logical model-predictive control, sandboxing, and reach-avoid correctness. Key implementation decisions are discussed in Section 7.4. Kaisar is evaluated against Bellerophon, KeYmaera X's unstructured language for proofs, on existing case studies in Section 7.5.

7.1 Related Work

Related works for Kaisar include other structured proof languages, annotation-based proof languages, tactic languages, and unstructured proof languages.

Structured Proofs. Structured proofs were introduced over 40 years ago in Mizar, which has continued to be used since then (Bancerek et al., 2015; Wenzel & Wiedijk, 2002).

Another early high-level proof language is the original proofchecker for (first-order, discrete) dynamic logic (Litvintchouk & Pratt, 1977) where facts are named and each assertion has an unstructured proof step which specifies assumptions and propositional unification hints. Isar (Wenzel, 2006, 1999, 2007) is one of the most mature structured proof languages today, and has been successfully applied to large-scale verification tasks (Nipkow, 2002; Klein et al., 2010; Lochbihler, 2007). Other structured languages include DECLARE (Syme, 1997) for HOL, TLAPS for TLA$^+$ (Cousineau et al., 2012; Lamport, 1992), SSReflect (Gonthier & Mahboubi, 2010) and declarative proofs (Corbineau, 2007) in Coq.

Kaisar is best contrasted with related works by identifying two classes of existing approaches: structured languages and annotation-based provers, i.e., provers driven by contract or invariant annotations on programs. The goal and strength of Kaisar's structured program proofs is to combine the readability of concise annotations with the generality and explicit control of structured proofs. Fully-automatic provers for undecidable properties of annotated programs will always be incomplete, thus harder to scale or debug. By allowing explicit proofs in annotations, we allow proofs to start simple and grow smoothly in complexity as needed. We demonstrate that our combination of annotation and general proof leads us to a fruitful and under-explored point in the design space of proof languages.

Kaisar inherits a number of structuring principles from previous structured languages. Many structured languages, Kaisar included, support lexical scope, definitions, forward-chaining proof terms, and convenient notation for fundamental proof techniques such as induction. Kaisar contributes a generalization of lexical scoping to program logics and convenient notations for ODE verification. Kaisar pushes unstructured proof commands to leaves. Kaisar shares this design decision with Mizar, where an assertion uses a single unstructured step, but not Isar, which allows arbitrarily-complex unstructured proofs in Isabelle's "apply-script" language. Surface-level syntax decisions in Kaisar often follow Isar terminology, including the keywords **note** and **let**. Structured proof principles have been provided in a number of other provers through "Mizar modes" which lie outside the prover's core. Mizar modes are available in Cambridge HOL (Harrison, 1996), Isabelle (Kaliszyk, Pak, & Urban, 2016), and HOL Light (Wiedijk, 2001). The "mode" approach was also used to write the Iris Proof Mode (IPM) in Coq (Krebbers, Timany, & Birkedal, 2017), which implements the Iris concurrent separation logic in Coq's $\mathcal{L}$`tac` language. The "Mizar mode" approach works well in provers with small LCF-style cores, where the Mizar mode can employ the core as a "library". Coq and KeYmaera X both have small LCF-style cores, which have even been partially formalized, respectively in (Barras & Werner, 1997) and Chapter 2. Kaisar is not implemented as a user-level "mode" because we wish to target CdGL, which does not have an existing theorem prover, and because the axiomatic sequent calculus of KeYmaera X differs significantly from the CdGL natural deduction approach, which more readily serves as a foundation for Kaisar. An early proposal for Kaisar (Bohrer & Platzer, 2019) featured a prototype "mode" implementation for classical dL. Experience implementing the early prototype suggested that the "mode" approach would present significant maintenance challenges as KeYmaera X's tactic libraries evolve. The early Kaisar prototype had a formal proof calculus with context management. The proof calculus from that work had high syntactic complexity and provided relatively little insight, thus we decided not to pursue a formal calculus here.

Annotation-Based Verification. Kaisar's second main line of inspiration comes from theorem provers based on annotation. Most annotation-based provers build on the idea of *proof outlines*, introduced by Owicki (Owicki, 1975) and further studied by Apt et al. (K. Apt, De Boer, & Olderog, 2010). Proof outlines annotate programs with assumptions, invariants, and optionally with assertions that must hold at a given intermediate point. When we discuss the strengths and weaknesses of annotation-based provers as a class, the same strengths and weaknesses will apply to proof outlines because proof outlines *are* the canonical annotation-based proof system. KeYmaera X features a limited form of annotation-based proof: loops and ODEs can be annotated with invariants which will be used in proof automation; in simple cases, invariants suffice to completely automate a proof if the resulting arithmetic subgoals are amenable to solvers. There have long existed tools which use annotations to automatically reason about program correctness, such as the ESC family (Leino, 1998; Flanagan et al., 2002) of tools. Over time, a wide variety of such tools have emerged which combine annotations with automated proofs using SMT solvers or other solvers, among which prominent systems include but are not limited to Dafny (Leino, 2010), Gappa (de Dinechin, Lauter, & Melquiond, 2011), F* (Swamy et al., 2016), OpenJML (Cok, 2011), Why3 (Filliâtre & Paskevich, 2013), and Leon (Blanc, Kuncak, Kneuss, & Suter, 2013). Automated solvers for most nontrivial languages are fundamentally incomplete, and incompleteness can be sidestepped by increasing the expressiveness of annotations. For example, Dafny allows chains of equational reasoning to assist automation, but even equational proofs can be fragile and challenging to debug when backed by an opaque SMT solver. Meta-F* (Martínez et al., 2019) shares our interest in integrating annotation-based proofs for programs with interactive proofs, but their emphasis is on extending annotation-based proof with metaprogramming over tactics, while we emphasize the importance of integrating annotation-based and interactive proof even in the presence of changing program state. Ivy (Padon, McMillan, Panda, Sagiv, & Shoham, 2016) avoids the limitations of opaque solvers by targeting a decidable logic; the tradeoff is that modeling a system in a decidable logic can be challenging. Ivy's decidable approach does not apply to CdGL, which is undecidable.

Next, we contrast Kaisar's structured proofs with verification-condition generators (VCGs) (King, 1971). VCGs underlie many annotation-based provers: the prover uses annotations to generate a first-order logic formula (verification condition) which, if true, implies correctness of the program. The VC is then fed to either an automatic prover or an interactive prover. A major limitation of VCGs is that in many cases it is undecidable whether the VC is valid, so any automation used is incomplete. Even when VCs fall within a decidable logic, they are often infeasible in practice. This is the case, for example, when KeYmaera X generates VCs which belong to first-order real arithmetic, because first-order real arithmetic is decidable (Tarski, 1951), but often intractable (Davenport & Heintz, 1988; Weispfenning, 1997). For this reason, invariant annotations typically do not suffice to automate KeYmaera X proofs in practice.

Approaches which rely solely on automated backends then fail to scale to large proofs, but the interactive approach of KeYmaera X can also manifest its own *traceability* issues which become increasingly noticeable as scale increases. As discussed in Section 3.8, the text of a verification condition (or proof context) generated in the course of a KeYmaera X

proof can differ significantly from the text of the hybrid program model, which makes it more difficult for the Logic-User to prove or debug the verification condition because its relationship to the model is indirect.

In contrast, annotated program texts make the connection between program text and annotated condition visually immediate. Because those annotated conditions can be written in terms of labeled proof locations and defined symbols, they can often be written at a high enough level of abstraction that the intuitive dynamical justifications behind the proof are apparent as well. Because Kaisar elaborates labeled expressions and definitions before proving an annotation, Kaisar's elaboration pass is also a simple VCG. By comparison to other VCGs, the advantage is that the VCs are generated locally for individual proof steps in terms of labels and definitions whose meaning can readily be traced through the proof text. While the reader may need to consult other lines of the proof or symbol definitions to understand an annotation, all such dependencies are explicit in the annotation and can be traced by reading a single artifact, rather than trying to match up separate models and proofs that must be read side-by-side.

Tactics and Unstructured Proofs. We also compare structured proof languages with natural-language structured proof paradigms (Lamport, 2012, 1995). While their main goals are to ensure that paper proofs are correct and readable, they also inspired the design of TLAPS, a formal proof language for TLA$^+$ (Cousineau et al., 2012; Lamport, 1992). Lamport's structured proofs emphasize hierarchy: any non-trivial assertion is broken into further steps. When the reader wants a concise proof, the detailed steps can be skipped, leaving a high-level proof. In a formal proof language, small proof steps may be omitted if they could be proved automatically, but the concept of increasing detail is still valuable: it is natural to attempt an automated proof first, then add details only where necessary.

Structured proofs aside, two classes of languages used in theorem-proving are tactics languages (which implement reusable automation) and unstructured proof languages (for concrete proofs). Automation can often be written in the prover's implementation language: OCaml in Coq, ML in Isabelle, or Scala in KeYmaera X. Automation and structuring are not mutually exclusive: notably, the `auto2` (Zhan, 2016) language for Isabelle/HOL integrates standard structured proof concepts into an ambient proof search algorithm. Tactic languages include untyped $\mathcal{L}$`tac` (Delahaye, 2000), reflective Rtac (Malecha & Bengtson, 2015), and dependently typed Mtac (Ziliani, Dreyer, Krishnaswami, Nanevski, & Vafeiadis, 2013) in Coq, Eisbach (Matichuk, Murray, & Wenzel, 2016) in Isabelle, and VeriML (Stampoulis & Shao, 2010, 2012) in a standalone checker.

The KeY theorem prover (Ahrendt et al., 2016) features a soundness-critical language of "taclets" (Habermalz, 2000) with limited expressiveness, which can be understood as allowing user-specified inference rules. For the creation of concrete proofs, as opposed to the specification of new inference rules, KeY emphasizes (graphical) user interactions, which allow specifying individual rule applications, applying automated methods, or both. KeY's emphasis on the combination of automation and interactions (as opposed to proof scripts) is thematically similar to the earlier KIV (Reif, 1995) (Karlsruhe Interactive Verifier), another interactive prover for dynamic logic. In turn, the combination of automation and

graphical interaction in KeY was a direct ancestor to KeYmaera (Platzer & Quesel, 2008a) and indirect inspiration for the graphical interface of the successor KeYmaera X (Fulton et al., 2015). While tactic languages and taclets have been used to implement impressive automation, the problem they solve differs from the one we are interested in: expressing concrete proofs. Kaisar is not concerned with implementing entirely new automation, though Kaisar proofs do often employ automated procedures.

Examples of unstructured languages are the Coq (*Coq Proof Assistant*, 1989) script language, the Isabelle (Nipkow et al., 2002) apply-script language, the Bellerophon language (Fulton et al., 2017) for unstructured proofs and tactics in KeYmaera X, and the KeY Proof Script (Grebing, 2019, Ch. 7) language. Bellerophon is an important point of comparison for Kaisar, as both target hybrid system and game proofs. Bellerophon consists of regular expression-style tactic combinators (sequential composition, repetition, etc.) and a standard tactics library featuring, e.g. sequent calculus rules, automated proof search, and invariant reasoning. Bellerophon's strength is in tactics that combine the significant automation provided in its library. Its weakness is in performing large-scale concrete proofs. For example, assumptions in Bellerophon are unnamed and referred to by their index or by searching for a given formula or formula pattern. Indexed references are particularly unreadable and brittle at scale. However, Section 3.8 argues that even search-based references in Bellerophon can become difficult to maintain when one change in a model or proof induces nonlocal changes in proof contexts and creates the need to nonlocally update many proof steps to search the context for a different fact. Bellerophon lacks both line labeling and structuring; the Bellerophon combinators in common use are a strict subset of the combinators in apply-script. The KeY Proof Script (Grebing, 2019, Ch. 7) language places a strong emphasis on using pattern-matching to select which proof goals an unstructured proof method should be applied to, a feature which enables concise proofs for problems with large numbers of similar subgoals. While pattern-matching selectors can be useful across proof paradigms, subgoal selection is specific to unstructured languages because structured languages such as Kaisar explicitly state and prove subgoals for the sake of improved traceability, even when traceability requires increasing length.

Recent releases of Bellerophon have partially addressed scalability issues with features such as auxiliary tactic and expression symbol definitions. The symbol definition feature is based on uniform substitution (Platzer, 2017a). As argued in Section 3.8, recent Bellerophon features do not fully overcome Bellerophon's traceability, readability, and maintainability issues. For example, Bellerophon does not *require* the shape of a proof to follow the shape of a model, which has in practice caused opaque failures in automated tools which wish to process an unstructured proof but need its shape to match that of the model. As of this writing, there is no available reference publication for the Bellerophon definition mechanism in the literature. While Bellerophon has evolved since its initial release and will continue to do so, our point is not that Kaisar supports structured proof styles that could never be emulated in any future release of Bellerophon. Rather, we note that the design of Bellerophon historically *allows* proof styles that have various issues concerning traceability, structuring, and maintainability, so Kaisar is designed to make it outright impossible (in the case of unnamed proof facts) or substantially more difficult (in the case of traceability and maintainability) for common known issues in the use of Bellerophon

proofs to occur in Kaisar proofs. Just as structured programming languages make the use of unstructured `goto` statements impossible, our goal is to make entire classes of unreadable and unmaintainable proofs impossible. Both unstructured programming languages and unstructured proof languages can respectively be used to write readable, maintainable programs or proofs, but the prevalence of unstructured programs or proofs in practice is sufficient to motivate the design of structured languages.

Kaisar draws some inspiration from Bellerophon, but Kaisar often provides a more general alternative to a Bellerophon feature inspired by a more general theoretical foundation. Generalized features such as labeled reasoning take on a qualitatively new role when combined with features shared between Kaisar and Bellerophon, such as function and predicate definitions. The labeled expressions of Kaisar in particular are inspired by nominals *à la* hybrid differential dynamic logic dHL (Bohrer & Platzer, 2018), which was first proposed as $\mathsf{d}\mathcal{L}_h$ (Platzer, 2007b). In dHL, nominal formulas enable stating and proving theorems about named states. Our goal differs: labeling is usually unimportant to our final theorem, but labeling provides a convenient and maintainable structuring principle during the proof.

While we provide the most general treatment of historical state and its theory via labels, there is a rich tradition of historical reference in program proofs. It has long been known that historical reference, implementable with ghost state, is an essential component of proof in many domains (K. R. Apt, Bergstra, & Meertens, 1979; Owicki & Gries, 1976; Owicki, 1975; K. Apt et al., 2010; Clint, 1973). In the context of hybrid systems for example, both KeYmaera 3 and KeYmaera X can introduce subscripted ghost variables x_i which remember old values of some variable x, though the latter prefers to minimize the number of subscripted variables when possible. It has been known equally long (Clarke, 1980) that manual ghost arguments can make proofs clumsy. A number of verification tools (Ahrendt et al., 2016; Fulton et al., 2015; Leino, 2010, 2008; Barnett, Leino, & Schulte, 2005; Leino, Müller, & Smans, 2009; Leavens, Baker, & Ruby, 1999; Kleymann, 1999) offer ad-hoc historical reference constructs without theoretical justification which can reference only the initial program state. We generalize them. We generalize Bellerophon too, which features a limited version of labeling. For example, loop invariant and differential invariant formulas can write $old(f)$ for the value of f at the start of the respective loop or ODE. VDM (Jones, 1991; Kleymann, 1999) also has a feature analogous to *old*: the postcondition of a procedure writes `x~` for the input value of variable `x`. As argued in Section 7.3.7, the `old` notation is inadequate for the full needs of hybrid systems and games.

In the hybrid systems (and games) setting specifically, *differential* ghosts are also essential (Platzer & Tan, 2018, 2020; Platzer, 2012a) for reasoning about differential equations. While the use of differential ghosts is distinct from labeling, Kaisar seamlessly integrates differential ghost reasoning in its annotation-based syntax.

7.2 Relationship of Kaisar to CdGL

Because Kaisar proofs are structured as programs annotated with proof statements rather than as natural-deduction proof trees, the relationship between Kaisar and CdGL (Chapter 5) deserves a moment's reflection. The most basic Kaisar proofs are simply programs

Kaisar	CdGL
`x:=f;`	$x := f$
`pf1 pf2`	$\alpha;\beta$
`?(P);`	$?P$
`x := *;`	$x := *$
`pf1 ++ pf2`	$\alpha \cup \beta$
`{x' = f & ?(P)};`	$x' = f \,\&\, P$
`{pf}*`	α^*

Figure 7.1: Demonic strategy connectives.

which, under the Curry-Howard correspondence for games, are winning strategies that constitute proofs of game theorems. We discuss the language of strategies in Section 7.2.1 to ease the introduction of Kaisar proofs, then discuss the relationship between a strategy and a theorem in Section 7.2.2. That relationship is based on refinements (Chapter 6). Though the discussion of refinement is technically involved, our approach's use of refinement will serve to simplify Kaisar's learning curve in practice, because it allows us to build game verification on top of strategy verification, or equivalently hybrid system verification, which is well-studied.

7.2.1 Kaisar Strategy Language

The Kaisar proof language is presented in full detail in Section 7.3. Here, we ease the presentation by emphasizing the strategic connectives of Kaisar and their corresponding CdGL connectives. Demonic strategy connectives represent steps of gameplay that are under the opponent Demon's control, so they passively accept Demon's choices and wait for Angel's next turn. Angelic strategy connectives represent steps that are under our player Angel's turn, so they are responsible for committing to a specific strategy among Angel's available moves.

The most basic Kaisar strategy for each Demonic connective is given in Fig. 7.1. Note that the plaintext notation for a Demonic choice $\cup$ is written `++`. As in Chapter 6, we assume Demon moves first, so that Angelic connectives appear under dualities and Demonic ones do not. Note that unannotated hybrid systems can also be read as Demonic strategies in Kaisar. This makes sense because Angel does not make any strategic choices during Demon's turn, so there are no choices to resolve in a Demonic strategy.

The Angelic connectives, in contrast, make the strategic decisions, as shown in Fig. 7.2. Because deterministic assignment `x:=f` and sequential composition `pf1 pf2` are self-dual, there is no distinction between their Demonic or Angelic strategies, that is $x := f$ and $\{x := f\}^d$ are semantically equivalent. Assertions (written `!(P);`) resolve Angelic tests by proving that the test condition `P` holds. A proof of `P` may be written (Section 7.3.2), else automatic, but incomplete proof automation is used. Angelic nondeterministic assignments $\{x := *\}^d$ are played by choosing a value for x, thus their strategies are just deterministic assignments $x := f$. In Section 7.2.2, we discuss how refinements can be used post-hoc to distinguish strategies of $\{x := *\}^d$ from strategies of $x := f$. Demonic nondeterministic

Kaisar	Game
`x:=f;`	$\{x := f\}^d$
`pf1 pf2`	$\{\alpha^d; \beta^d\}^d$
`!(P);`	$\{?P\}^d$
`x:=f;`	$\{x := *\}^d$
`switch { case P1 => pf1 case P2 => pf2 }`	$\{\{?P1; \alpha^d\} \cup \{?P2; \beta^d\}\}^d$
`{x' = f, t' = 1 & P & ?(t:=dur)};`	$\{x' = f, t' = 1 \,\&\, P\}^d$
`for(...) {pf}`	$\{\{\alpha^d\}^*\}^d$

Figure 7.2: Angelic strategy connectives.

assignments, unlike Angelic ones, force Angel's strategy to use a value of x chosen by the opponent Demon. Angelic strategies for discrete choices must decide when each branch is taken, thus Angelic choice strategies are `switch` statements. Each case is guarded by a formula and the disjunction of all guards must be constructively provable. By proving that at least one guard always holds, we ensure that the case analysis is total. By proving a *constructive* disjunction, we ensure that the proof of the `switch` statement can be interpreted computationally as a program which computes which branch to take and then executes the strategy corresponding to the branch. By default, totality is checked automatically by Kaisar. If a manual totality proof is necessary, it can be written in parentheses immediately after the `switch` keyword using the proof term notation which we will introduce in Section 7.3.1. Manual totality proofs are especially useful if the `switch` statement is only total by virtue of some assumption from the context, e.g., if the context contains an assumption $x = 0 \vee x = 1$ and the guards of the `switch` statement are $x = 0$ and $x = 1$. After choosing which branch to take, Angel is also responsible for proving the corresponding guard, then returns control to Demon. An Angelic ODE strategy decorates the ODE `x' = f, t' = 1 & P` with an assignment `t:=dur` that specifies Angel's choice of `dur` for the ODE duration. The duration `dur` is a term (and thus a function of the state) which Angel chooses cleverly to achieve her objectives. The test syntax `?(t:=dur);` here is an instance of an advanced, assumption-like notation for assignments which allows giving a name `eq` to the equality fact induced by the assignment by writing `?eq:(t:=dur);`. Because Angel is responsible for proving termination of an Angelic loop, Angel structures her strategy as a `for` loop with an explicit termination metric and convergence proof. The full convergence argument will include an invariant formula which holds throughout the loop as the metric changes.

In this section, we have presented the simplest instance of each strategy connective. In a real Kaisar proof, assertions are annotated with proofs, while ODEs and loops are annotated with invariants (Demonic and Angelic) and metrics (Angelic loops). Our proofs will also use Kaisar features that do not represent a specific hybrid game, such as definitions and forward-chaining proof terms.

7.2.2 Connecting Kaisar, Games, and Systems with Refinement

The relationship between constructive hybrid games and strategies is not one-to-one: a single game can admit many strategies. Conversely, strategies often contain additional steps which contribute to the proof but do not change the meaning of the game. However, CdGL refinement (Chapter 6) provides a principled connection between games, CdGL proofs, and hybrid systems. The same connection applies when connecting *Kaisar* to games and systems, as Kaisar proofs *are* a syntax for game proofs. That is, the game statements contained in a Kaisar proof correspond to a particular strategy. Once a strategy (Kaisar proof) has been written, one can check whether it is a strategy *of* a given game and also extract a hybrid system which captures the states reached when playing the given strategy in the given game.

One strength of annotation-based proof is that the Logic-User can first simply write a program, then add proof annotations gradually. This flattens the learning curve by allowing a new user to write simple models before learning to write proofs. However, annotation-based proof methodologies require some care regarding a fundamental question: what theorem does a proof prove? While the annotations on a Kaisar strategy make the preconditions and postconditions of a model clear, the Kaisar strategy does not explicitly state what CdGL formula was proved. As in any verification tool, proofs are only useful if they prove an interesting theorem, and theorem statements should be kept simple and readable because they must be trusted. That is, logic cannot prove whether our theorem statement is an interesting one; that is up to the author.

While a Kaisar script does not explicitly mention its corresponding CdGL theorem, every Kaisar proof does correspond to a CdGL proof. The Kaisar proofchecker provides clear, rigorous specifications by automatically computing the CdGL formula proved by a Kaisar script as the script is checked. It so happens that every CdGL formula and every CdGL sequent is equivalent to a formula $[\alpha]true$ for some game α, and Kaisar presents conclusions in this format for the sake of regularity and consistency with Chapter 6 [1] [2].

In keeping with the terminology of Chapter 6, the game α computed for the conclusion $[\alpha]true$ of a Kaisar strategy is called the *game reification* of the strategy. A Kaisar proof is accepted by the proofchecker if it witnesses any theorem, i.e., if the strategy has a game reification α where $[\alpha]true$ is proved. By applying Chapter 6, the game reification can be reified further, giving the *system reification* of the strategy. The game reification contains less information than a full Kaisar proof and more information than the system reification: unlike the system reification, it remembers all assertions proved throughout the strategy; unlike the Kaisar proof, it erases the proofs of those assertions. The top-level Kaisar command `conclusion proofName;` displays the CdGL conclusion of `proofName`.

[1] For the purposes of this chapter, we find it is easier to explain the structure of conclusions when the postcondition is uniformly *true*. In the implementation of Kaisar it would arguably improve readability to allow arbitrary postconditions. There is no fundamental reason that the implementation of Kaisar could not do so if desired for convenience.

[2] Readers familiar with normal dynamic logics like dL will recall that every formula $[\alpha]true$ is vacuously valid in normal dynamic logics. In contrast, because CdGL is subnormal, they are not vacuously valid. Every CdGL formula, valid or not, can be expressed in the form $[\alpha]true$ with the equivalences $\langle\alpha\rangle\phi \leftrightarrow [\alpha^d]\phi$ and $[\alpha]\phi \leftrightarrow [\alpha;\{?\phi\}^d]true$.

Reification and refinement allow us to connect a Kaisar strategy both to a high-level CdGL theorem statement and to the synthesis tool of Chapter 8. To check that a strategy proves a given theorem, the user can write a Kaisar file which contains their proof, their game, and a `proves` statement which checks whether the proof proves the given theorem.

```
proof exampleProof = <theProof> end
let exampleGame ::= { ... };

proves exampleProof  "[exampleGame](<formula>)";
```

The statement `proves pf fml` reifies `pf` to some game α, rewrites `fml` as some $[\beta]true$, then attempts to automatically prove $\alpha \leq_{[]} \beta$. Once $\alpha \leq_{[]} \beta$ is proved, then $[\beta]true$ holds in one step by rule R$[\cdot]$ (Chapter 6), so `fml` holds.

Recall from Chapter 6 that the language of refinements is undecidable because it can express all of CdGL, so the refinement search performed by `proves` is fundamentally incomplete and heuristic. In practice, however, Kaisar proofs satisfy a few restrictions under which refinement can be checked by a recursive syntactic procedure. Kaisar proofs typically follow the shape of the corresponding game, up to simple game-algebraic (Goranko, 2003) rewriting steps such a distributivity ($;\mathrm{d}_r$). Kaisar proofs also support *ghost* statements (Section 7.3.5) that indicate which statements should be (soundly) erased before reification. Between judicious use of ghost statements and following roughly the same shape in models and proofs, the implementation of `proves` reduces to a simple structural refinement check which says α may use a more precise connective than β but not vice-versa. For instance, a game model might use a nondeterministic assignment in a controller where a proof would assign deterministically. Differences in assignments are one way that a single game can have many winning strategies. At each leaf, `proves` applies the relevant refinement rule, just like reification in Chapter 6. Likewise, recall that invariant proofs for Demonic ODEs are reified to guarded assignments, which refine the source ODE.

In the event that `proves` fails, Kaisar reports the location of and reason for the refinement failure. Refinement failures are often fixed by marking ghosts appropriately or by weakening or strengthening assumptions so that they match between model and proof. Otherwise, a refinement failure may indicate a true mismatch between the proof and model which the Logic-User wrote. Such mismatches arise in any system where models can be written separately from proofs, but Kaisar offers several solutions.

There are two reasons not to despair about the possibility that the `proves` command will fail when mismatches between model and proof arise. The `conclusion` and `proves` command are meant to be used together to minimize duplicate effort. After writing an initial draft of a proof, the author can use the `conclusion` command to auto-generate a theorem statement that is true but potentially complex. If the author is satisfied with the generated theorem statement because they find it simple and readable enough, there is no need to employ the `proves` command at all and no potential for a mismatch between statement and proof. If the author finds the generated statement complicated, they can still use it as a basis when writing a theorem statement to be checked with `proves`. If the `proves` statement ever fails in a future version of the proof, the author has two ways to explain the failure: they can either consult the error message provided by `proves` or run

`conclusion` and compare the generated statement to the statement written by hand. As in other languages, the Logic-User is also encouraged to use top-level `let` statements to reduce duplication between models and proofs.

The execution of strategies extracted from Kaisar proofs is discussed in Chapter 8. As shown in Chapter 6, refinement simplifies the synthesis of executable code from Kaisar proofs because it enables a reduction: our synthesizer can compute the system reification of a Kaisar proof, then simply synthesize a system. Because the reified system provably satisfies any safety property proved by the strategy, so does the synthesized code as long as the synthesizer can safely synthesize code from the system. Just as our use of refinement allows a reduction from game synthesis to system synthesis, it also enables the Logic-User to build her understanding of game verification on top of her understanding of hybrid systems verification, which crucially helps provide a smooth learning curve for Kaisar.

A second, conceptual advantage of our refinement-based approach with regards to synthesis is that our approach emphasizes the fact that a single game may admit multiple winning strategies, each of which corresponds to different programs and may be useful in different cases. For example, the driving game of Chapter 5 could admit several different strategies among which an aggressive strategy would drive at the highest safe velocity in order to reach its goal quickly, whereas a conservative might drive slowly to preserve fuel or provide comfort to a passenger. Conceptually, a game model serves as a common interface for these multiple strategies. As a result, we expect that our refinement-based approach will ultimately enable the Engineer to, without much effort, switch her implementation between the synthesized code (Chapter 8) from different Kaisar proofs, so long as each proof addresses the same game and thus results in code that admits the same interface.

7.3 Kaisar by Example: Proving Hybrid Games

We now present the full syntax and meaning of Kaisar proofs with a series of example proofs. When reading the examples, note that the Kaisar format allows multiple proofs in one file, so we assume in each case that our examples are enclosed within a proof block (`proof <name> = ... end`) before checking them in Kaisar.

Our examples are intended to provide a gentle introduction, thus a significant portion of the section is expended in providing toy demonstrations of basic syntax before reaching examples that demonstrate fundamental, reusable proof idioms. For these demonstrations of core proof idioms such as logical model-predictive control, safe sandboxing, and reach-avoid correctness, see Section 7.3.8. The reach-avoid correctness example is particularly insightful in that it shows how game logics are a natural setting for the class of reach-avoid properties, a class which is both highly expressive and also well-suited for synthesis. Though the idioms presented in this section are of practical interest, this section emphasizes introducing the Kaisar language and its usage in detail and in context. For an empirical evaluation of Kaisar on concrete case studies, see Section 7.5.

7.3.1 Core Propositional Connectives

Assumptions are represented as Demonic tests or, equivalently, implications. Optionally, the full syntax `?name:(formula);` annotates a `name` for the assumed `formula`. The dual to an assumption is an assertion or Angelic test, which is notated with `!` rather than `?`. Assertions are one of the most fundamental proof constructs because they express the conclusions of our theorems. As the name suggests, assumption statements are equally fundamental for stating the assumptions of theorems. Our syntax for assumption and assertion is an analogy to the widely-used process algebra (Hoare, 1978) operators for receiving and sending. Fact names are used in proof arguments (Section 7.3.2) to refer to a known fact, as opposed to appearing, for example, in later formulas. Assertions are optionally annotated not only with names but also with unstructured proof steps, which will be introduced in Section 7.3.2.

We demonstrate these crucial proof statements using a toy example proof of the arithmetic property $x = 1 \rightarrow x > 0$. In the example, the conclusion $x > 0$ does not need an explicit unstructured proof because it proves automatically. The assumption and conclusion are respectively given the names `xValue` and `xPositive`. In general, names are used to refer to facts in later explicit proofs. Because this example needs no explicit proofs, the names are inessential and serve only as documentation.

```
?xValue:(x=1);
!xPositive:(x > 0);
```

Demonic and Angelic choices are represented with `++` and `switch`, respectively. In the following Demonic choice example, we assume that x is either negative or positive, from which we can conclude that it is nonzero in both cases. While the example considers a toy arithmetic property, branching will be widely used in branching controllers, for which practical proofs will often wish to prove that some property holds uniformly on every branch. Because Kaisar targets the constructive logic CdGL, disequality is interpreted constructively: $x \neq 0$ means that $x = 0$ implies a contradiction.

```
{?xNeg:(x < 0); ++ ?xPos:(x > 0);}
!xNonzero:(x != 0);
```

The previous example demonstrates a subtlety in Kaisar's sound, automatic handling of proof contexts for branching proofs. Because the proof of `xNonzero` is written after the choice ends, it can assume that either `xNeg` or `xPos` holds, but it does not know which one holds. That is, it assumes $x < 0 \vee x > 0$, which suffices to show $x \neq 0$.

The next example demonstrates an Angelic choice (`switch`) following a Demonic choice. In doing so, it demonstrates an even more subtle case of Kaisar's proof context handling: the automatic handling of named facts. On each branch of the Demonic choice, fact name `bit` will correspond to different formulas. When `bit` is mentioned after the end of the choice, it refers to the (constructive) disjunction `x=0 | x=1` because Angel did not get to choose which branch was taken. We call such lookups *disjunctive lookups* because they disjoin facts across branches. Disjunctive lookups will be of practical use for providing concise proofs about branching controllers in Section 7.5. If x is neither 0

nor 1, then Demon fails a test regardless which branch he takes, at which point the game terminates and Angel wins.

The Demonic choice is followed by a `switch` which accepts as its argument the disjunction `bit`. Each `case` has a guard formula, which acts as an assumption within the branch. Not shown, guard formulas support the same fact naming notation `name:(formula)` used in assumptions. A `switch` may omit its argument, in which case Kaisar attempts an automated proof that the disjunction of branch guards is constructively valid, i.e., that the `switch` is total. For example, we accept case analyses which can be decided with CdGL rule splitReal. This example needs an argument because the guards do not cover all program states, let alone constructively. Practical proofs could likewise need arguments to `switch`es for branching controllers whose totality requires assumptions on the program state. The proof of the disjunction of guards determines how Angel plays the `switch`. She plays the first branch whose guard she has proved.

```
{?bit:(x = 0); ++ ?bit:(x = 1);}
switch (bit) {
 case (x = 0) => !nonneg:(x >= 0);
 case (x = 1) => !nonneg:(x >= 0);
}
```

While we used a fact variable as the argument in the example above, the argument can be any forward-chaining natural deduction proof term, which we now discuss in the context of the `note` statement.

The `note` statement is used to write forward-chaining natural deduction proofs, meaning proofs which work forward from known facts available in the context to deduce their conclusion using introduction and/or elimination rules. Proof terms use (uncurried) function-like syntax. The next example is a toy application of the `note` statement which takes the conjunction of two assumptions, using the `andI` rule to introduce a conjunction. Specifically, the formula proved by the `note` statement, after expanding the definition of `square`, is $x < 0 \wedge y \cdot y > 0$. The full list of supported proof rules will be given in Fig. 7.3

```
let square(z) = z * z;
?left:(x < 0); ?right:(square(y) > 0);
note both = andI(left, right);
```

Unlike assertions, `note` proofs are not annotated with a formula for the conclusion, as the proven formula can be inferred from the proof term. In nontrivial and practical proofs, including the reach-avoid example in Section 7.3.8.3 and the case studies in Section 7.5, a major reason to use the `note` statement is that it can perform simple proof steps without writing down a conclusion. The ability to avoid writing the conclusion becomes particularly useful as conclusion formulas grow large. To support conclusion inference for a natural deduction system, the proof terms for certain rules (e.g., `orIL` and `orIR` in Fig. 7.3) may however need to be given formula arguments which act like annotations.

Proof rules (e.g., those for quantifiers) have the standard side conditions for first-order natural deduction. For rules which have a side condition, the side condition is shown next to the conclusion in Fig. 7.3. In `allI`, the side condition is that the universally quantified

variable x must not appear free in any of the assumptions used in the proof term. In Fig. 7.3, the rule `implyW` is named by analogy to weakening, because it ignores the assumption of the implication when proving the conclusion. In rule names, `I` is mnemonic for *introduction* (meaning a new logical operator is proved, or introduced) and `E` is mnemonic for *elimination* (meaning we take a proven formula and consume, or eliminate a logical operator). In rule names, `L` and `R` are mnemonic for *left* and *right*, respectively.

Name	**Arguments**	**Conclusion**
`trueI`	`()`	`true`
`andEL`	`(pf: P & Q)`	`P`
`andER`	`(pf: P & Q)`	`Q`
`equivEL`	`(pf: P <-> Q)`	`P -> Q`
`equivER`	`(pf: P <-> Q)`	`Q -> P`
`notI`	`(pf: P -> false)`	`!P`
`andI`	`(pf1: P, pf2: Q)`	`P & Q`
`orIL`	`(pf: P, Q: formula)`	`P | Q`
`orIR`	`(P: formula, pf: Q)`	`P | Q`
`notE`	`(pf1: !P, pf2: P)`	`false`
`falseE`	`(pf: false, P: formula)`	`P`
`implyW`	`(P: formula, pf: Q)`	`P -> Q`
`allI`	`(x: variable, pf: P)`	`∀x P` (x∉assump.)
`allE`	`(pf: (forall x p(x)), f: term)`	`p(f)`
`existsE`	`(pf1: exists x P, pf2: forall y (P->Q))`	`Q` (if $x, y \notin \mathrm{FV}(Q)$)
`orE`	`(pf1: A | B, pf2: A -> C, pf3: B -> C)`	`C`
`existsI`	`(x: variable, f: term, pf: p(f))`	`∃x p(x)`

Figure 7.3: Kaisar forward-chaining natural deduction proof rules.

The `let` statement introduces function and predicate definitions. Crucial concepts within a model, because they are frequently used, are typically given definitions, which promote readability and compositionality. Realistic examples appear later in our proofs about 1-dimensional driving (Section 7.3.8), which will define crucial concepts such as safe breaking distances and collision-freedom.

Function definitions use the equals sign =, while predicate definitions use the if-and-only-if sign `<->` and programs use the grammar production sign `::=`. As a toy example, we define squaring and nonnegativity to conclude that all squares are nonnegative.

```
let square(x) = x*x;
let nonneg(x) <-> x >= 0;
!allSquaresNonneg:(nonneg(square(y)));
```

We do not require all variables mentioned on the right-hand side of a `let` to be given as arguments. For example, the following is valid Kaisar code and would remain valid even if assignments to `x` were inserted between each proof step. It is equivalent to the previous example except that it shows $x \cdot x \geq 0$ rather than $y \cdot y \geq 0$.

```
let squareX() = x*x;
let nonnegX() <-> squareX() >= 0;
!squareXNonneg:(nonnegX());
```

In Kaisar, the main purposes of definition parameters are to allow different invocations of the same symbol to supply different arguments or to assign concise (parameter) names to complex expressions. Because definitions in Kaisar can mention arbitrary variables on the right-hand side, their closest counterparts in the dL or dGL uniform substitution calculus are functionals (Chapter 2) and predicationals (Platzer, 2017a), not the function and predicate symbols whose syntax in the dL uniform substitution calculus looks like the syntax for defined symbols in Kaisar.

7.3.2 Unstructured Proof Steps

Unlike the examples in Section 7.3.1, most practical Kaisar proofs cannot rely entirely on proof automation and will need some additional guidance. That guidance is provided with unstructured proof steps, the simplest of which are written `by <method>`. The supported methods for assertions include:

```
method ::= auto | prop | rcf | exhaustion | hypothesis
         | solution | induction | proof <theProof> end | guard
```

The method `auto` is used by default and applies propositional introduction and elimination rules before applying an arithmetic solver at the leaves. Method `prop` applies propositional rules only, while `rcf` applies (**r**eal-**c**losed **f**ield) arithmetic only, meaning it invokes an arithmetic solver to decide first-order real arithmetic formulas. Method `exhaustion` proves exhaustiveness of case analyses while `hypothesis` looks for the conclusion in the proof context. Methods `solution` and `induction` are only used in differential equation proofs (Section 7.3.4). The method `proof <theProof> end` allows an assertion to be proved by writing a structured Kaisar proof which proves the assertion formula. We expect the method `proof <theProof> end` to be used less often than other methods, but we expect it to be useful when a single assertion requires a long, involved proof. The `guard` method is only used in `for` loops (Section 7.3.8.3).

Assumptions to a proof step are specified by writing `using <assumptions>` before `by`. Each assumption can be a proof term as in `note`, most often an identifier. If the identifier `x` is listed as an assumption and there is no fact named `x` in the context, then `x` is interpreted as a program variable and all facts which mention `x` are selected. If no assumptions are given at all, the default heuristic selects all facts which mention any free variable of the conclusion. Facts which mention old versions of the free variables of the conclusion are selected, but the selection process is not transitive, i.e., the heuristic does not compute the free variables of selected facts in order to select further facts containing those variables. To use the default selection heuristic in combination with an explicit list of assumptions, ellipses can be put after the assumptions, as

in `using <assumptions> ... by <method>`. Unstructured proof steps are essential when proof contexts grow large, as automated arithmetic solvers struggle with large numbers of assumptions. The `prop` method is particularly helpful for goals that mention complex arithmetic expressions, yet have simple propositional proofs:

```
?a:(x = 0 -> y=1);
?b:(x = 0 & ((z - x*w^2/(w^2+1))^42 >= 6));
!c:(y=1) using a b by prop;
```

7.3.3 Verifying Discrete Programs

So far we have discussed tests, choices, and sequential compositions. In order to support discrete systems (and by extension, games) , we add assignments and loops. As in CdGL, a (Demonic) nondeterministic assignment to variable x is written $x := *$ with a right-hand side of `*`. After an assignment $x := f$ is executed, an equality of the form $x = f$ holds[3], so assumption-like syntax `?id:(x := f);` can optionally be used to give the equality a name `id`.

The Kaisar syntax `x := *` always denotes a Demonic assignment. Kaisar does not have a distinct syntax for Angelic nondeterministic assignments because Angel plays a nondeterministic assignment game `x := *` by computing a new value for `x`. That computation is captured in some term `f` so that playing some Angel strategy for `x := *` is identical to executing (or playing) some `x := f`. While Kaisar's proof language does not distinguish Angelic nondeterministic assignments from deterministic assignments, it may be useful to distinguish the two in a theorem statement. We show how refinement allows the use of Angelic nondeterministic assignments in theorem statements to be bridged with the use of deterministic assignments in proofs, in Section 7.2.2.

The next toy example uses an unstructured proof step to show that $z > x$ holds when z is assigned y right after y is assigned $x + 1$.

```
x := *;
y := x + 1;
?zFact:(z := y);
!compare:(z > x) using zFact ... by auto;
```

Demonic loops α^* are verified by (co)inductive reasoning with some invariant formula Q. The base case is expressed by an assertion immediately before the loop. The assertion identifier is reused as an inductive hypothesis in the loop body, which must end by asserting the same invariant to prove the inductive step. Thus, our syntax requires the invariant to be annotated in two places, which is more verbose than the use of a single annotation `@invariant(<formula>)` in KeYmaera X (Fulton et al., 2015), but importantly provides a place to write proofs of base cases and inductive steps in addition to making it visually clear where base cases and inductive steps fit into system execution.

[3]It is not sound to assume $x = f$ in the general case without renaming, e.g. when assigning $x := x + 1$. In Section 7.4.2, we discuss in greater detail how Kaisar's use of labeling and static single assignment provide a high-level named mechanism for referring to old values of `x` even as the value of `x` changes.

Following the end of the loop, the base case identifier refers to the proven conclusion $[\alpha^*]Q$. Assertions from before the loop (such as `xZero` and `yZero` in the example below) remain in scope after the loop, but `xZero` means only that the initial value of `x` was zero, a distinction which is essential for soundness. Because `y` does not change during the loop body, `yZero` also says that the final value of `y` is zero. Recall that the assumption-like syntax used for the initial assignments to y and x introduces fact names (`yZero` and `xZero`) for the respective formulas $y = 0$ and $x = 0$ which hold immediately after executing the respective assignments.

The below example first initializes x and y to 0, then shows that $x \geq 0$ is an invariant of the loop that (just) increments x, thus $x \geq y$ holds after the loop since y is unchanged.

```
?yZero:(y := 0);
?xZero:(x := 0);

!inv:(x >= 0);
{
 x:=x+1;
 !inductiveStep:(x >= 0);
}*
!geq:(x >= y) using inv yZero by auto;
```

If one wishes to avoid writing the entire invariant formula (`x >= 0`) twice, a `let` statement should be used to abbreviate it.

7.3.4 Verifying Hybrid Games

By adding ordinary differential equations (ODEs) to Kaisar, we complete[4] our support for proofs of hybrid games. Just as in CdGL, differential equation proofs include solution reasoning, invariants, and cuts. The syntax for ODE proofs reflects the syntax for assertions and assumptions, so assertions and assumptions in the domain of an ODE use `?` and `!` notation. Statements in the domain are &-separated and optionally semicolon-terminated. A terminating semicolon at the end of the entire ODE is also optional, but strongly encouraged for the sake of clarity. Differential ghost reasoning is also supported, but its discussion is deferred to Section 7.3.5 where we provide a uniform treatment of discrete and differential ghost and weakening proofs. The simplest ODE proof is written as an ODE system with assumption syntax for the domain constraint, if any. ODE proofs are wrapped in braces to avoid confusion with discrete assignment proofs on part of a human reader. Only a single pair of braces are needed on the outside of the ODE proof, regardless of the number of equations in the ODE system.

The next example initializes $x := 0$ and $y := 2$, then automatically proves that $x+y \leq 4$ holds at the end of an ODE $x' = 2, y' = -1$ whose domain constraint is $y \geq 0$.

[4]Note that there is no separate Kaisar statement for the dual connective α^d, but rather Kaisar provides matching Angelic and Demonic proof constructs for each connective.

```
x := 0; y := 2;
{x' = 2, y' = -1 & ?dom:(y >= 0)};
!xFinal:(x + y <= 4);
```

ODE variables can be labeled with fact identifiers which will be bound to the variable's solution or will report an error if the ODE does not have a polynomial solution. Solution annotations are useful for proofs which rely on the solutions of some but not all variables.

The next example uses the same initial conditions and ODE as the previous one, but instead proves (explicitly) that $x \geq 0$ holds at the end of the ODE. Solution annotation `xSol` expresses the final value of x in terms of its initial value, so `xSol` and `xInit` are used together to show `xFinal`.

```
?xInit:(x := 0); y := 2;
{xSol: x' = 2, y' = 1 & ?dom:(y >= 0)};
!xFinal:(x >= 0) using xInit xSol by auto;
```

Recall that the *differential cut* rule of dL and CdGL allows us to prove that one condition holds throughout an ODE and then use it as an assumption in further proofs. The Kaisar counterpart to a differential cut is an assertion in the domain constraint of an ODE. To prove a differential cut, use assertion syntax in the domain constraint. When a domain contains multiple statements (assumptions or assertions), they are separated by ampersands (`&`) and the proof of each assertion may refer to all of the domain assumptions as well as those domain assertions which appear before it.

Differential cut assertions can only use the `auto`, `induction`, and `solution` methods, of which the latter two can only be used in differential cuts. The `induction` and `solution` methods correspond to differential invariant and solution reasoning in the dL and CdGL proof calculi. When used in a differential cut assertion, the `auto` method uses `solution` if the ODE has a polynomial solution, else `induction`.

As with loop invariant proofs, differential invariant proofs have both base cases and inductive cases. In contrast to a loop invariant proof in Kaisar, it is optional to explicitly assert the base case of a differential cut assertion. If no explicit base case is given, the `auto` method is applied using a slight variant of the default fact selection heuristic: rather than selecting all facts which mention free variables of the goal, the heuristic selects (the facts induced by) assignments to the free variables of the goal. Our tests revealed this heuristic to be useful because it can improve proofchecking speed by selecting fewer facts than the default heuristic, but it still captures the assignments which we will introduce when we implement static single assignment (SSA) in Section 7.3.6, without which few proofs have any chance of succeeding.

Kaisar adopts different handling for base cases in differential invariants as compared to loop invariants for two reasons. Firstly, loop invariant base cases help clarify the definition of the invariant formula in a way not necessary for differential invariant proofs: unlike an ODE proof, the body of a loop may contain many intermediate assertions about many intermediate states, so it is helpful both for implementation and readability to clearly assert the invariant formula at the beginning of the loop in order to avoid any confusion about the definition of the invariant formula. Secondly, base cases of differential invariants can

often be proved automatically, so it is worthwhile to do so by default. While loop invariant base cases may often have automatic proofs as well, base case annotations on loops remain useful for the sake of clarity and are thus required.

The next example initializes $x := 2$ and $y := 0$, then proves $x \leq 2$ and $x \geq 0$ as postconditions of $y' = 1, x' = -2 \,\&\, x \geq 0$. A solution annotation `xSol` is used to name the solution of x. The domain constraint assertion `xSolAgain` demonstrates that a solution for x can instead be asserted explicitly. The assertion `xSolAgain` proves `automatically`, using `solution` reasoning under the hood.

```
?xInit:(x := 2); y := 0;
{y' = 1, xSol: x' = -2 & ?dom:(x >= 0) & !xSolAgain:(x = 2*(1 - y))};
!xHi:(x <= 2) using xInit xSol by auto;
!xLo:(x >= 0) using dom by auto;
```

The syntax for inductive proofs is similar to that for solution proofs. The following example of inductive proofs is a canonical one: circular motion. The assertion `circle` proves the canonical invariant that point (x, y) remains on the unit circle. Circular motion will be essential in practical case studies on 2D driving (Section 7.5) as well.

```
x := 0; y := 1;
{x' = y, y' = -x & !circle:(x^2 + y^2 = 1) by induction};
```

The induction method will by default attempt proofs of both the base case and inductive step. A proof may optionally give an explicit proof of the base case, which `induction` will use instead. This is useful if automation struggles to prove the base case. When automation struggles to prove the *inductive* case, more creativity is required on part of the proof author. If the author knows the invariant to be true but Kaisar rejects the assertion, it is often because the inductive proof relies on another more basic fact about the system, which should be proved by inserting an additional assertion before the assertion that was rejected. If the author knows the invariant to be true but proofchecking hangs, the hang is typically due to automated arithmetic reasoning failing to handle difficult arithmetic goals. The author should simplify arithmetic reasoning, e.g., by:

- selecting a smaller set of assumptions in the proof of the assertion, if there exists a smaller set that entails the assertion,
- minimizing the number of branches in automated proof search, e.g., by minimizing the number of selected assumptions which are disjunctions or which contain function symbols that are defined by disjunction (`min`, `max`, `abs`), and
- identifying, asserting, and using as assumptions any equations which could simplify arithmetic subgoals. For example, if some complicated term appearing in the goal is provably zero, prove that it is so and use that fact to reduce the desired assertion to an assertion whose arithmetic is simpler.

The next example repeats the circular motion proof, but adds an explicit base case.

```
x := 0; y := 1;
!bc:(x^2 + y^2 = 1);
{x' = y, y' = -x & !circle:(x^2 + y^2 = 1) by induction};
```

The examples above are Demonic ODE proofs. We now turn our attention to Angelic ODE proofs. To prove an Angelic ODE, we specify the duration in the domain constraint using assignment notation. The assignment must be to a clock variable, meaning one which is initially `0` and evolves at rate `1`. The clock variable is traditionally named `t`, but need not be. Duplicate duration assignments are disallowed: an Angelic ODE proof must contain exactly one duration assignment while a Demonic proof contains exactly zero. Because Angel is responsible for proving the domain constraint, assumption statements are disallowed in every Angelic domain constraint proof.

The next example considers a single iteration of a time-triggered model of 1-dimensional acceleration-controlled driving where Angel controls the timing. It shows that she can reach the position $x = acc * (T^2)/2$ by running the ODE for the maximum time $t := T$.

```
?(T > 0); ?accel:(acc > 0);
x:= 0; v := 0; t := 0;
{t' = 1, x' = v, v' = acc
  & !vel:(v >= 0) using accel by induction
  & !vSol:(v = t * acc) by solution
  & !xSol:(x = acc*(t^2)/2) by induction
  & ?dur:(t := T)};
!finalV:(x = acc*(T^2)/2) using dur xSol by auto;
```

An assertion can be proved by induction or solution, and must hold at all times from 0 to the duration (here, `T`). The proof of each domain constraint assertion can refer to all preceding assertions; in a Demonic proof, they can additionally refer to assumptions. In both Demonic and Angelic domain constraint proofs, references to other domain constraint facts allow us to assume those facts hold at the current state, where the current state ranges from the solution of the ODE at time 0 to its solution at a time equal to the duration of the ODE. Following the end of the ODE, mentions of a fact variable introduced by a domain constraint mean that the fact holds at the final state of the ODE. In solution-based proofs especially, it is sometimes important to know that the domain constraint holds at all previous times throughout the ODE. In future work, we hope to provide a syntax for truth of the domain constraint at all times throughout the ODE, but providing such a syntax has not been a priority in the current version because of our emphasis on invariant-style reasoning where such a feature has proved less essential in practice thus far.

The reification of ODE proofs into CdGL games (Section 7.2), differs subtly between Demonic and Angelic ODEs. In the Demonic case, the reified domain constraint contains only the assumptions, because Demon is responsible for showing the domain constraint of a Demonic ODE, which Angel can thus assume. The reified domain constraint of an Angelic ODE contains assertions, because Angel is responsible for proving the domain constraint, and indeed the Angelic domain constraint *proof* is also prohibited from containing assumption statements. If the user wishes to exclude an assertion from an Angelic ODE's reification, they will enclose it in a (forward) ghost, which we introduce next. Ghosts are not specific to ODEs, however, and can be applied to any sequence of Kaisar statements.

7.3.5 Uniform Ghost Reasoning

Kaisar provides a uniform notion of ghost proofs, where `/++ <pf> ++/` indicates a forward ghost proof and `/-- <pf> --/` indicates an inverse ghost proof. Intuitively, the forward ghost proof `/++ <pf> ++/` means that `<pf>` is not part of the "real" game, and is added only for proof purposes. Conversely, the inverse ghost proof `/-- <pf> --/` means that `<pf>` is part of the "real" game, but will be ignored in the proof. Our ghosting construct subsumes weakening[5]: a forward ghost assertion corresponds to a cut while an inverse ghost assertion corresponds to weakening. Forward and inverse ghosting are both enforced by the proofchecker: forward ghost assignments are only visible in other ghost assignments and tests, while inverse-ghost facts are only visible in inverse-ghost proofs.

Forward ghosts are used more often in practice, but inverse ghosts are sometimes helpful for performance and lead to a more symmetric language design. Forward ghosts are important in Section 7.2.2 when extracting a CdGL theorem from a Kaisar proof, because ghost statements allow us to specify that we mean to prove some "simpler" game, with extra statements used only for the proof. Forward *differential* ghosts in particular are crucial (Platzer & Tan, 2018, 2020; Platzer, 2012a) for proving invariants which are not inductive, such as exponential decay properties. Inverse ghosts have the opposite meaning: an inverse-ghosted statement is treated as a "real" statement, but is *ignored* in the proof. While Kaisar emphasizes positive references to assumptions, inverse ghosts also provide a negative notion of reference without the maintainability issues of weakening rules: an inverse-ghost fact will be uniformly ignored in all automatic (and manual) steps that follow. Inverse ghosts could also occasionally be useful for reach-avoid proofs: if we wish to prove safety at each iteration, but do not need to remember safety when proving liveness, we may wish to inverse-ghost the proof of safety to improve automation.

The simplest use of ghosting is to indicate that some assertion should be erased once the proof is complete, generally because that assertion was intended only as a lemma. The main practical difference between a ghost assertion and any other assertion is that the top-level `conclusion` and `proves` commands (Section 7.2.2) erase ghost assertions when determining the theorem statement of a proof or checking whether it proves a given theorem, respectively. Operationally speaking, Kaisar assertions are already no-ops because they are assertions that provably pass, thus their erasure never changes the operational behavior of a Kaisar strategy. Like a lemma, a ghost assertion may be used in later proof steps, as shown in the next example. The example, which is contrived for demonstration purposes, assumes a disjunction ($x = 0 \vee x > 0$) stating that x is nonnegative, then proves `abs(x) = x`, using ghost assertions to prove lemmas about each disjunct. The disjunction elimination rule `orE`, which was defined in Fig. 7.3, is used to derive the final conclusion from the lemmas.

[5]Weakening remains a fundamental rule and we do not mean to claim that we have replaced it. We simply mean that we have identified a uniform notation for weakening and ghosting, which allows us to introduce them to a novice in a uniform way.

```
?xSign:(x = 0 | x > 0);
/++
  !xZero:(x = 0 -> abs(x) = x);
  !xPos:(x > 0 -> abs(x) = x);
++/
!absEq:(abs(x) = x) using orE(xSign, xZero, xPos) by hypothesis;
```

Ghost assignments are useful for remembering old values of states, which the next example demonstrates. In practice, remembering old values is often easier using line labels (Section 7.3.7). However, we present ghost assignments both because they help explain line labels and because ghost assignments are an important ingredient for *differential* ghost proofs. The ghost shows that $x > 0$ is preserved by a loop whose body just increments x. The ghost variable y remembers the initial value of x.

```
?xInit:(x > 0);
/++
   ?yInit:(y := x);
   !inv:(x >= y);
++/
{
 x := x + 1;
 /++
   !(x >= y) using inv by auto;
 ++/
}*
!positive:(x > 0) using inv yInit xInit by auto;
```

Note that in the example above, formulas (e.g., assertions) which mention the ghost variable `y` are also ghosted, which is required for soundness (K. Apt et al., 2010). For example, it is sound to assume that $y = x$ is true after `yInit` is executed, but it is certainly not sound to assume $y = x$ in general after erasing the `yInit` assignment. Ghost assertions, unlike ghost assignments, have unrestricted scope: we can and do use ghost facts to prove additional statements which do not mention the ghost variable, such as the final assertion `positive`. Otherwise, ghost assertions and ghost assignments alike would fail their major goal of enabling proofs of statements that would be unprovable in the absence of ghosts.

Our next example shows how a *differential ghost* can be used to add a continuously-changing variable y to an ODE. Variable y must be initialized in a discrete ghost assignment and the right-hand side for y' must be linear in y. Linearity ensures soundness by guaranteeing that the introduction of y does not introduce an infinite asymptote which truncates the existence interval of the ODE. Differential ghosts are often necessary (Platzer, 2012d) to prove invariants that are not inductive. Our example is a canonical one: in an exponential-decaying ODE such as $x' = -x$, the decaying variable x may converge toward 0 while always remaining positive. Under exponential decay, ghosts are crucial to the proof of our non-inductive conclusion $x > 0$. While the invariant of the ghosted system (below, $xy^2 = 1$) is often unintuitive, it can be constructed systematically in many cases (Platzer,

2018a, Ch. 12), notably in automated procedures for checking whether a formula is an invariant of a given ODE (Platzer & Tan, 2018, 2020; Platzer, 2012a). The crucial contribution of the differential ghost variable y' is that it enables an inductive invariant ($xy^2 = 1$) from which $x > 0$ can be proved, whereas the original system did not have any inductive invariant that implies $x > 0$. Our initial choice of $y := \sqrt{1/x}$ happens to assign $y := 1$ when $x = 1$ holds initially, but works equally well for all positive initial values of x.

```
x := 1;
/++
  y := (1/x)^(1/2);
  !inv:(x*y^2 = 1) by auto;
++/
{x' = -x, /++ y' = y * (1/2) ++/
  & !inv:(x*y^2 = 1) by induction};
!positive:(x > 0) using inv by auto;
```

Differential ghosts demonstrate the subtle interaction between the scoping rules for ghosts. The base case assertion `inv` is allowed to mention the ghost variable y because it is a ghost assertion. Because the base case of `inv` is explicitly marked as a ghost, the inductive case of `inv` is automatically treated as a ghost assertion as well, thus allowed to mention ghost variable y. The non-ghost assertion `positive` above is permitted to use ghost facts in its proof (think: cuts or lemmas), but crucially would be rejected if it mentioned the ghost variable y.

It is *not* the case that non-ghost statements in the domain constraint of a ghost ODE can always soundly use the ghost variable of the ODE. Non-ghost assumptions of Demonic ODEs and non-ghost assertions of Angelic ODEs are prohibited from mentioning the ghost variable, because it would otherwise escape its defining scope. For example, these assumptions and assertions are part of the proof's canonical theorem statement (top-level command `proves` in Section 7.2.2), a theorem statement which has no hope of being true if it retains a ghost variable in the domain constraint of an ODE but erases the dynamics of the same ghost variable.

We now discuss inverse ghosts. Whereas forward ghosts represent proof statements not appearing in a game, inverse ghosts represent elements of a game which must not be used in the proof. Our next example shows how inverse ghosts in ODEs can be used to forget irrelevant dynamics in order to optimize (automatic or manual) reasoning about the remaining dimensions of the dynamics. Our example is a 3-dimensional helix where linear motion occurs in dimension z and circular motion occurs in dimensions x and y. We prove that our helical motion has $z \geq 0$ as an invariant after initializing $z := 0$.

Because the x and y coordinates of an object on a helix are uniquely determined by the z coordinate, we expect that many useful proofs about helices could be written which ignore the x and y coordinates and only consider the z coordinate. By using inverse ghosts, we can forget the x and y coordinates and reason about the remaining equation for z. This is useful because the circular dimensions are not solvable by Kaisar, yet the equation for z has a linear solution which enables automated reasoning, such as an automated proof (`zPos`) that the z coordinate remains nonnegative.

```
z := 0;
{/-- x' = y, y' = -x --/ ,  z'=1 & !zPos:(z >= 0) by solution};
```

Based on past proofs about helices in KeYmaera 3 (Peterson & Swanson, 2013), we expect that inverse ghosts would make such proofs easier if they were written in Kaisar.

Though we have not presented examples of them, inverse ghost *tests* would also be useful in principle, for example to indicate that a test should not be selected by automated fact selection heuristics in following proof steps.

We briefly discuss the interactions between forward and inverse ghosts. It is permissible to use inverse and forward ghosts in the same ODE system, in which case forward ghosts are added before inverse ghosts are eliminated. We expect this combination to be rarely-used because forward ghosts are particularly useful for invariant-based proofs whereas inverse ghosts are particularly useful for solution-based proofs. Though an ODE may contain both forward and inverse ghosts, it is an error to nesting forward ghosts inside inverse ghosts or vice versa. We are not aware of any use for nested ghosts that could not be expressed without them. For example, the pattern `/++ proof1; /-- proof2; --/ ++/` would be rewritten as `/++ proof1;++/ /--proof2;--/`.

7.3.6 Static Single Assignment

Before introducing Kaisar's labeled reasoning feature (Section 7.3.7), we discuss static single assignment (SSA) because labeled reasoning makes fundamental use of SSA. A program or proof is in *SSA form* if no variable is assigned twice. In compilers, the *SSA transformation* is a transformation which accepts an arbitrary program and returns a program whose externally observable behavior is identical but which is in SSA form. The SSA transformation is a standard compilation pass because the resulting SSA representation simplifies a wide array of program analyses and optimizations.

In Kaisar, the main motivation for using SSA is that it systematically generates distinct variable names x_i which it uses to remember past values of each variable x. Because labeled reasoning in Kaisar needs to systematically reason about past program state, SSA is a natural representation for proofs which contain labeled reasoning. Kaisar translates all proofs to SSA form as one step of the proofchecking procedure. The SSA transformation for Kaisar proofs is analogous to the SSA transformation for programs, except that all transformations must be applied in proof steps in addition to program statements. For the sake of simplicity, we do not present every Kaisar proof connective, only constructs that correspond directly to some hybrid game, which are the most interesting cases for SSA. To reduce notational complexity, we present the Kaisar proof connectives with fixed rather than arbitrary arity, e.g., we present `switch` statements with only 2 branches and present singleton ODE systems. For connectives which can introduce fact variables, we present the case with no fact variable annotation for the sake of notational simplicity and because SSA does not change the names of fact variables, only program variables. We refer to the subscripted variables x_i as the SSA-variants of x.

The standard SSA algorithm uses a special meta-instruction called a ϕ-node, a variant of assignment which merges two SSA-variants x_i and x_j used for a given variable x in

different basic blocks. That is, if two basic blocks α and β each bind x before jumping to basic block γ and the highest variants of x bound in the respective blocks α and β are x_i and x_j, then γ begins with a ϕ-node which assigns x (i.e., some x_k) to x_i when coming from α and x_j when coming from β. We do not use ϕ-nodes in Kaisar because they do not already exist in the language of games and we do not wish to define their semantics or proof rules. Rather, our transformation simulates ϕ-nodes by implementing their *elimination*, i.e., by equivalently expressing the meaning of ϕ-nodes using standard deterministic assignments across multiple basic blocks. In practice, it is not harder for Kaisar's implementation to process proofs which use this representation: Kaisar's data structures contain metadata which explicitly tag the ϕ assignments so they can easily be detected whenever special handling is necessary. It is important however to recognize that our representation makes the name "SSA" a misnomer, because a variable can be assigned multiple times, not only in distinct basic blocks but in the same basic block. Rather, our representation ensures:

- in each block, each variable has at most one ϕ assignment and one other assignment
- two basic blocks α and β can only contain non-ϕ assignments to the same x_i if α and β are on separate branches of some Angelic or Demonic choice, or if they are separated by the end of a loop.

For purposes of labeled reasoning, reuse of variable names across independent branches is acceptable because the canonical applications of labeled reasoning only refer to past or future states, not states that would rise from hypothetical execution of parallel branches[6]. In principle, it might be useful to ensure unique names across loop boundaries so that after a loop has ended, one can consistently access values assigned during the final iteration of a loop. However, that feature has not been needed so far, so at this time we have not revised our SSA algorithm to ensure uniqueness across loops.

The SSA transformation algorithm maintains a name table S (additional names: R, T) mapping each variable x to an index i where variable x_i is the *current* version of x. Formally, we define x_i to be the current version of x at a given point in the program when i is the largest number such that x_i has been assigned before the given point, or if i is 0 and no x_i has been assigned. Index lookups are written $S[x]$. We call the table S a *snapshot* because it captures a moment in time during execution of a strategy. The input of SSA uses only undecorated variables x and the output uses only decorated variables x_i, starting from x_0 for the initial value of x. Thus, the SSA algorithm assumes an initial snapshot S which maps every variable $x \in \mathcal{V}$ to index 0, i.e., $S(x) = 0$ for all $x \in \mathcal{V}$.

The concept of a snapshot is also useful because it summarizes which variables x_i are current at a given line label, a purpose for which they will be used in Section 7.3.7. The implementation of the SSA transformation inserts the current snapshot as metadata on each label declaration so that the implementation of labeled reasoning (Section 7.3.7) can

[6]Kaisar does allow such parallel references, and an example is even given in Section 7.3.7. The reuse of variable name indices can cause these nonsensical references to elaborate to nonsensical terms, yet soundness is not compromised. In the worst case, parallel references might cause Kaisar to try to prove nonsensical facts which might either happen to be true, or whose proofs would be rejected in order to ensure soundness.

easily access the snapshots through the label declarations.

We are now ready to introduce notations used to define the SSA algorithm. We specify the SSA transformation for Kaisar proofs by a judgement $(\texttt{pf1}, S) \rightsquigarrow (\hat{\texttt{pf1}}, T)$ which holds when the translation of `pf1` with initial snapshot S is the SSA proof $\hat{\texttt{pf1}}$ with final snapshot T corresponding to the program point at which `pf1` or $\hat{\texttt{pf1}}$ ends. Outputs such as $\hat{\texttt{pf1}}$ are annotated with hats. Because the SSA transformation is total and deterministic, the SSA judgement constitutes a function: it is well-moded with input $(\texttt{pf1}, S)$ and output $(\hat{\texttt{pf1}}, T)$. In rules which rely on auxiliary definitions, we write the definitions in the premise, despite the fact that the definitions are not literally premises.

Several notations are used throughout the rule definitions. We write $S[x \mapsto i]$ for the snapshot R such that $R[x] = i$ and $R[y] = S[y]$ for all $y \neq x$. By analogy, $S \uparrow \texttt{pf1}$ is the snapshot R which increments each bound variable of `pf1`, meaning $R[x] = S[x] + 1$ when $x \in \text{BV}(\texttt{pf1})$ and $R[x] = S[x]$ otherwise. We do not present the formal definition of the bound variables of a proof because they are analogous to the bound variables of games. We assume there exist correct SSA transformations for CdGL expressions. The SSA transformations of many CdGL constructs are analogous to the proof cases we present because many CdGL constructs are analogous to some corresponding strategy construct. We write $(\phi, S) \rightsquigarrow (\hat{\phi}, T)$ and $(\alpha, S) \rightsquigarrow (\hat{\alpha}, T)$ for SSA translations of formulas and programs, which can bind variables. Because terms do not bind variables, we just write $S(f)$ for the result of applying SSA to term f in snapshot S. We also assume there exists an SSA translation $S(\texttt{method})$ for unstructured proof methods. In principle, methods can mention formulas which bind variables and thus the method translation can modify the snapshot, but we write method SSA without a snapshot output because in practice the SSA algorithm for strategies would ignore the output snapshot of method SSA. We write $S \cup R$ for the snapshot T which takes the maximum indices from S and R, meaning $T[x] = \max(S[x], R[x])$ for each x. We write $S := R$ for the sequence of assignments which assigns each current variable of S to its corresponding version in R. For a given family $\alpha(x)$ of assignments, indexed by program variables x, let $;_{\{x \mid p(x)\}}\alpha(x)$ be the sequential composition of each $\alpha(x)$ for all x satisfying $p(x)$. Then $(S := R) \equiv ;_{\{x \mid S[x] \neq R[x]\}}\{S(x) := R(x)\}$ is the sequence of assignments $S(x) := R(x)$ for all x whose indices in S and R differ.

In Fig. 7.4, we present the SSA rules for all the major Angelic and Demonic strategy constructs, including the `for` loop construct, which is not explained in detail until Section 7.3.8.3. We present SSA and `for` loops in this order because reading the examples in Section 7.3.8.3 requires an understanding of labeled reasoning, which in turn is based on SSA. We encourage the reader to consult Section 7.3.8.3 before reading the SSA rule for Angelic loops (rule SSAfor).

We discuss the SSA rules presented in Fig. 7.4. Rule SSA? says that SSA transforms assumption statements by applying itself recursively to the test formula. In the case that the test formula contains program variable binders, the output snapshot T will be updated for each binder[7]. Rule SSA:= updates the snapshot S to the snapshot T which increments

[7]Updating the snapshot is not strictly necessary in this case because we only specified that we wish to ensure distinct variable names for *strategies* that bind, but it does no harm to apply SSA in the assumption formula either.

$$\text{(SSA?)}\quad \frac{(\phi, S) \rightsquigarrow (\hat{\phi}, T)}{(?\phi, S) \rightsquigarrow (?\hat{\phi}, T)}$$

$$\text{(SSA:=)}\quad \frac{T = S[x \mapsto S[x] + 1]}{(x := f, S) \rightsquigarrow (T(x) := S(f), T)}$$

$$\text{(SSA:*)}\quad \frac{T = S[x \mapsto S[x] + 1]}{(x := *, S) \rightsquigarrow (T(x) := *, T)}$$

$$\text{(SSA;)}\quad \frac{(\mathtt{pf1}, S) \rightsquigarrow (\hat{\mathtt{pf1}}, R) \quad (\mathtt{pf2}, R) \rightsquigarrow (\hat{\mathtt{pf2}}, T)}{(\{\mathtt{pf1}; \mathtt{pf2}\}, S) \rightsquigarrow (\{\hat{\mathtt{pf1}}; \hat{\mathtt{pf2}}\}, T)}$$

$$\text{(SSA}\cup\text{)}\quad \frac{(\mathtt{pf1}, S) \rightsquigarrow (\hat{\mathtt{pf1}}, R) \quad (\mathtt{pf2}, S) \rightsquigarrow (\hat{\mathtt{pf2}}, T)}{(\mathtt{pf1} \cup \mathtt{pf2}, S) \rightsquigarrow (\{\{\hat{\mathtt{pf1}}; (R \cup T) := R\} \cup \{\hat{\mathtt{pf2}}; (R \cup T) := T\}\}, R \cup T)}$$

$$\text{(SSA}*\text{)}\quad \frac{(\mathtt{pf1}, S \uparrow \mathtt{pf1}) \rightsquigarrow (\hat{\mathtt{pf1}}, R)}{(\mathtt{pf1}^*, S) \rightsquigarrow (\{(S \uparrow \mathtt{pf1}) := S; \{\hat{\mathtt{pf1}}; (S \uparrow \mathtt{pf1}) := R\}^*\}, S \uparrow \mathtt{pf1})}$$

$$\text{(SSA}'\text{)}\quad \frac{R = S[x \mapsto S[x] + 1] \quad (Q, R) \rightsquigarrow (\hat{Q}, T)}{(x' = f \,\&\, Q, S) \rightsquigarrow (\{R := S; \{R(x)' = R(f) \,\&\, \hat{Q}\}\}, R)}$$

$$\text{(SSA!)}\quad \frac{(\phi, S) \rightsquigarrow (\hat{\phi}, R)}{(!\phi\ \mathtt{method}, S) \rightsquigarrow (!\hat{\phi}\ S(\mathtt{method}), R)}$$

$$\text{(SSAswitch)}\quad \frac{\begin{array}{c}(\phi, S) \rightsquigarrow (\hat{\phi}, R_1) \qquad (\mathtt{pf1}, R_1) \rightsquigarrow (\hat{\mathtt{pf1}}, R_2) \\ (\psi, S) \rightsquigarrow (\hat{\psi}, T_1) \qquad (\mathtt{pf2}, T_1) \rightsquigarrow (\hat{\mathtt{pf2}}, T_2)\end{array}}{\begin{array}{l}(\mathtt{switch\ \{\ case}\ \phi\ \mathtt{=>\ pf1\ case}\ \psi\ \mathtt{=>}\ \hat{\mathtt{pf2}}\mathtt{\}}, S) \\ \rightsquigarrow (\mathtt{switch\ \{} \\ \quad \mathtt{case}\ \hat{\phi}\ \mathtt{=>}\ \hat{\mathtt{pf1}}; (R_2 \cup T_2) := R_2 \\ \quad \mathtt{case}\ \hat{\psi}\ \mathtt{=>}\ \hat{\mathtt{pf2}}; (R_2 \cup T_2) := T_2 \\ \mathtt{\}}, R_2 \cup T_2)\end{array}}$$

$$\text{(SSAfor)}\quad \frac{\begin{array}{c}S_0 = S[x \mapsto S[x] + 1] \qquad R = (S \uparrow \mathtt{pf1})[x \mapsto S[x] + 1] \\ (\phi, R) \rightsquigarrow (\hat{\phi}, R_1) \qquad (\psi, R) \rightsquigarrow (\hat{\psi}, R_2) \qquad (\mathtt{pf1}, R) \rightsquigarrow (\hat{\mathtt{pf1}}, R_3)\end{array}}{\begin{array}{l}(\mathtt{for(x:=init;}\ !\phi\ \mathtt{method;}\ ?\psi\mathtt{;x:=inc)\{pf1\}}, S) \\ \rightsquigarrow (R := S; \\ \quad \mathtt{for}\,(S_0(x)\mathtt{:=}S(\mathtt{init})\mathtt{;}\ !\hat{\phi}\ R(\mathtt{method})\mathtt{;}\ ?\hat{\psi}\mathtt{;}\ S_0(x)\mathtt{:=}R(\mathtt{inc})) \\ \quad\ \{\hat{\mathtt{pf1}}\mathtt{;}\ R := R_3;\ \} \\ \quad, R)\end{array}}$$

Figure 7.4: SSA algorithm for strategies.

the index associated with x by 1 and leaves all other indices untouched. The assigned variable x is translated to the fresh variable $T(x)$ while the right-hand side is translated in the old snapshot S because the right-hand side of an assignment is executed before the variable is bound. Because the initial index of every variable is 0, the smallest index ever assigned in an SSA strategy is index 1, with index 0 reserved for the initial values of variables. Rule SSA:$*$ is analogous, but simpler because nondeterministic assignments

have no terms on their right-hand sides. Rule SSA; says the composition $\texttt{pf1;pf2}$ is translated by translating each of $\texttt{pf1}$ and $\texttt{pf2}$ where the final snapshot R of $\texttt{pf1}$ is used as the initial snapshot of $\texttt{pf2}$, then finally returning the final snapshot T of $\texttt{pf2}$ as the final snapshot of the composition. Rule SSA$\cup$ is the first example of rule which must merge two snapshots by introducing renaming assignments which serve the same role as ϕ-nodes in traditional SSA. The branches $\texttt{pf1}$ and $\texttt{pf2}$ are both translated from the same initial snapshot S, resulting in two possibly distinct final snapshots R and T. Recall that $R \cup T$ denotes the snapshot where each index is the maximum of corresponding indices in R and T. Snapshot $R \cup T$ is a natural choice for the final snapshot because it captures all of the indices that are assigned in either branch, but no more. Recall that in each branch, $(R \cup T) := R$ and $(R \cup T) := T$ respectively stand for vectors of assignments which define all the current variables of $R \cup T$ by copying their lower-indexed counterparts from R or T as necessary. Like a ϕ-node, these assignments merge the variables of each branch by ensuring that regardless of which branch is taken, variable $(R \cup T)(x)$ is current for all $x \in \mathcal{V}$ as required because $R \cup T$ is the final snapshot. Rule SSA$*$ starts by using $S \upharpoonright \texttt{pf1}$ to introduce fresh SSA variables for every bound variable of $\texttt{pf1}$. The snapshot $S \upharpoonright \texttt{pf1}$ serves both as the final snapshot of the loop and the initial snapshot of the loop body. The main subtlety in the loop translation is that the body may repeat many times sequentially. At the start of the translated loop, $(S \upharpoonright \texttt{pf1}) := S$ initializes the variables $(S \upharpoonright \texttt{pf1})(x)$, thus serving two purposes. Firstly, the assumptions of the recursive call that translates the loop body are satisfied: the variables $(S \upharpoonright \texttt{pf1})(x)$ are current at the loop body's start, which is essential because $S \upharpoonright \texttt{pf1}$ is used as the initial snapshot of the recursive call that translates the loop body. Secondly, when the loop body executes 0 times, each current variable $(S \upharpoonright \texttt{pf1})(x)$ of the final snapshot contains the value that x had at the start of the loop, which is crucial because $S \upharpoonright \texttt{pf1}$ is the final snapshot of the loop and thus each $(S \upharpoonright \texttt{pf1})(x)$ must contain the final value of the corresponding x, which is also its initial value in the case of 0 loop iterations. After the translated loop body, the variables $(S \upharpoonright \texttt{pf1})(x)$ are assigned again $((S \upharpoonright \texttt{pf1}) := R)$, to restore the property that each $(S \upharpoonright \texttt{pf1})(x)$ contains the current value of the respective x, which is again a crucial assumption to the correctness of future loop executions, or, if we have executed the loop body for its last time, is crucial to ensure that the variables specified by the final snapshot of the loop translation contain the intended final values for the final state of the loop. In summary, the assignments $(S \upharpoonright \texttt{pf1}) := S$ and $(S \upharpoonright \texttt{pf1}) := R$ implement the ϕ-node for the loop. Rule SSA$'$ likewise defines a final snapshot R that allocates a fresh SSA variable for the bound variable x at the start, which is initialized by the assignment $R := S$. The bound variable, right-hand side, and domain constraint of the ODE all use R for their translations. Note the contrast with discrete assignments $x := f$ which use the initial snapshot to translate f: an ODE continuously changes the bound variable x at a rate that can depend on the *current* value of x, not just the initial value, for which reason it is translated using the snapshot R rather than S. The final snapshot T from the translation of domain constraint Q is safely ignored because Q is a formula rather than a strategy. Rule SSA! is like SSA? except that the unstructured proof of the assertion is also translated. The translation of the unstructured proof is written $S(\texttt{method})$, using a function-like syntax which indicates that the SSA translation of unstructured proofs, like

the translation of terms, does not modify the snapshot because neither terms nor methods bind program variables. Rule SSAswitch is like SSA$\cup$ but generalizes the same principle to include guard formulas ϕ and ψ which must be translated. Rule SSAfor extends the basic structure of SSA$*$ to Angelic loops. The intermediate snapshot S_0 captures the index variable assignment `x := init`, while the final snapshot R additionally captures a fresh variable for each bound variable of the body. Before the loop begins, $R := S$ initializes the current variables of R. Every component of the header is translated from the relevant starting snapshot. The final snapshots R_1 and R_2 of the invariant ϕ and guard ψ are discarded because they only differ from R on variables that are bound in formulas, not bound in a strategy. The statement $R := R_3$ is used to merge the final snapshot R_3 of the body with the final snapshot R of the loop. Note that because the snapshots R and S assign different indices to x, the initial assignment $R := S$ includes an assignment to x which is redundant with the initialization of x performed by the `for` loop. While this assignment is redundant, we include it for the sake of consistency with Demonic loops. These redundant assignments only arise for Angelic `for` loops because only Angelic looping has built-in initialization assignments.

We briefly mention the relationship between SSA and refinement-checking. When Kaisar uses refinement to test whether a given proof solves a given game (Section 7.2.2), the proof and game are both normalized to SSA form. So long as the proof and game assign the same variables in similar places, they will use the same SSA names, so that the SSA transformation will not interfere with refinement checking.

7.3.7 Time-Traveling Proofs with Labeled Reasoning

We introduce labeled reasoning, a Kaisar feature which greatly generalizes previous principles for references across states (Section 7.1) by freely mixing references to past, future, and hypothetical states. Labeled reasoning crucially streamlines major CPS idioms such as those presented in Section 7.3.8.

As discussed in Section 7.1, many provers for stateful systems must reason across multiple states, and CPSs are no exception, with several control and proof paradigms that rely heavily on relations between states:

- Model-predictive controllers (e.g. in (Mitsch et al., 2017)) compute the physical behaviors that result from each available control decision in order to select a safe but effective control choice.
- Blackbox sandbox controllers (Bohrer et al., 2018) allow an untrusted source to propose a control choice, but again use predictive reasoning to filter out unsafe choices and apply a fallback controller when safety demands it.
- Differential equations frequently demand invariant (Mitsch et al., 2017) (or variant) proofs that compare the initial and final values of physical variables.
- Stability proofs (Coq example in (Chan et al., 2016)) must track the distance between current and initial positions.
- Liveness (Mitsch et al., 2017) and reach-avoid (Chapter 5) proofs must compare the

value of a termination metric at the beginning and end of a control cycle.

- Some systems (Platzer & Clarke, 2009) pass through several mission phases, making the intermediate states important. Tangential roundabout maneuvers (Platzer & Clarke, 2009) and interplanetary flight paths for spacecraft (Vallado, 1997) pass through multiple control regimes and even different gravitational models.

Past provers have provided notations for remembering past states, usually a single initial or intermediate state (Section 7.1). The motivations listed above demonstrate however that major high-level CPS proof idioms (without exception to other verification domains) require relational reasoning across multiple states which may include not only the current and initial state but intermediate and future states as well. Thus, an elegant proof system for CPS should not only remember initial states, but simplify references to intermediate states and support predictions about future states, within a common framework. We provide a common framework for initial, intermediate, and hypothetical future states by allowing a line to be given a `label`. Labels are introduced by writing `label:` with a colon (`:`) in an allusion to assembly code labels. The value of an expression `expr` at some other location `label` can then be expressed as `expr@label`, a notion which generalizes to predictive reasoning (Section 7.3.7.3).

It is ironic that statement labels, which are one of the world's most detested programming language constructs, raise rather than lower the level of abstraction when recast as a proof construct. Labels in proofs serve a drastically different role from labels in code: rather than encouraging unstructured goto-based programming, proof labels make our structured code more concise and clear by freeing the Logic-User from manually and redundantly computing the value of an expression at different states while enabling human-readable names to refer to those states. Kaisar's support for labeled reasoning is a testament to the value of its structured, persistent contexts: labeled reasoning relies fundamentally on the fact that Kaisar does not forget past state. That automatic structuring is provided in part by a static single assignment (SSA) translation. While SSA is transparent to the Logic-User, we discuss the role of SSA as we discuss labeling.

Terminology. In order to define labeled reasoning operations, we recall definitions of terms from Section 7.3.6 and introduce additional terms. The statement `label:` is referred to as a label statement. Because a label need not appear at the start of a line, we refer to the location of a label statement as a *labeled point* rather than a labeled line. We will interchangeably refer to a labeled point by referring to its label. The expression `e@label` is a *located expression*, which is located at `label`. Point `label` is the *referent* of the located expression while point at which `e@label` is written is called the *referrer*. Recall from Section 7.3.6 that a single variable x from the Kaisar source is translated to a family of subscripted SSA variables x_i. An expression e is *mobile to point* `label` if every free (subscripted) variable x_i of e for which $i > 0$ has been assigned by point `label`. The subscripted variables x_i are crucial for making mobility possible because they remember old values of x throughout the future even if the current value of x gets overwritten[8].

[8]In the case of loops which repeat an unbounded number of times, finitely many x_i naturally cannot remember all values of x throughout the future. When x_i is bound in the body of a loop and loop execution

Subscripts x_0 indicate the (overall) initial value of x and are thus mobile to everywhere. The subscripted variables x_i are called the SSA-variants of x. The *current* variant x_i of x at `label` is the greatest x_i which is mobile to `label`. Our treatment of assignments in this discussion also applies to any assignments implied by a solvable ODE.

7.3.7.1 Historical Proof with Backward Labels

The simplest case of label reference is a backward reference. SSA introduces ghost variables x_i wherever a variable x is assigned, with each SSA-variant x_i remembering the value of x from a different point. In the backward case, the Kaisar proofchecker elaborates a located expression `expr@label` by expanding each variable x to the x_i which is current at `label`. The SSA pass annotates each label declaration with the current snapshot, i.e., with every current x_i, so that we can easily tell which variables x_i are current at each label. Label statements are written `label:` as they are in assembly language. Common uses of backward labels are for referring to the initial state of the game, the initial state of a loop, or the initial state of an ODE.

The next example uses a label to remember the initial state of a loop which increases x and y at different rates, then uses the label to write an invariant which solves y as a function of the current and initial values of x. Invariants which relate current states to previous states are common in practice, and are used even more commonly with ODE proofs than they are with loops.

```
init:
?(y = 0);

!bc:(y = 2*(x - x@init));
{ x := x + 1; y := y + 2;
  !step:(y = 2*(x - x@init));
}*
```

In particular, it is common to prove inductively how some quantity changes throughout the evolution of an ODE. The next example shows that the value of x throughout the ODE $x' = 1$ is always at least the initial value of x.

```
old:
{x' = 1 & !greater:(x >= x@old)};
```

7.3.7.2 Predictable Futures and Forward Determined Labels

The full power of labeled reasoning lies not in the fact that we can concisely refer to past states, but that past, future, and hypothetical states can be referred to with a common construct. At first, references to future states may seem like a magic trick, because one

finishes, x_i stores the value from the final iteration. Because our SSA algorithm does not ensure uniqueness across loop exits, the value of variable x_i could later be overwritten by assignments after the exit which bind the same x_i.

traditionally cannot predict the future. However, the future *can* be predicted in some cases, and *hypothetical reasoning* is possible even when exact predictions are not. Because forward reasoning is subtle, we ease into its discussion by first discussing the *determined* fragment of forward reasoning in this section. For useful applications of forward reasoning, see the discussion of the hypothetical forward reasoning case, which is more general, in Section 7.3.7.3. In contrast to Section 7.3.7.3, the examples in the present section are solely meant to explain to explain how forward reasoning works, not serve as useful applications in their own right. The determined case differs from the full (hypothetical reasoning) case in that we only allow forward references passing over intervening deterministic assignments and tests, as opposed to the nondeterministic constructs such as choices, loops, ODEs, and nondeterministic assignments.

When an expression makes a forward reference `e@label`, the forward reference is resolved by first computing a list of deterministic assignments (called the *difference*) which must be executed on every path from the referrer to the referent. The forward determined case only supports deterministic assignments and tests along that path because the effects of executing other constructs are not reducible to deterministic assignments. The difference is then traversed in reverse and each assignment is applied to `e` as a substitution. Consider the following contrived example:

```
x := 0;
init:
!(x < x@final);
x := x + 1;
x := x + 2;
final:
```

In the above example, the difference between `init` and `final` contains the assignments `x:=x +1;` and `x:=x + 2;`. We unroll the assignments, which yields `x@final = (x@init + 1) + 2 = (0 + 1) + 2`. The essential insight is that $(0 + 1) + 2$ (or even `(x@init + 1) + 2`) is mobile to point `init`, meaning it does not depend on any SSA-variant x_i that is assigned after point `init`.

A referrer inside a discrete choice can have a forward referent outside of the choice and vice versa. The following contrived example demonstrates a referrer outside a discrete choice which makes a forward reference into it:

```
x := 0;
y := x@mid;
init: {
   x := x + 3; mid: x := x * x;
++ x := 5;
}
```

In the above example, y has value 3. At y's binding site, it is unknown whether point `mid` will be evaluated, but it is known that *if* `mid` is reached, 3 will have been stored in x. For that reason, Kaisar static elaborates `x@mid` to `0 + 3`, whose value is 3 *regardless* of whether line `mid` is ever actually executed. The `x := 5` branch of the proof is welcome

to use the expression `x@mid` whose value continues to be 3, it just chooses not to because the `x:=5` branch stands to gain little from reasoning about the parallel branch on which the label `mid` occurs.

The following example shows a referrer inside a discrete choice with a forward referent outside of it. Variable y will again have value 3, because once the binding site of y has been reached it is always the case that the assignment $x := 2$ occurs before finishing the branch, followed by $x := x + 1$.

```
{    y:= x@final; x := 2;
 ++  x := 5;}
x := x + 1;
final:
```

There are important cases in which a reference cannot be resolved:

- when a cyclic dependency is present and
- when a term has multiple possible values.

For example, the following contrived proof is cyclic:

```
x:= x@two; one:
x:= x@one; two:
```

Each line assigns x to the value it receives from executing the other line. In this cyclic definition of x, the value of x on both lines is underdetermined. Cyclic assignments are an error in Kaisar, and are detected by resolving the located term, then reporting an error if the resolved term is not mobile to the referring point.

A term could take on multiple possible values whenever the difference between referrer and referent contains a nondeterministic statement: an ODE, nondeterministic assignment, loop, or choice. In the following 3 toy examples, the located expression is underdetermined and will give an error. In choices and loops, we respectively do not know which lines are executed or how many times, while in nondeterministic assignments and ODEs we do not know which value is assigned when the line runs.

```
y:=x@theEnd;
{x:= 1; ++ x := 2;}
theEnd:
```

```
x := y@theEnd;
y := *;
theEnd:
```

```
x := y@theEnd;
y := 0;
{y' = 2 & y <= 5}
theEnd:
```

As discussed in Section 7.3.7.3, hypothetical reasoning provides a simple alternative when forward references cannot be resolved.

7.3.7.3 Unpredictable Futures and Hypothetical Label Reasoning

We have shown that in general, forward located expressions do not always have unique values, because the future has not yet been decided. Because it is desirable to reason about the future, we instead reason *hypothetically* about the future, asking questions of the form "Suppose x gets assigned y, what is the value of e?" To enable hypothetical reasoning, we allow label statements to take form `label(var1, ..., varN):` where `var1, ..., varN` are program variables. An expression `e` located at `label` is now written `e@label(f1, ... fN)`. Each `varI` is replaced with `fI` during resolution.

Hypothetical assignments overcome the nondeterminism of assignments and ODEs, thus we can now use located expressions `e@label(f1,...,fN)` almost[9] anywhere.

7.3.8 Proof Patterns for CPS

We now demonstrate, by example, how Kaisar in general and labeled reasoning in particular streamline proofs of major recurring CPS proof idioms. Our first idiom is *logical model-predictive control* (Loos, 2016, Sec. 8.1), a proof idiom distinct from but inspired by *model-predictive control* (Mitsch et al., 2017). In control theory, a model-predictive controller simulates a physical model to estimate the impact of, and choose between, each available control action. In *logical* model-predictive control, we predict the physical change resulting from each available control choice to determine a range (or envelope) of safe decisions. From that point, we can either show that a specific control choice is safe or simply monitor the safety of an untrusted controller. Logical model-predictive control is used at greater length in the case studies (Section 7.5), but we give an example here.

The *sandbox* paradigm extends the logical model-predictive control approach by Demonically assigning a control choice and checking it against the model. The sandbox paradigm is crucial for integrating verified models with practical implementations (Chapter 8) because it treats controller implementations, which are potentially complex, as untrusted blackboxes. The sandboxing approach has been used in case studies because of these practical implications (Section 7.5).

7.3.8.1 Logical Model-Predictive Control

To demonstrate logical model-predictive control, we give a 1D driving example where Angel wants to maintain a safe braking distance $SB()$ between her and the goal $(d - x)$ while Demon controls the ODE duration $t \in [0, T]$. Angel predicts the effect of each acceleration in the worst case $t = T$ and allows (`env`) all accelerations that preserve $SB() \leq d - x$. The proof completes with an arithmetic step showing that safety for the worst case $t = T$ implies safety for all $t \in [0, T]$.

Recall from Section 7.3.1 that the nullary function syntax used in this example just means function `SB()` has no parameters; the definition of `SB()` may still This fact gives

[9]The exception is when the difference contains the entirety of a choice or loop that binds a non-hypothetical free variable of e, which is a rare case.

`let` definitions of functions in Kaisar less in common with the dL uniform substitution calculus's (Platzer, 2017a) function symbols and more in common with its functional symbols, likewise for predicate definitions.

```
let SB() = v^2/(2*B); let safe() <-> (SB() <= (d-x));
?bnds:(T > 0 & A > 0 & B > 0);

?initSafe:(safe());
{
 acc := *; ?env:(-B <= acc & acc <= A & (safe() & v >= 0)@ode(T));
 t := 0;
 {t'=1, x'=v, v'=acc
 & ?time:(t <= T) & ?vel:(v >= 0)};
ode(t): !step:(safe()) using env bnds time vel ... by auto;
}*
```

In step `env`, Kaisar successfully automates the resolution of the high-level located expression `(safe() & v >= 0)@ode(T)`. In this located expression, the values of x and v are taken from location `ode(T)`. The resolution of `(safe() & v >= 0)@ode(T)` is non-trivial because it requires determining the values that `v` and `x` will have at location `ode(t)` when parameter `t` is bound to `T` as well as transitively expanding `SB()`. If `safe()` appears in assertion (e.g., `step`) with no surrounding locator, the values of x and v take on their current values from the use site, which can be distinct from the values they had at the definition site.

In this case, the ODE is solvable, so once the duration $t = T$ has been proposed, Kaisar automatically computes the solutions of x and v. In presenting the solutions, we use an explicit multiplication dot $\cdot$ to prevent any confusion between products and variables whose names contain multiple letters.

$$\begin{aligned} x@ode(T) &= x + v \cdot T + \mathtt{acc} \cdot T^2/2 \\ v@ode(T) &= v + acc \cdot T \\ t@ode(T) &= T \end{aligned}$$

so that `safe()@ode(T)` resolves to $(v + \mathrm{acc} \cdot T)^2/(2 \cdot B) \leq d - (x + v \cdot T + \mathrm{acc} \cdot T^2/2)$. For ODEs whose solutions Kaisar cannot compute, x and v would be made into label parameters, for which arguments must be passed manually..

A major way that hypothetical references streamline idioms including logical model-predictive control is that they allow automatically deriving the meaning of a high-level specification like `(safe() & v >= 0)@ode(T)`, thus reducing work for the Logic-User.

Even for ODEs which Kaisar cannot solve, hypothetical reasoning can still be used, specifically by making the bound variables of the ODE into label parameters in this case. While the manual specification of label arguments represents additional work for the proof author compared to the solvable case, labeled reasoning across unsolvable ODEs could still provide two benefits: firstly, variables other than the parameters will still be resolved automatically; secondly, labeling an expression can serve as documentation that an expression is being used to reason about the state indicated by the label.

Though deriving `safe()` requires determining a definition for `SB()`, even `SB()` can be derived from its first principles using logical model-predictive control, as the next example will show. From first principles, we want `SB()` to be the least distance `d-x` for which full braking `B` preserves `safety` throughout the time `ST()` where the vehicle fully stops. We then define `safety` as obeying `x<=d` at the stopping time `ode(ST())`, i.e., the final state of the ODE whose duration was `t`. Lastly, `print(safe())` prints the text of `safe()` to the user, who uses it to define of `SB()`. The first-principles derivation of `ST()` is omitted because it is analogous. Instead, we define `ST` and assert (`stopTime`) that `ST` is the stopping time.

We define the system to be `safe()` if the position x has not exceeded the obstacle d by the stopping time.

```
?(B > 0);
let ST() = v / B;
!stopTime:(v@ode(ST()) = 0);
let safe() <-> x@ode(ST()) <= d;
print(safe());
t:= 0;
{t' = 1, x' = v, v' = -B  & ?(v >= 0)};
ode(t):
```

Kaisar solves the ODE, so `safe()` resolves to $x + v(v/B) + (-B/2)(v/B)^2 \leq d$, which simplifies by algebra to $v^2/(2B) \leq d - x$.

7.3.8.2 Sandbox Control

Next, we extend the predictive model to a *sandbox controller model*. The `switch` statement implements the sandbox, with the fallback guarded by `true` because it is provably safe regardless of `accCand`. The guard `true` ensures Kaisar's `case` totality check succeeds trivially. Each assertion `predictSafe` reasons predictively. The latter assertion predicts motion at time `min(T,v/B)` specifically to capture the case where the system brakes to a complete stop early, i.e., in time $t < T$. The assertion `!step` uses disjunctive lookups for a concise argument: the disjunction of the `predictSafe` assertions implies safety in the worst case, which (by arithmetic) implies safety in every case.

```
let SB() = v^2/(2*B);
let safe() <-> SB() <= (d-x);
?bnds:(T > 0 & A > 0 & B > 0);
?initSafe:(safe());
{ accCand := *;
  let admiss() <-> -B <= accCand & accCand <= A;
  let env()    <-> (safe() & v >= 0)@ode(T, accCand);
  switch {
    case inEnv:(env()) =>
      ?theAcc:(acc := accCand);
      !predictSafe:((safe() & v >= 0)@ode(T, acc));
    case true =>
```

```
      ?theAcc:(acc := -B);
      !predictSafe:((safe() & v >= 0)@ode(min(T,v/B), acc));
  }
  t:= 0;
  {t' = 1, x' = v, v' = acc & ?time:(0 <= t & t <= T) & ?vel:(v>=0)};
ode(t, acc):
  !step:(safe()) using predictSafe bnds initSafe time vel ... by auto;
}*
```

Because Kaisar's underlying logic CdGL is constructive, we briefly discuss the computational interpretation of `switch`. The `switch` above is executed by testing whether `env()` holds and applying the `true` branch only when `env()` cannot be shown true. The `env()` branch is taken whenever possible because its guard allows accelerating, whereas `true` branches must be conservative, e.g. by braking. CdGL compares numbers up to some precision $\delta > 0$ because exact equality is undecidable for its real numbers. The `env()` branch is always taken when `env()` holds by a margin of δ, but either branch can be taken when `env()` holds by a margin in $[0, \delta)$. It is free to use any precision `delta` for the comparison because the choice of `delta` only determines what facts are proved in the case where `env()` *cannot* be confirmed, which is exactly the case where the fallback controller, whose trivial guard is already satisfied, would be executed.

Note that rather than use comparisons over constructive reals, classical VeriPhy (Chapter 3) uses integer intervals while constructive VeriPhy (Chapter 8) will use rational intervals. The need for intervals arises for a similar reason to comparison precisions: sensing and computation will both be inexact. Conversely, constructive real computations are like interval-valued computations which can adapt to dynamically provide arbitrarily high precision upon request. That arbitrary precision is preferable in proofs which desire to prove tight bounds, but presents an implementation gap which will have to be bridged in Chapter 8. Nonetheless, the gap between constructive real comparisons and intervals is smaller than the gap between classical comparisons and intervals (Chapter 3), because both constructive real comparisons and intervals must cope with inexactness.

7.3.8.3 Angelic Loops and Reach-Avoid Reasoning

Our final idiom example is a reach-avoid proof, for which we introduce our final proof construct, `for` loops. Reach-avoid is important because it combines the fundamental properties of safety and liveness. We model a 1-dimensional, velocity-controlled vehicle which stops as close as it can to some goal `d`.

The `for` loop syntax used here formalizes an Angelic loop convergence proof. The syntax is inspired by the `for` loop syntax that is common in imperative languages, but it is distinct from that syntax because our `for` loop headers have four components and those components include an assertion and fact variable binders as opposed to pure executable code. Kaisar's `for` loop construct is inspired by existing exist convergence rules for Angelic loops. Several such rules exist, including CGL's rule $\langle * \rangle$I from Chapter 4 and the dL convergence rule (Platzer, 2008a, Rule G4). We repeat the rules in Fig. 7.5 for reference. Note that rule $\langle * \rangle$I is written with proof terms and named contexts because it is a CGL

rule, while G4 is written without them because it is a dL rule.

$$(\langle*\rangle\text{I})\quad \frac{\Gamma \vdash A:\varphi \quad p:\varphi, q:\mathcal{M}_0 = \mathcal{M} \succ \mathbf{0} \vdash B:\langle\alpha\rangle(\varphi \wedge \mathcal{M}_0 \succ \mathcal{M}) \quad p:\varphi, q:\mathbf{0} \succcurlyeq \mathcal{M} \vdash C:\phi}{\Gamma \vdash \text{for}(p:\varphi(\mathcal{M}) = A; q; B)\,\{\alpha\}\,C:\langle\alpha^*\rangle\phi}\ {}^{1}$$

$$(\text{G4})\quad \frac{\vdash \forall^{\alpha}\forall v{>}0\,(\varphi(v) \to \langle\alpha\rangle\varphi(v-1))}{\exists v\,\varphi(v) \vdash \langle\alpha^*\rangle\exists v{\leq}0\,\varphi(v)}\ {}^{2}$$

[1] $\mathcal{M}_0$ fresh, $\mathcal{M}$ effectively-well-founded
[2] where $\forall^{\alpha}$ denotes $\forall x_1 \cdots \forall x_n$ for $\{x_1, \ldots, x_n\} = \text{BV}(\alpha)$, the bound variables of α

Figure 7.5: Two rules for Angelic loops.

Both rules incorporate invariant and termination reasoning, but do so differently. In $\langle*\rangle$I, a termination *metric* $\mathcal{M}$ is written, an explicit term which decreases after each iteration until some minimum value $\mathbf{0}$ upon which the loop terminates. Formula φ is proved to hold throughout the loop, including at its end. In Rule G4, a fresh, decreasing *index* variable v is used in place of an explicit metric term, then the index variable serves as the parameter of a *variant* predicate $\varphi(v)$, which generalizes the notion of invariant by adding that index parameter. Kaisar, like Rule G4, introduces a loop index variable and shows that it decreases in order to show termination. A `for` loop also contains a guard formula which must be an inequality or conjunction of inequalities. Some conjunct of the guard condition must provide a (provably constant) upper (respectively lower) bound on the index variable. Each loop iteration must provably increase (respectively decrease) the index variable by at least some lower-bounded amount, so that the loop is guaranteed to terminate. Computationally speaking, nondeterministic inexact comparisons over constructive real numbers with some precision `delta` are used to implement guards. We emphasize that the comparison `delta` is *not* the main source of subtlety in the loop header. In the example discussed below, the variable `eps` is used in the header to establish minimum progress `eps/2` in each loop iteration, and it would remain necessary to use `eps` to ensure progress even if comparisons were exact, specifically the use of exact comparisons would cause *no* change in the header text. We will *only* need to discuss the comparison precision `delta` when discussing the state of the proof *after* the loop has terminated. We will ultimately suggest setting the comparison `delta` to `eps` as a default, but we will do so only for the sake of convenience. They remain entirely distinct conceptually.

We will introduce `for` loops with an example, but because the full example is only printed after its (lengthy) discussion, we first preview the example loop header here for the sake of readability of the discussion.

```
for (pos := 0;
    !conv:(pos<=(x-x@init) & x<=d) using epsPos consts ... by auto;
    ?grd:(pos <= d - (eps + x@init) & x <= d - eps);
    pos := pos + eps/2)
```

The `for` header contains the following four steps:

- The header initializes an index variable which will be used to ensure termination. In this example, the index variable `pos` is initialized to 0.

- The loop invariant is proved to hold initially using an assertion. Here the invariant formula `pos <= (x - x@init) & x <= d` says that the loop index is never more than the difference between the current and initial value of `x`, while `x` never exceeds the goal `d`. The base case of the invariant proves automatically from `consts` and `epsPos` in this example. As in Demonic loops, `conv` can be used in the body to refer to the assumption that the invariant holds at the beginning of the body.
- A guard condition is defined, optionally with a name (`grd`). Our example guard condition is a conjunction `pos <= d - (eps + x@init) & x <= d - eps`. The first conjunct is only used to ensure the loop terminates: Kaisar only accepts the proof because the guard provides a bound `d - (eps + x@init)` which Kaisar can automatically prove is constant throughout the loop. The first conjunct is an upper bound on `pos` which is in turn a *lower* bound on `x` per the invariant. Thus, the first conjunct does not provide an upper bound on `x`. The safety and progress arguments both require an upper bound on `x`, so we provide that bound with a second guard conjunct `x <= d - eps`. The role of `eps` in the guards is subtle: the conceptual role of `eps` here is *not* as a comparison `delta`, nor is `eps` the lower bound on the increase in variable `x` per iteration. The minimum progress `eps/2` will be specified in the following bullet point. Rather, the guard `x <= d - eps` was chosen because it will be much easier for the loop body to make progress if there is actually some progress (`eps`) left to be made. That is, the progress made in each iteration is a function of `d - x`, so the guard enforces `eps` as a lower bound on the remaining distance `d - x`, which indirectly makes it possible to establish a lower bound (`eps/2`) on progress. Because `eps` is used to ensure lower bounds on progress, an intuitive moniker would be "the progress `eps`ilon" to contrast it with[10] "the comparison `delta`." The lower progress bound is essential for practical reasons because it rules out undesirable Zeno behaviors and essential for foundational reasons because those Zeno behaviors correspond to unsound ill-founded convergence arguments. While the use of a comparison `delta` did not impact our choice of guard condition, we note that comparisons do play a key role in the execution of a Kaisar `for` loop. Operationally speaking, the loop guard is implemented by checking each guard conjunct using the nondeterministic comparison operator on constructive real numbers with some comparison `delta`. The loop only repeats so long as every comparison proves every conjunct to be true and terminates as soon as the comparison operation fails for any conjunct.
- To complete the termination metric's definition, statement `pos := pos + eps/2` indicates that the index variable `pos` increases by exactly `eps/2` every iteration. Because `eps/2` is positive and constant, the loop eventually terminates when `pos` approaches `d - eps`. Adversarial dynamics introduce significant subtlety into progress arguments. In our adversarial setting, Angel can guarantee that x increases by at least `eps/2` per iteration, but Demon could choose to let it increase by up to `eps`.

[10]When asked to infer a comparison `delta`, Kaisar will search for a progress `eps`ilon and reuse it in order to minimize the number of variables. Thus, the same variable might be used for both purposes in practice, but we emphasize the distinct goals of ensuring progress and modeling decidable comparisons.

Rather than use x as a loop index whose progress varies between iterations, we use an index `pos` which increases at exact rate `eps/2` and then we use the invariant `conv` to ensure that `pos` is a lower bound for x.

The loose relationship between `pos` and x also explains why `x <= d - eps` must be tested in the loop guard. Suppose Angel chose in advance to repeat the loop `(d - x@init)/(eps/2)` times because each iteration is only guaranteed at least `eps/2` progress. If Demon then chose to advance x by `eps` in each iteration, the final value of `x` could unsafely exceed `d`. When Angel cannot perfectly predict the state in advance, it is natural that the guard condition must always consult the current state rather than running for a predetermined number of iterations. On the other hand, loop index variable `pos` remains helpful for establishing progress bounds to ensure termination of the loop.

During development, we experimented with designs of the `for` loop connective which allow reusing state variable x in the progress argument. While it is possible to design such a connective, we found its meaning so subtle and its syntax so misleading that we preferred the present `for` loop construct which, while complex, makes all subtleties salient in its syntax and should not exhibit surprising semantics once the basic syntax has been mastered.

In general, use of the division operator `/` in Kaisar requires care to avoid division by zero and the soundness issues that division by zero can pose. Division by nonzero numeric literals such as `2` poses no potential of division by zero, however.

```
?epsPos:(eps > 0);
?consts:(x = 0 & T > 0 & d > eps);
init:
for (pos := 0;
    !conv:(pos<=(x-x@init) & x<=d) using epsPos consts ... by auto;
    ?grd:(pos <= d - (eps + x@init) & x <= d - eps);
    pos := pos + eps/2) {
  vel := (d - x)/T;
  t := 0;
  {t' = 1, x' = vel & ?time:(t <= T)};
  !safe:(x <= d) using time conv grd ... by auto;
  ?high:(t >= T/2);
  !prog:(pos + eps/2 <= (x - x@init)) using high ... by auto;
  note step = andI(prog, safe);
}
!done:(pos >= d - (eps + x@init) - eps | x >= d - eps - eps) by guard;
!(x <= d & x + 2 * eps >= d) using done conv by auto;
```

The beginning line is labeled `init`. The proofs of safety and liveness demonstrate the rich back-and-forth dynamics available in games. First, Angel makes a control choice for `vel` without knowing the ODE duration t. Angel sets `vel` to `(d-x)/T` so that the goal is reached in the best case and at least `eps/2` progress is made in the worst case. Next, Demon chooses the duration of the ODE, which is equal to the value of t upon termination

of the ODE. Angel proves safety under weak assumptions on timing: $0 \leq t \leq T$. It is essential that Angel proves safety without assuming a positive lower bound on t, as safety ought to hold at all times. However, liveness needs a positive lower bound: if the ODE were to evolve for 0 time, no progress would be made. Thus, Demon announces a lower bound $t \geq T/2$, but only announces the bound after safety is proved. The specific lower-bound $T/2$ is chosen for the sake of simplicity. Angel is then responsible for proving the invariant, which includes both progress (`prog`) and safety (`safe`) invariants in a reach-avoid proof. Specifically, it is proved that invariants `prog` and `safe` will hold *after* the update `pos := pos + eps/2` is executed, thus the assertion `prog` writes `pos + eps/2` where `conv` wrote `pos`. The proof of `safe`ty appeals to `conv`, which, when accessed from the loop body, supplies the fact that the invariant held at the beginning of the body. The `note` statement shows the invariant by conjoining `prog` and `safe`.

Recall that nondeterministic, inexact comparisons on constructive reals are used to check termination. Such comparisons prove disjunctions: in this example, the comparison of `x` against `d - eps` proves either `x <= d - eps` or `x >= d - eps - delta` where `delta > 0` is the precision with which we chose to compare the terms. The loop continues so long as the conjunction returns the first branch, correspondingly we learn that the second branch holds upon termination of the loop. We call that second branch the weak negation of the guard because it is the inflation of the negation of the guard by some `delta`. Immediately after the end of the loop, a special proof method called `guard` (not to be confused with fact name `grd`) can be used to learn that the weak negation of the guard holds. The major subtlety in implementing the weak negation is that until this point, the loop has not specified a comparison delta, but the weak negation formula will be phrased in terms of a comparison delta. We implemented the special `guard` method (as opposed to, say, automatically making the negated guard available in some fact variable) as a way to provide control over the comparison delta. Used immediately after the end of the loop, the `guard` method optionally accepts an argument (with the syntax `guard(delta)`) in which case `delta` is used as the comparison delta of the loop. If the optional argument is omitted, then the `guard` method attempts to choose one heuristically by looking for variables mentioned in the guard which are constant and positive, in this case `eps`. The "positive, constant" heuristic is a natural one both because positive comparison deltas are required for soundness and because "progress `eps`ilons," such as the variable `eps` in our example, are typically constant and positive. While progress `eps`ilons and comparison `delta`s are distinct concepts, the heuristic allows using a single variable for both concepts as a simplification, unless the proof author specifically wishes otherwise.

For examples such as ours with conjunctive guards, the weak negation of a conjunction is a constructive disjunction of weak negations, e.g., the weak negation of the guard formula (`grd`) is `pos >= d - (eps + x@init) - delta) | x >= d - eps - delta` in our example for the given comparison `delta`, that is, `eps`. The disjunction holds constructively whenever one of the guard conjuncts fails: if the first conjunct of the guard condition fails, the first disjunct of the negation holds, likewise for the second. In this example specifically, `delta` is automatically defined equal to `eps` by the heuristic of the `guard` method. The proof ends by asserting upper and lower bounds on x, corresponding to safety and liveness.

This concludes our presentation of Kaisar by example, the first proofchecking language and tool for CdGL. Labeled reasoning was a major novel feature which simplified key proof paradigms for hybrid games. At the same time, Kaisar's design combined high-level features including definitions, proof terms, ghosts, and lexical scope in the challenging context of hybrid games.

7.4 Implementation

We discuss key aspects of the Kaisar implementation and how our implementation decisions reflect our CdGL foundations.

7.4.1 Constructive Arithmetic

Recall that classical first-order real-closed-field arithmetic is decidable (Tarski, 1951) in doubly-exponential time (G. E. Collins & Hong, 1991). Conversely, the constructive first-order real arithmetic decision problem is an open question (Lombardi, 2020; Cohen & Mahboubi, 2012). Some arithmetic facts hold classically but not constructively, e.g. trichotomy:

$$x < 0 \lor x = 0 \lor x > 0$$

The meaning of the trichotomy axiom differs between classical and constructive logic because the disjunction operator has a different meaning in each logic. Examples also exist where the different meanings of $\exists$ in classical vs. constructive logic cause a formula to be valid in one but not the other. For example, the following formula is classically valid, as witnessed by instantiating y to 0 when $x \leq 0$ and 1 otherwise:

$$\forall x\ \exists y\ ((x \leq 0 \to y = 0) \land (x > 0 \to y = 1))$$

However, it is constructively invalid. If it were constructively valid, then by the existence property there would exist a witness term (computable function) defining the satisfying value y in terms of x. By construction, that function would be a discontinuous piecewise function, contradicting the assumption that it, being a computable real function, was therefore continuous. Thus, the formula is only valid classically and not constructively.

Thus, it is not sound to apply a classical solver to an arbitrary first-order formula and conclude that it holds constructively. However, there are fragments for which constructive and classical truth agree. A notable and flexible class of formulas on which classical and constructive truth agree is the class of hereditary Harrop formulas (Miller, Nadathur, Pfenning, & Scedrov, 1991), so we restrict the use of classical solvers to hereditary Harrop formulas of real arithmetic[11].

We recall the informal definition of hereditary Harrop formulas. A hereditary Harrop formula is one where $\lor$ and $\exists$ only appear in assumption positions. When an assumption is an implication (e.g. the full formula is $(A \to B) \to C$), the conclusion B is considered an

[11]The standard definition of hereditary Harrop formulas contains a case for atomic predicates. In real arithmetic, we include all comparison formulas in the set of atomic formulas.

assumption and the assumption A is not: because implication is a *contravariant* connective, it reverses the polarity of A and B. Negation is treated in accordance with the constructive definition $\neg P \leftrightarrow (P \to \bot)$, meaning that negation also reverses the polarity of P.

Since classical and constructive truth agree for hereditary Harrop formulas (Miller et al., 1991), Kaisar's arithmetic checker first checks whether its input[12] is a hereditary Harrop formula and applies a classical solver if so, or reports an error otherwise. While many important formulas are not hereditary Harrop, arithmetic solving is supplemented by constructive proof search which aids in proof of disjunctions. Kaisar's standard proof heuristics apply proof search to eliminate $\vee, \to, \leftrightarrow$, and $\neg$, then apply solvers to the simplified goal. If one wishes to prove a goal that is not hereditary Harrop because it contains an existential quantifier in conclusion position, manual proof steps can be used to eliminate quantifiers, resulting in a hereditary Harrop formula. Specifically, the `existsI` rule for forward-chaining proof terms can be used (Fig. 7.3).

7.4.2 Contexts

Context management is an essential aspect of any proof calculus, Kaisar included. Fact names are an important feature in Kaisar, but naming is also a standard feature in any natural deduction calculus.

However, the Kaisar proofchecker expects significantly more structure: we must remember which variables have changed since a fact was proved, local definitions, ODE solutions, and other structural data. When looking up an assignment or assertion, we may wish to know contextual information like whether it was a ghost and whether it was part of a looping or branching proof. Our design solution is surprisingly simple: a context is simply a Kaisar syntax tree with extra metadata nodes. Kaisar proofs easily maintain structural information such as branching structure, ghosting, and sequencing. When a sequence of Kaisar statements is checked, each statement is appended to the context. The context remembers when a looping proof is in progress, both so that the inductive hypothesis can be handled and in order to soundly remember which variables might have been bound in previous loop iterations. Metadata can be generated during proofchecking and exploited in other proof steps: ODE proofs can remember their solutions, while `note` remembers its conclusion. By virtue of being a context, we remember that every assertion in the context is already proven, and we do not recheck assertion proofs when looking up a proven fact.

Recall from Section 7.3.6 that Kaisar translates proofs into SSA form before proofchecking in order to elaborate line labels. SSA serves a supporting role in context management by making contexts persistent. In a non-SSA proof, an assignment can invalidate the truth of a fact which was previously proved: e.g., if $x = 0$ is proved and $x := 1$ is then assigned, the fact $x = 0$ no longer holds. The SSA transformation would (for some i) prove $x_i = 0$ before assigning $x_{i+1} := 1$, which does not invalidate truth of $x_i = 0$.

[12]The input is a constructive sequent, which Kaisar converts to a single formula before checking that the resulting formula is hereditary Harrop.

7.5 Results and Evaluation

We evaluate Kaisar against Bellerophon. We ported the 1D and 2D driving models from Chapter 3 (PLDI-DC (Bohrer et al., 2018) and RA-L (Bohrer, Tan, et al., 2019, Thm. 1)) from Bellerophon to Kaisar, as well as an earlier 2D driving model from the literature (IJRR (Mitsch et al., 2017, Thm. 1) which inspired the theorem of Chapter 3. After porting the above models, we then generalized the PLDI-DC model in four stages (PLDI-AS, PLDI-TAC, PLDI-RA, PLDI-RAD), which we back-ported to Bellerophon to provide additional porting experience and data for the reverse direction.

Recall that the Logic-User's goals for Kaisar include maintainability, traceability, and readability, and that the Engineer also stands to benefit from the robust implementation of synthesis that a structured language could provide. Some of the Logic-User's goals, such as traceability and readability, are subjective and are thus better appreciated by reading the examples from previous sections, rather than through empirical evaluation. In particular, the value of features such as fact naming is considered to be self-evident because we consider it common knowledge that reading a variable name is easier than reading a numerical index. The Engineer's goal of a new synthesis implementation will be satisfied by implementing one in Chapter 8. In that chapter, 1D and 2D Kaisar proofs will be reused to evaluate the synthesis tool as well.

Because our goal of maintainability, in contrast, *is* amenable to an empirical evaluation, we provide one. We use the PLDI series of models (PLDI-AS, PLDI-TAC, PLDI-RA, PLDI-RAD), each of which builds upon the previous one, to simulate proof maintenance under lab conditions. In each of Bellerophon and Kaisar, we measure how many lines of changes, and what kind of changes, each proof of the series required with respect to its predecessor. Though we prioritize goals such as maintainability over properties such as conciseness, we also measure the length of each model and proof as an inexact proxy for the Logic-User's productivity. The IJRR and RA-L models are used primarily to assess Kaisar's ability to scale to larger examples, rather than maintainability.

Line counts, on which we will base our evaluation, require careful analysis. It is encouraging that many Bellerophon proofs got shorter in Kaisar. One obvious, yet noteworthy, reason for shorter artifacts in Kaisar is that the use of annotation-based proofs in a single artifact eliminates the potential for duplication between models and proofs. To draw further conclusions, further interpretation is needed, particularly because short code in a given language does not universally reflect higher productivity. This can be the case either when languages have incidental syntactic differences or when a verbose feature nonetheless promotes productivity. For example, Kaisar consciously chose named assumptions for readability, even at cost of verbosity. Expert users' line counts can also differ from typical users. Though these limitations must not be ignored, we mitigated them by providing detailed counts of each line's purpose to provide insight into the relative effort expended on tasks such as modeling, initial proof attempts, and proof maintenance. Likewise, because we have measured the number of changed lines in each model and *in each language*, we are able to gain a sense of the *relative* maintenance effort necessary in each language when adding new functionality to a model and its proof. We do not attempt to measure every conceivable kind of proof maintenance: for example, we do not assess the ease or difficulty

Model Name (Bellerophon)	Lines	Model	Proof	Assump	Same + Diff
PLDI-DC	15	13	3	0	N/A
PLDI-AS	42	15	27	9	9 + 33
PLDI-TAC	39	15	24	5	19 + 20
PLDI-RA	28	19	9	0	10 + 18
PLDI-RAD	29	20	9	0	27 + 2
IJRR	88	36	52	4	N/A
RA-L	294	67	227	97	N/A

Model Name (Kaisar)	Lines	Model	Proof	Assump	Same + Diff
PLDI-DC	7	4	3	0	N/A
PLDI-AS	10	7	4	0	6 + 4
PLDI-TAC	9	7	4	0	7 + 2
PLDI-RA	15	11	6	0	2 + 13
PLDI-RAD	17	12	7	0	12 + 5
IJRR	62	31	31	12	N/A
RA-L	491	168	323	135	N/A

Table 7.1: Proof metrics for Bellerophon proofs and Kaisar ports.

of merging conflicting changes which were made in parallel to a proof.

The fact that the Kaisar models and proofs were performed by the language's developer is a clear limitation of the evaluation, but a well-motivated one. The main alternative, a user study, would have its own problems. A user study would necessarily involve new, lightly-trained users, but we expect such users would not reflect the behavior of the Logic-User, who we expect will ultimately acquire nontrivial Kaisar expertise in practice. Because CPS verification has a relatively small and specialized user base, it may also be impossible to establish statistical significance in any user study.

We now present the data from our evaluation. Line counts are given in Table 7.1, with Bellerophon results in the first table and Kaisar in the second. The full Kaisar and Bellerophon scripts are given in Appendix D.1. All the models and proofs are also available in an archive file which accompanies this thesis. The Bellerophon proof of the RA-L model uses KeYmaera X version 4.7, the rest use version 4.9.2.

Specifically, we measure lines of model, total lines of proof, and lines of proof which specify the assumptions passed to proof automation. Sizes are given in non-blank, non-punctuation, non-comment lines. The total line count can be less than the sum of modeling and proof lines because the same line may contribute to both the model and proof. Assertions are counted as proof but not model. The (Assump) column counts all lines which contain `using` clauses in Kaisar and `hide` (weakening) steps in Bellerophon. The `hide` steps specify that a given assumption should be *unused* by proof automation. Line counts are based on 70-character lines, except we counted atomic proof steps in the Bellerophon version of RA-L because the proof line count would be inflated to at least 340 (thus inflating the total count to at least 407) by line-breaking because such verbose proof steps are used. That is, we err on the side of reporting low counts for Bellerophon to ensure

fairness to the competition. In the difference column (Same + Diff), newly added lines are considered different.

We describe the example models and proofs displayed in Table 7.1. The first four versions of the PLDI (Bohrer et al., 2018) 1D driving model are the same four versions which will be discussed and synthesized in Section 8.5.1 of Chapter 8:

- The *Demonic Choice* model (PLDI-DC) is a direct port of the Chapter 3 model. The controller model is a Demonic choice where the first branch allows Demon to choose the driving rate and the second branch fully, immediately brakes. Demon controls the duration of the system loop.
- In the *Angelic Sandbox* model (PLDI-AS), Demon suggests a velocity first, then Angel uses a `switch` statement to check whether the suggested velocity is safe. If the suggested velocity is not safe, Angel fully, immediately brakes. Demon controls the loop duration.
- The *Timed Angelic Control* model (PLDI-TAC) uses a `switch` statement with no input from Demon: Angel decides whether to drive at maximum speed or brake. The system loop is an Angelic `for` loop which runs for a fixed duration, specifically 10 seconds, which is the same duration used in our other tests.
- The *Reach-Avoid* model (PLDI-RA) differs from the Timed Angelic Control model in that the `for` loop is guarded by distance to the obstacle: the first time that Angel decides to brake for safety, the loop stops. Both safety and liveness bounds are proved.
- The *Reach-Avoid with Disturbance* model (PLDI-RAD) adds Demonic actuator disturbance to (PLDI-RA).

There are also two 2D driving models listed in Table 7.1: IJRR and RA-L, which are named after the publication in which each system was first modeled and proved in dL (Mitsch et al., 2017, Thm. 1)(Bohrer, Tan, et al., 2019, Thm. 1). In contrast to the PLDI series of models, RA-L model is not a direct modification of IJRR model, but is a spiritual successor which addresses the same general topic of 2D driving. IJRR emphasizes its support for inexact sensing and actuation, while RA-L emphasizes relative coordinates, speed limit-following, and inexact waypoint-following. Because RA-L is only a spiritual successor to IJRR, not a direct modification of it, they have almost no lines in common, and thus we do not measure their textual similarity in Table 7.1. The great textual difference between conceptually-related models reinforces the importance of evaluating proof maintenance with a series of models and proofs (the PLDI examples) which were directly built on one another, otherwise one should expect an extremely low amount of shared text.

We discuss proof length. The Kaisar files were shorter, except for RA-L. One important source of the reduction is that Kaisar removed duplication between models and proofs by combining them into one artifact. Bellerophon had the shorter RA-L proof because its case analysis rule excels at deduplicating proofs of differential cuts. We plan to provide the same ability, and thus the shorter proof, in Kaisar by allowing `switch` inside a domain constraint. We report this long RA-L proof as a reminder why the evaluation is important: *every* new language will have cases in which it is less elegant than predecessors, but by

identifying those cases through practical use, one can often identify simple feature proposals that restore elegance.

The PLDI examples were shorter than RA-L in both languages. For PLDI-DC, the only Kaisar proof steps were invariants, to which Bellerophon added one invocation of general-purpose proof automation. No manual assumption reasoning was needed. The greater length of the Bellerophon model largely owes to the greater use of definitions. As discussed below, the remaining PLDI examples had non-trivial proof scripts in Bellerophon and concise, annotation-style proofs in Kaisar.

In Kaisar, the largest change occurred in PLDI-RA because the transition from a timed paradigm to a reach-avoid paradigm affected almost every part of the model, including assumptions, loop structure, controllers, and invariants. As PLDI-RAD shows, later changes may affect fewer lines, with PLDI-RAD only changing two lines of model to introduce actuation disturbance and three lines of proof where assumptions on the disturbance are explicitly used. In PLDI-RA and PLDI-RAD, the use of multiple line labels (for the initial state and the start of the loop body, respectively) allowed terms and formulas to be written in a stable, maintainable way. In PLDI-TAC and PLDI-AS in Kaisar, changes were minimal, because their differences in control schemes are expressible in a few lines. In Bellerophon, the largest change came in PLDI-AS because the proof approach switched from highly-automated proof by annotation to an explicit proof with multiple branches, one for each controller branch. Incidentally, this branching-style proof typically increases the number of `hides` in Bellerophon. In principle, a shorter proof could be constructed which uses monotonicity reasoning to share one ODE proof between the branches, but the author used only basic reasoning principles in order to better simulate a non-expert user.

In Bellerophon's defense, it too achieved nontrivial reuse with auxiliary definitions, but the parts which changed are telling: small conceptual model changes can require many changes in hides, cuts (assertions), and any proof steps which refer to facts by numeric identifiers. These are specific cases which Kaisar sought to, and did, address. Bellerophon's reuse numbers are strongest in proofs (PLDI-RAD) where most assumptions are used in most proof steps. As model size increases, however, proof steps need to minimize their assumptions for performance reasons. Thus, the advantage fades not only because more `hides` are required, but because maintenance of `hides` is fundamentally non-local, when compared to maintenance of Kaisar-style assumption lists. In Bellerophon proofs for Angelic loops, another contributing factor to proof length is the fact that these loops are proved using convergence arguments, which do not have an annotation syntax in Bellerophon as of this writing.

We discuss the IJRR model. In contrast to RA-L, IJRR had more concise casing structure when expressed in Kaisar. The concise case structure demonstrated the value of Kaisar's disjunctive lookups: each of the 3 control cases proved a correctness lemma, after which a single proof of the ODE is written which automatically appeals to the disjunction of control lemmas. Notably, the Bellerophon proof had fewer assumption-management lines. This reflects the fact that the available assumptions were simple enough that automated solvers could prove the assertions with little manual assumption hiding. While Kaisar had more explicit assumption lines, its assumption syntax has benefits for readability and maintainability while remaining reasonably concise. The Bellerophon and Kaisar proofs

both used printing functionality for debugging: the Bellerophon version had 17 printing statements throughout it, while the Kaisar proof starts with 3 lines of options which instruct the prover how to display (all) statements as they execute.

We discuss RA-L further. While Kaisar's simpler case-analysis connective led to a longer proof, RA-L was a useful stress-test which showed Kaisar's ability to handle nested cases and multiple ODEs across different branches. Many lines had `using` clauses: 135. In practice, the user often starts writing assumptions everywhere once they are used anywhere; it is unclear how many assumption lines were necessary. However, the large amount of space dedicated to assumptions does not invalidate the benefits of positive expression management: assumption lists are easy to read in the sense that their relationship to the surrounding proof and model text is clear; additionally, adding an assumption does not require maintenance to unrelated proof steps. Bellerophon had 97 `hide` statements, fewer than the Kaisar proof, yet a smaller percentage difference between languages than in the smaller IJRR example.

A silver lining of the highly verbose RA-L model in Kaisar is that it serves as a great example of a Kaisar model with non-trivial model structure far beyond the fragment supported in classical VeriPhy. The model features nested cases where the ODE is repeated on each branch (specifically, 6 times) with different invariants, in stark contrast to the control-plant loop idiom required by classical VeriPhy. When code is synthesized from the RA-L model in Chapter 8, it validates the claim that Kaisar admits a synthesis tool which supports the complex proof structure used in this model.

On a side note, the ODE duplication in the Kaisar RA-L proof is an example of a common idiom of GLs and DLs generally: using different proofs for the same program on different branches, as expressed by the derived axiom:

$$[\{\alpha \cup \beta\};\gamma]\phi \leftrightarrow [\alpha;\gamma]\phi \wedge [\beta;\gamma]\phi$$

which follows immediately from the axioms for choice and sequential composition. This DL composition idiom is useful if the proof for γ varies significantly in each case, whereas Hoare-like composition reasoning[13] is often used in DL and Kaisar when the same proof for γ suffices in both branches α and β. In DLs, Hoare-like composition reasoning is definable from monotonicity (rule M in Chapter 4) and the DL axiom for sequential composition.

The Kaisar proof for RA-L also contained a significant number of print statements: 37. Because of Kaisar's emphasis on traceability, viewing the proof state was not a use of the print statements, in contrast to their use in Bellerophon. Rather, they were used to help track proofchecking progress because the proof took significant time (≈10 minutes) to check. They were also used for the incidental reason that early drafts of the proof were written as a Scala string literal rather than a standalone Kaisar file; since Kaisar reported line numbers relative to the start of the string rather than the start of the Scala file, Kaisar's line number reporting was less useful than in a standalone Kaisar file and so print statements were used as a surrogate to help trace the locations of errors.

In addition to the case studies listed here, the Kaisar implementation comes with a test suite which includes all examples from this chapter as well as several dozen unit tests

[13]That is, proving some postcondition ψ of $\alpha \cup \beta$, then proving that the postcondition of γ holds if ψ holds at the start of γ.

for the parser and proofchecker. The test suite for the proofchecker includes soundness tests which attempt to exploit past or imagined soundness bugs, ensuring that the checker rejects these known-unsound proofs.

7.6 Summary

This chapter presented Kaisar, the first proof language for CdGL. To meet the Logic-User's needs, including readability, maintainability, and traceability, Kaisar takes a structured approach. A major new structuring principle was labeled reasoning, which streamlined support for paradigms including logical model-predictive control, sandboxing, and reach-avoid verification. Other structured features, which Kaisar shares with other structured languages, include block structure, persistent named facts, and definitions. The features were supported by novel technical contributions, such as SSA-style variable numbering that supported high-level reasoning across changing states. In whole, these features provide a smooth learning curve for Kaisar, letting users focus more on developing the key insights of their proofs. The basic design of the Kaisar language and its implementation are likely of broader interest beyond CPS, as well.

We evaluated Kaisar against Bellerophon to show how Kaisar's structured features simplified maintenance. As a bonus, the evaluation also showed how Kaisar could scale to models of non-trivial complexity and showed many instances in which the Kaisar proofs were more concise.

In Chapter 8, we will use Kaisar as the input to a new implementation of the VeriPhy synthesis tool which supports games and the synthesis of concrete whitebox controllers.

Chapter 8

Proof-Directed Game Synthesis

This chapter presents a ground-up reimplementation in Scala of the VeriPhy synthesis tool (constructive VeriPhy), which synthesizes monitors and controllers from Kaisar proofs for constructive hybrid games. In addition to being a new implementation, constructive VeriPhy targets a new logical foundation: CdGL rather than dL. We call the core algorithm of constructive VeriPhy "ProofPlex" in an homage to SIMPLEX (Seto et al., 1998) and ModelPlex (Mitsch & Platzer, 2016b) and to emphasize that the ProofPlex synthesis algorithm is fully proof-directed: insights from a Kaisar proof are exploited to drive the synthesis of monitors and controllers.

8.1 Introduction

Before introducing constructive VeriPhy, we first recall the limitations of the original VeriPhy implementation (classical VeriPhy, Chapter 3) in Section 8.1.1 and explain why those limitations were accepted in Section 8.1.2. Next, we show how the limitations are addressed in constructive VeriPhy, in Section 8.1.3 We then discuss how constructive VeriPhy makes different design tradeoffs from classical VeriPhy in order to overcome the limitations of the former. In Section 8.1.4, we additionally report on the limitations that result from the new design tradeoffs.

8.1.1 Limitations of Classical VeriPhy

The following limitations of classical VeriPhy have been identified (Chapter 3):

1. Fixed-precision integer arithmetic requires the Engineer to carefully pick units so that all computations fit in a machine word.
2. Classical VeriPhy requires controllers to be monitored and cannot statically guarantee their correctness, even when a correctness proof is available for the corresponding controller model.
3. Classical VeriPhy can only create sandboxes for programs which fit a certain format, so the Logic-User may be surprised to find that Classical VeriPhy rejects a correct model that does not fit the format.

4. Classical VeriPhy must attempt to guess the structure of the safety proof and attempt to reconstruct a proof that the sandbox model is safe. This reconstruction process is not guaranteed to succeed.
5. Classical VeriPhy's use of conservative fixed-precision interval arithmetic can cause monitors to fail even when the monitor condition generated by ModelPlex would be true if evaluated with precise arithmetic. The Engineer may be frustrated if she frequently encounters errors as a result of the conservativity introduced by the arithmetic representation and, even worse, if those errors are not informative. The solution to such frustrations would be to limit the amount of conservativity and increase the detail of error messages, so their source can be better understood in all cases, regardless of whether their source is conservativity or not.
6. Classical VeriPhy always attempts a fully-automatic proof that the fallback controller is safe. If an automated proof fails, the Logic-User is stuck.

Issue 1 results from classical VeriPhy's use of fixed-precision integer arithmetic, which also makes Issue 5 more severe than necessary: while even constructive VeriPhy will use approximate arithmetic, allowing arbitrarily precise approximations can reduce the frequency of precision-related monitor failures in practice. The remaining issues result from classical VeriPhy's reliance on classical dL and Bellerophon and are exacerbated by the commitment to exhaustive proof artifacts at every step. Issue 2 stems from the fact that safety and liveness are separate theorems in dL[1]: there is no single proof artifact which can readily synthesize both monitoring and control. Issue 3 and Issue 4 both stem from the weak structural relationship between dL models and Bellerophon proofs. Classical VeriPhy's limitation is not that it ignores proof content, but that Bellerophon gives classical VeriPhy a very limited view of proofs: as a proof is checked, classical VeriPhy can see a global, flat list of ODE invariants, but model and proof structure are left to guesswork. This guesswork is bound to fail on complex model structures and thus implies a stated restriction on the shape of models (Issue 3) and unstated restriction on the shape of proofs (Issue 4). On the other hand, Issue 6 stems from classical VeriPhy's desire to fully automate sandbox verification given a system safety proof.

While the specific errors highlighted in Issue 5 arise from conservative arithmetic, Issue 5 also highlights the value of precise, actionable explanations of monitor failures in general. The structural changes intended to resolve Issue 3 and Issue 4 stand to support detailed error messages as well: once we allow systems to contain multiple monitors at multiple places in the model, it becomes even more important to track the relationship between models and monitors so that an error reporting mechanism can report precisely which monitor caused a given failure, even for arbitrarily complex models containing arbitrarily many tests in arbitrary positions.

[1]It is true that dL supports so-called "box liveness" properties of form $[\alpha](\mathit{safe} \wedge \langle\alpha\rangle \mathit{live})$, which approximate reach-avoid correctness. What we mean to say is that synthesis driven by a "box liveness" theorem is not significantly easier than synthesis from separate liveness and safety proofs as the proof is still separated into a safety phase followed by a liveness phase, rather than a single loop which ensures safety and liveness at each step.

8.1.2 Justifications for Limitations of Classical VeriPhy

Though the design decisions of classical VeriPhy, which are discussed above, were costly, they were and still are motivated.

- Fixed-point arithmetic was used because its time and space behavior are extremely predictable, allowing us to argue that our approach extends to hard real-time systems (avionics, nuclear reactors, etc.) and soft real-time systems alike.
- Bellerophon and dL were necessarily used because CdGL and Kaisar did not yet exist, and their development was in fact guided by our experience with classical VeriPhy. Bellerophon and dL remain the more mature language and more widely-applied logic. Even Kaisar, which does not use the core KeYmaera X proof state data structures or any KeYmaera X tactics, acknowledges the maturity of the KeYmaera X implementation by reusing certain syntactic computations from KeYmaera X libraries that were first used in Bellerophon, such as ODE integration, static semantics computations, expression traversal, and substitution.
- Full automation is a laudable goal, and it was only by trying an automatic approach that we could discover the need for manual proofs in practice.
- Formal proof artifacts at every synthesis step provided the Logician with the highest degree of confidence. Providing high confidence for the Logician remains a worthwhile goal that constructive VeriPhy does not currently meet.

  The full cost of these artifacts only becomes apparent after developing and attempting to maintain them. The cost became particularly clear for the verified cross-checker of KeYmaera X proof terms in Isabelle/HOL: our proof exporter was tightly coupled to both KeYmaera X and our Isabelle/HOL formalization, and frequently broke when new axioms were added or even when tactics were changed. Even though the proof term exporter and verified proof term checker were successfully maintained every time the proof term-checking process broke in the past, we expect that they would remain difficult to maintain through the future. Wherever the implementation of KeYmaera X differs from the underlying logic dL, maintenance will be a challenge. For a maintainable proof term exporter and checker to be possible, one would need to eliminate the conceptual gaps between KeYmaera X and dL by extending the dL theory to cover the features needed in practice, reducing the set of features implemented in KeYmaera X, or both.

In summary, constructive VeriPhy not only adopts new design decisions to overcome the limitations of classical VeriPhy, but it fundamentally targets a different tradeoff which has its own weaknesses: it gives greater attention to the Engineer and Logic-User, but asks the Logician to be satisfied with a tool whose logical foundations are clear but which does not come with a rigorous proof of correct code extraction, let alone an exhaustive chain of proof artifacts. A major reason we choose to give the Logician so few guarantees in constructive VeriPhy is that constructive VeriPhy seeks to be a more maintainable implementation which is open to future feature additions, but the rigorous proof and thorough formal artifacts provided by classical VeriPhy work best when implementations change rarely, since they both need extensive maintenance when foundations or features change.

8.1.3 Goals of Constructive VeriPhy

Major new goals for constructive VeriPhy include supporting games and controller synthesis as well as making end-to-end verification sustainable and accessible by providing greater flexibility in modeling, proofs, and arithmetic. As of this writing, our goal of increased flexibility in models, proofs, and arithmetic requires flexibility in the implementation of constructive VeriPhy because new flexibility needs on part of the Engineer and Logic-User will continue to be discovered over time as new practical applications are explored. Barriers to a rigorous proof of correct code extraction include the complexity of the Kaisar proofchecker and the rich combination of features that the CdGL semantics would demand from a host theorem prover (as discussed in Chapter 5). Nonetheless, classical VeriPhy has successfully proved the concept that a chain of formal artifacts can be provided, which raises the hope of a formal proof and artifact chain for constructive VeriPhy if its design and implementation stabilize and its trusted code base is reduced. While the correctness arguments presented in this chapter are informal, they are directly inspired by the correctness proof for classical VeriPhy.

Our design and implementation decisions differ thusly from classical VeriPhy:

- Constructive VeriPhy uses rational arithmetic, which nearly eliminates errors related to finite arithmetic precision while remaining amenable to verification in principle. The decision to use rational arithmetic is discussed further in Section 8.3.3.1.
- Constructive VeriPhy uses CdGL as its foundation and Kaisar as its input format; the former provides a unified framework for control and monitoring while the latter provides the necessary structure for a robust, maintainable implementation. As a result, constructive VeriPhy supports a far wider range of models: any weak-test[2] model is supported, rather than only a fixed template.
- Constructive VeriPhy makes controller sandboxing entirely optional, since whitebox controllers can be synthesized instead.
- When constructive VeriPhy does create a sandbox, the approach is different from classical VeriPhy: we generate a Kaisar file containing a sandbox model and automated proof attempt. Like classical VeriPhy, the sandbox proof is not guaranteed to succeed, but unlike classical VeriPhy, the Logic-User can easily edit the sandbox proof if needed. The constructive VeriPhy sandbox generator will even suggest a fallback controller if the Logic-User does not specify one, though we expect fallback controllers to be specified manually in practice because the heuristic for fallback generation is extremely simplistic.

8.1.4 Limitations of Constructive VeriPhy

Of constructive VeriPhy's design decisions, the most limiting is the omission of a rigorous correctness proof for the synthesis process and absence of accompanying full proof artifacts. To provide the same level of rigor as classical VeriPhy, we would need to prove that:

- every Kaisar proof has a corresponding CdGL proof term,

[2]meaning that tests contain no modalities or quantifiers

- the CdGL soundness and reification theorems hold in a proof assistant, and
- the execution backend soundly implements system execution.

We reiterate that formal proof artifacts are useful for the Logician, but that the constructive VeriPhy implementation does not pursue them at this time because we expect the implementation to continue to evolve. We believe it is natural to declare a formal artifact chain for constructive VeriPhy as outside the scope of this thesis for the following reasons: *i)* constructive VeriPhy reflects different priorities from classical VeriPhy, *ii)* we believe the effort involved in a formal artifact chain is extensive, and *iii)* we believe that because classical VeriPhy has already proved the concept of a formal artifact chain for synthesis, providing a second artifact chain for a new implementation can take lower priority. One major hurdle to an artifact chain for constructive VeriPhy would be the fact that the underlying type theory of the CdGL refinement calculus is known (Section 6.1) to have limited support in modern proof assistants and to require significant formalization tricks. Additionally, generalizing classical VeriPhy's original CakeML-based execution backend for use in constructive VeriPhy would require significant HOL4 expertise beyond the scope of this thesis. However, we remain hopeful that the design of the present work has left open a pathway for this future work. Our backend has a simple, well-defined interface to enable future integration with verified components. The Kaisar proofchecker is structured to elaborate away high-level constructs, so that the core language has a clearer connection with the CdGL calculus and could more easily be integrated with it.

The remainder of this chapter discusses implementation, modeling concerns, sandbox generation, and evaluation. In Section 8.2 we outline the usage workflow, architecture, and informal correctness argument of constructive VeriPhy. In Section 8.3 we discuss the implementation of constructive VeriPhy, starting with the sandbox generator and synthesis passes before detailing data structures for representation of strategies and runtime state. In Section 8.4 we discuss our experience with the process of adapting verified models for synthesis and compare that process with classical VeriPhy. The models used in this chapter are the Kaisar (Chapter 7) ports and generalizations of the driving case studies from Chapter 3. In Section 8.5 we also compare the efficacy of the synthesized code against classical VeriPhy, for example the frequency with which monitors fail and whether sandboxing interferes with overall system objectives. This chapter does not have a related work section because the related work discussion of Section 3.1 applies here as well.

8.2 Design

We discuss the design architecture of constructive VeriPhy. We first give a step-by-step outline (Section 8.2.1) of the workflow for using constructive VeriPhy and all the steps in its synthesis pipeline. We then give a step-by-step outline (Section 8.2.2) of an informal correctness justification for each step of the pipeline.

Justification of liveness guarantees will rely on a notion of *total controllers*: to transfer liveness guarantees to implementations, the controller should have no Demonic tests (corr. no controller monitor) because Demon test failures would cause the system to terminate before reaching its liveness objective. While plant monitor failures would also cause early

termination, we allow them because they are so fundamental; rather, we argue that liveness guarantees transfer to implementations if plant monitors pass.

8.2.1 Workflow and Pipeline Outline

We enumerate the many phases of the workflow, of which the *Synthesis* step invokes a further pipeline of automated steps. We give only a high-level outline here, with details discussed in Section 8.3. The names for each step are italicized.

- The input of the pipeline is a Kaisar proof (`pf`).
- *Sandboxing* (Section 8.3.1) is optionally used to insert a fallback into `pf` and output a *total-controller* model `pfSb`, i.e., one with no Demonic tests (monitors) in the controller. Any proof attempts for the fallback are based on automation and thus are not guaranteed to succeed.
- *Manual editing* by the Logic-User fixes any failing proofs in `pfSb`. We call the resulting model `pfTotal` because it should be total if transfer of liveness guarantees is desired. If the fallback controller is not live, the Logic-User should either write a new, live whitebox controller or only prove safety.
- *Conclusion checking* means the Logic-User optionally writes a CdGL theorem statement $[\alpha]\phi$ and uses the `proves` command to confirm that `pfTotal` proves it. Else the `conclusion` command can be used to display the auto-generated CdGL theorem statement of `pfTotal`.
- *Synthesis* is where the Logic-User invokes the `synthesize` command on `pfTotal`, which initiates the ProofPlex algorithm for automated strategy extraction. The following steps describe each stage of synthesizing and ultimately executing a strategy.
  - *Closure* returns the universal closure `pfClosed` which prefixes the proof with Demonic assignments to all of its free variables.
  - *Erasure* translates `pfClosed` to an intermediate representation `angelStrat` of Angel's strategy (Section 8.3.3.1) with proof annotations removed.
  - *Demonification* reifies `angelStrat` into a program `simpleStrat` in a simplified intermediate representation (IR) that uses only Demonic connectives, i.e., it plays a fixed Angel strategy and allows nondeterministic play by Demon.
  - *Semantic transformation* reinterprets `simpleStrat` as a program over rational interval arithmetic rather than real numbers.
  - *Serialization* saves `simpleStrat` and metadata from the compilation process to a text format for later use.
  - *Interpretation* executes the deserialized `simpleStrat` in an interpreter against a strategy for Demon, provided by the Engineer by implementing a Scala trait (corr. `interface` in Java) called `DemonStrategy` (Section 8.3.4).

8.2.2 Correctness Argument

We give an informal correctness argument for constructive VeriPhy, which could serve as the outline for a future formal correctness proof. The correctness argument follows the structure of the constructive VeriPhy workflow and pipeline. Just as classical VeriPhy showed that safety guarantees hold for the CPS, we seek to show that safety *and liveness* hold for the CPS in constructive VeriPhy. However, we leave a rigorous proof as future work due to the large amount of code involved and the mathematical complexity of the logical foundations. Because the machine-checked proofs for classical VeriPhy constitute over 20,000 lines of Isabelle/HOL code (Bohrer et al., 2018) and because constructive VeriPhy's underlying logic CdGL requires more powerful foundations (Section 5.3.1), an optimistic estimate for the effort required to create a mechanized correctness proof for constructive VeriPhy would be measured in person-months and a conservative estimate would exceed a person-year. Here we only discuss aspects essential to the high-level correctness argument. Implementation details are discussed in Section 8.3.

- *Sandboxing* makes the input proof `pf` into a total-controller sandbox proof `pfSb` by making assumption steps `?x:(P);` into exhaustive Angelic choices, e.g., the `switch` statement `switch { case x:(P) => ... case (true) => ... }`. The output statement is trivially total because a `switch` with a fall-through case with guard `true` is always constructively total. The rest of the pipeline takes a Kaisar model as its input and does not care whether it was produced by sandboxing. Even non-total-controller models are accepted, but totality is required for transfer of liveness guarantees rather than only safety guarantees.
- *Manual editing* is a human step of the workflow which thus does not have its own correctness justification. With regards to system safety, the human edits need not be trusted at all because the Kaisar checker rejects invalid proofs. With regards to liveness, our current implementation trusts the human to preserve totality if a total-controller model with liveness guarantees is desired. In principle, an automated, syntactic totality checker could be implemented by modifying the sandbox generator (Section 8.3.1) to report an error upon encountering a statement that is potentially non-total, as opposed to modifying the statement to become total. We do not foresee any major hurdles in the implementation of such a checker.
- *Conclusion Checking* provides a precise formal theorem statement in CdGL. It is crucial to formally state system guarantees at this step, because the remaining steps wish to transfer those guarantees to implementation level.

  Recall from Section 7.2.2 in Chapter 7 that every Kaisar proof has a *game reification* and *system reification*. The game reification is a hybrid game α which captures the full safety and liveness guarantees, e.g., the game reification remembers all intermediate assertions of the proof. The system reification forgets Angelic assertions, inlines Angel's strategy, and has only Demonic nondeterminism. Also recall from Chapter 7 that Kaisar applies steps like elaboration and SSA-transformation during proofchecking. Proof-checking returns an elaborated proof `pfElab`. The game reification α can directly be read off from `pfElab` by erasing proof annotations.

The simplest conclusion-checking workflow is to read off the game reification α of `pfElab` per the `conclusion` command of Kaisar. Soundness of the Kaisar proofchecker, which we assume, amounts to provability of the CdGL formula $[\alpha]true$ (see Section 7.2.2 for details). By soundness of CdGL (Theorem 5.9 in Chapter 5), $[\alpha]true$ is also *valid* in CdGL. Because α is a game, the fragment of formulas with shape $[\alpha]true$ is not a set of tautologies; but rather can express *all* of CdGL.

Because the formula $[\alpha]true$ is often not readable, the (optional) advanced conclusion-checking workflow applies a Kaisar command `proves pfTotal "[`β`]`ϕ`";` to automatically check that `pfTotal` is a proof of some specific CdGL formula $[\beta]\phi$. The implementation of `proves` consults the elaborated proof `pfElab` and structurally compares it to $[\beta]\phi$. The refinement check performed by `proves` follows the refinement axioms of Chapter 6 closely, especially the axioms for algebraic normalization of game structure and refinement of nondeterministic assignments. The correctness argument for the `proves` command is that it straightforwardly implements the axioms of Chapter 6, which are sound by Theorem 6.1. The implementation of `proves` is by recursion on the proof `pfElab` for the sake of implementation convenience, but the result is that $\alpha \leq_{[]} \{\beta; \{?\phi\}^d\}$ is valid for the game reification α of `pfElab` when the `proves` command succeeds. Then by rule R[·] from Chapter 6 it follows that $[\beta; \{?\phi\}^d]true$ is valid, which simplifies to validity of desired formula $[\beta]\phi$.

- *Synthesis* is divided into multiple pipeline steps, starting from the elaborated proof `pfElab`. We give the justifications for each step. As in classical VeriPhy, one goal is to transfer safety properties: given a safety property $[\beta]\phi$ from the conclusion-checking step, we wish to show all system executions satisfy safety postcondition ϕ. However, we also wish to show that liveness guarantees transfer, which we do by totality reasoning: if the input `pfElab` of the synthesis step is total, then the extracted program has a total controller when interpreted over exact real semantics. Our definition of totality for the extracted program is that assuming any and all initial preconditions of the model hold, no controller tests fail during execution, so that the model always runs to completion unless plant monitors fail. Totality is a strong condition because in practice it rules out Demonic tests, which are typically not total (if total, it would probably have been made an Angelic test in the model). The consequences of totality for liveness are important, however: a typical liveness proof expresses the model as an Angelic loop and asserts that Angel has reached her goal by the end of the loop. When execution reaches the end of the model (thus the end of the loop) and passes all tests (thus the test that the goal is reached), then execution has satisfied the proven liveness property. Thus, total-controller strategies satisfy liveness except in the expected case where Demon violates plant assumptions, causing a plant monitor failure and thus early termination.

  As in Chapter 3, the assumption that plant monitors pass is a non-trivial assumption because it represents the assumption that our model of physics is accurate to real-world physics. However, as in Chapter 3, the assumption that plant monitors pass is absolutely fundamental: if we were unwilling to make assumptions on the physics of a system, it would be impossible to conclude safety of a system's physical behaviors.

Rather, the VeriPhy approach crucially combines verification with validation: by using the plant monitor during experimental evaluation, deviations between the model and implementation can often be automatically identified so that it is then possible to address them.

We now discuss the correctness argument for each individual stage of synthesis.

- *Closure* transforms `pfElab` into closed proof `pfClosed` by prepending a Demonic assignment of each free variable. Safety and totality of `pfElab` are clearly preserved by the introduction rule and semantics for demonic assignment, respectively. Closure is important for an executable interpretation of `pfElab` because it allows the interpreter to ensure all variables are defined before use. While the Closure step changes the model `pfElab` to a new model `pfClosed`, validity of the specification $[\alpha]true$ is preserved, where α is the game reification of `pfElab` or `pfClosed` respectively. Because validity amounts to truth in every state (with a single common proof), all CdGL formulas are equivalid with their universal closures.
- *Erasure* reads off Angel's strategy from `pfClosed` as a program `angelStrat` in an intermediate representation language for strategies (Section 8.3.3.1). Program `angelStrat` is produced by erasing proof information from `pfClosed` and is morally equivalent to its game reification but is expressed in a custom intermediate language for the sake of convenience. The correctness assumption for this step is that the game modeled by `angelStrat` is the same as the game reification α of `pfClosed`, which is a small assumption because both are easily implemented by deleting proof annotations.
- *Demonification* reifies `angelStrat` into a *simple strategy* `simpleStrat` that belongs to the same intermediate language but uses only constructs whose nondeterminism is controlled by Demon. That is, it reifies Angel's strategy in the same way as the reification operator of Chapter 6. The *simple strategy* fragment of the intermediate language is morally equivalent to the discrete fragment of hybrid programs.

  The correctness argument has two parts, one for safety and one for liveness. By the reification transfer theorem (Theorem 6.3 in Chapter 6), `simpleStrat` satisfies the postcondition ϕ from $[\beta]\phi$.

  Just as Theorem 6.3 requires the proof belong to the *system-test* fragment (Demonic tests can only mention modalities if those modalities are box modalities of systems), constructive VeriPhy fundamentally requires the system-test fragment. The current implementation goes further and requires weak tests, which are easier to check dynamically. Under the system-test assumption, the reification refinement theorem (Theorem 6.4 in Chapter 6) implies that the reachable states of `simpleStrat` are a nonstrict superset of those in `angelStrat`, so that totality is preserved.
- *Semantic transformation* reinterprets the semantics of `simpleStrat` over conservative rational interval arithmetic rather than exact real number arithmetic.

The safety correctness justification is by analogy to the interval semantics of classical VeriPhy and their theorem (Theorem 3.7 in Chapter 3) that safety guarantees are transferable. Recall that the inexactness of intervals can arise from several sources. Rounding is required for mathematical operations under which the rationals are not closed, notably square roots. A clever Engineer may also have sensors return intervals whose width is the precision of the sensor so that the interpreter correctly reports the truth value of a formula as unknown in the case that sensor precision is too low to determine its Boolean truth value. The precision of rounding operations is easily controlled as a parameter of the interpreter, while precision of sensors may be limited by the available hardware.

Liveness guarantees only transfer when all tests succeed, i.e., when all case analyses are total when evaluated over all intervals that arise in practice. At this point, it becomes crucial that the CdGL implemented case analysis using the constructive, inexact comparison rule splitReal from Section 5.4.1 rather than exact classical comparisons. To see why, observe that exact comparisons which were total over real numbers are typically not total over intervals. Consider, for example, the classically-valid but constructively-invalid disjunction $x \geq 0 \vee x < 0$. When x is reinterpreted as an interval, the disjunction is invalid because it does not hold whenever interval x contains both negative and positive numbers. Specifically, the disjunction only holds when some disjunct holds, but neither disjunct holds when there *exist* respective elements of interval x which falsify each disjunct. Exact comparisons are a problem for the liveness of code which branches on a disjunction: an interval program which branches on $x \geq 0 \vee x < 0$ will get stuck if neither disjunct holds of interval x, thus compromising liveness by getting stuck.

Even inexact comparisons are not a complete solution for liveness of comparisons. For example, consider the inexact comparison $x > -50 \vee x \leq 0$ which is constructively valid and compares x against 0 with a precision of 50. The disjunction $x > -50 \vee x \leq 0$ fails to hold when x is the interval $[-100, 100]$ which satisfies neither disjunct. Likewise, case analysis over the disjunction $x > -50 \vee x \leq 0$ is not total when x is the interval $[-100, 100]$.

However, totality of constructive comparisons, unlike classical comparisons, can transfer when interval size is *bounded* by the precision, or `delta`, of the comparison. A constructive comparison remains total over interval semantics when restricted to argument intervals whose combined width is less than the comparison `delta`, e.g., less than 50 in the example $x > -50 \vee x \leq 0$. Totality arguments remain subtle because when a comparison is applied to arbitrary *terms*, the required precision for individual *variables* may be tighter than `delta`. For example, if term $10 \cdot x$ is compared to constant 0 with a comparison `delta` of 50, then the implied interval width limit of x is 5. While the computation of variable precision bounds is out of scope for this thesis, we expect it could be automated, especially for typical loop-free control models. By combining automated computation of variable precision bounds with our use of inexact

comparisons, one could provide a conclusive totality proof for synthesized case analyses. Though we leave computing such bounds for future work, our use of inexact comparisons remains crucial both because it enables that work and because the use of inexact comparisons in the input makes both the successful execution of interval comparisons and the existence of sound width bounds far more likely in practice when compared to the use of exact comparisons.

If test failures due to arithmetic rounding are observed, the rounding precision should be increased. If test failures are due to limited sensor precision, the comparison `delta` should be increased, leading to more conservative, but safe and total, control behavior. That is, increasing `delta` provides an easy way to improve the totality of case analyses in practice, even in the absence of formal totality proofs.

- *Serialization* records the `simpleStrat` strategy along with metadata such as a mapping from locations in `simpleStrat` to line and column numbers in `ptTotal` and a mapping from Angelic constructs in `angelStrat` to their Demonic counterparts in `simpleStrat`. The latter mapping will be used to implement Angelic connectives in the interpreter and the former will be used for error reporting. We assume correctness of serialization and deserialization, which are implemented by simple one-pass printing and parsing algorithms for a fully parenthesized strategy language in prefix notation.

  It was important to provide a serialization format for strategies because the alternative would be to check `pfTotal` every time the strategy is needed, a process which is significantly more resource-intensive than strategy parsing, especially in the context of resource-constrained implementation platforms.

- *Interpretation* executes the strategy, a process which has a significant trusted code base. We start by discussing those trusted components which are reused across all invocations of constructive VeriPhy. The interpreter begins by deserializing `simpleStrat`, so the strategy parser is trusted. The interpreter is written in Scala and plays `simpleStrat` against a driver for Demon's strategy which is written by implementing a Scala trait named `DemonStrategy`. While the interpretation algorithm is trusted, it follows closely from the semantics of games which have been validated in depth in earlier chapters, see Section 8.3.3.1 for discussion. The interpreter relies on interval arithmetic algorithms taken from the `spire` library (Osheim & Switzer, 2011–) written in Scala, which are trusted but are considered stable production code. The Scala compiler is trusted. While the Scala compiler is well-tested, stable software, it is of non-trivial complexity.

  Notably, the implementation of the `DemonStrategy` is also trusted: if we want safety and liveness to transfer to the physical world, the `DemonStrategy` must soundly sense and actuate the world. The assumption of `DemonStrategy` correctness is a significant one because it is not a fixed component: every application of constructive VeriPhy typically has its own Demon driver. As in classical

VeriPhy, however, correctness of sensing and actuation is a natural assumption because it is a significant research problem in its own right which is largely orthogonal to the issues of correct extraction of control code and monitoring code that we consider here.

Safety of the interpreter demands that all states reached in the interpreter and the real world are states permitted by the interval semantics of `simpleStrat`, analogous to Theorem 3.9 from Chapter 3. Because the interpreter terminates when tests fail or when memory is exhausted, totality of the execution of the controller in the interpreter requires that memory is not exhausted and that all tests which pass in the interval semantics also pass in the interpreter. We expect that memory is not exhausted in practice. The only variable-size data structures used during execution are rational numbers, but their size would only be a significant concern if it increased indefinitely with each system loop iteration. That has not occurred thus far in practice: controller variables are typically recomputed at each iteration from sensed values and constants, implying a precision bound. The latter assumption, on tests passing, amounts to the assumption that the rational type from `spire` faithfully implements rationals and that the interpreter faithfully interprets terms and formulas.

In summary, the trusted base of constructive VeriPhy is significant, especially the trusted base of the Kaisar proofchecker and interpreter. At the same time, the basic structure of a safety transfer argument for constructive VeriPhy just extends the same basic structure of classical VeriPhy with additional passes. The additional passes of Conclusion Checking, Erasure, and Demonification connect safety guarantees for games in Kaisar to safety guarantees for hybrid systems, so that the basic proof approach of classical VeriPhy can be reused for remaining passes. The argument for transfer of liveness guarantees is new and amounts to showing that totality is preserved: execution of the interpreter always reaches the end of the strategy program and passes all assertions assuming the input model is total and assuming plant monitors pass. The major barrier to formal liveness proofs is that totality requires non-trivial, model-dependent interval precision bounds.

8.3 Implementation

We now discuss the detailed implementation of constructive VeriPhy. We discuss the generation of sandbox models in Section 8.3.1 followed by the ProofPlex synthesis algorithm proper in Section 8.3.2. Language and state data structures are discussed in Section 8.3.3, followed by the execution algorithm proper in Section 8.3.4. Drivers for Demon's strategy and their implementation in Scala are also discussed in Section 8.3.4.

8.3.1 Sandbox Generation

Classical VeriPhy (Chapter 3) uses a sandboxed control scheme where controllers are described nondeterministically by controller monitors and a fallback action is applied if a controller monitor fails. In both versions of VeriPhy, a plant monitor can also fail if the

physical assumptions of the model are violated. Fallback controllers are typically engaged in practice when a plant monitor fails as a best-effort response; when the laws of physics fail, a bulletproof safety guarantee cannot be expected. Formally, we consider execution to have terminated in Angel victory when a plant monitor (Demon test) fails, making both safety and liveness guarantees hold vacuously. That is, non-vacuous satisfaction of safety and liveness of the extracted code crucially rely on plant monitors being satisfied.

Sandboxing is optional rather than mandatory in constructive VeriPhy: sandboxes are redundant if controller correctness has been verified, but sandboxes remain important for executing complex untrusted controllers under the supervision of a verified model. Because constructive VeriPhy seeks to provide a more transparent and controllable development process than does classical VeriPhy, sandboxing is implemented by transforming an input Kaisar model into one which implements a sandbox, which could then be further viewed and edited by the Logic-User.

To ensure that our treatment of sandbox generation is robust to a wide range of models, we treat sandbox generation in general terms. We observe that the goal of sandboxing is to make controllers total while also safe. Because controller partiality arises from inexhaustive tests (such as testing whether a control decision lies within a safe range), we generate sandboxes by wrapping controllers containing Demonic tests with exhaustive `switch` statements. The `switch`es are exhaustive because they have fall-through cases with guard formula `true`. The fall-through case calls the fallback controller.

Our approach has several implementation details which must be decided: we must determine which tests are "controller" tests, and we must decide the contents of the fallback branch. When searching for "controller" tests, we ignore any block of assumptions at the very beginning of a model, which typically represent assumptions. We also ignore tests which follow an ODE, because such tests are typically timing lower-bounds that are relevant only to liveness rather than safety. Moreover, they are environmental assumptions on Demon in the spirit of plant assumptions, which should be kept. When prepopulating the fallback branch, it is unrealistic to provide a complete, correct fallback controller; as in classical VeriPhy, the user typically must specify the fallback action. As a heuristic, however, we consider the block between a controller test and any following ODE to be a controller, which is copied to the fallback case with a comment indicating that the author must fill in the fallback controller. Because the fallback controller should take no inputs from Demon, any Demonic assignments in the copied fallback case are uniformly changed to deterministic assignments to 0. Assertions from the controller block are also copied, and often will not prove once Demonic inputs are replaced with zeros, requiring manual repair effort afterward. However, prepopulating the fallback case is helpful because Kaisar controllers often include assertions which will be similar for the main and fallback cases in practice, with similar proofs. Prepopulating the fallback is intended to reduce manual effort, not eliminate it.

In contrast to classical VeriPhy, the correctness proof of the fallback controller is part of the Kaisar proof which serves as the input of constructive VeriPhy. This design decision resolves a limitation of classical VeriPhy where fallback correctness proofs can only be attempted with fixed, incomplete automation. As with any Kaisar script, correctness assertions can be written in the fallback model without explicit proofs, in which case an

incomplete automated proof procedure is applied. Thus, constructive VeriPhy does still attempt an automated proof by default, but provides greater flexibility when an automated proof attempt fails.

8.3.2 The ProofPlex Algorithm

ProofPlex is responsible for synthesizing (or extracting) an executable representation from a Kaisar proof. As in classical VeriPhy, the result is a discrete system which can then be interpreted against a Demonic driver or, in principle, compiled. Just as the Kaisar proofchecker (Chapter 7) contains several transformation passes, ProofPlex is divided into transformation passes. We describe each pass in turn. The final result is an intermediate representation (IR) of a discrete system. The IR format for Angel's strategy is described in Section 8.3.3.1, and its type in Scala is called `AngelStrategy`.

8.3.2.1 Closure

Recall that program variables in CdGL (and dL) are globally rather than lexically scoped, and variables are often read in CdGL models before they are assigned. In contrast, program execution expects all variables to be initialized before they are read. For that reason, ProofPlex begins by ensuring that the Kaisar proof is closed: the free programs variables x of the proof are computed and the proof is prefixed with a Demonic assignment $x := *$ for each free variable. At runtime, the Demonic driver will be asked to provide values for each assignment, which serve as the initial values of the variables.

8.3.2.2 Erasure

At this stage, we leave the language of Kaisar by erasing proof annotations, resulting in a simplified language of Angel strategies. For example, assertions are erased and function and predicate definitions are fully expanded. Angelic loops, choices, and ODEs are retained, but their unstructured proof steps are erased. At this stage, an error is reported if the model cannot be erased because its erasure contains tests that cannot be evaluated. In contrast to classical VeriPhy, we support any *weak-test* model, meaning one where tests contain no programs or quantifiers. The language of Angel strategies is described in Section 8.3.3.2, and is essentially an intermediate representation for the strategy fragment of Kaisar (Section 7.2.1).

8.3.2.3 Demonification

Next, we reify Angelic strategies to discrete systems in accordance with the CdGL reification rules (Section 6.4). This pass is called Demonification because the resulting model contains only connectives that are controlled by Demon, as Angel's strategy has been baked into the system. Case analysis statements become guarded Demonic choices and Angelic `for` loops become guarded Demonic repetitions (Section 6.5.2). Angelic and Demonic ODEs are both reduced to deterministic assignments or guarded nondeterministic assignments, so that the backend need only concern itself with discrete dynamic logic connectives. The

tradeoff is that the resulting system contains new Demonic tests, choices, and loops whose corresponding connectives in the source model were Angelic rather than Demonic. The output of Demonification is an intermediate representation of Angel's strategy which can be executed once we have implemented arithmetic.

It is important for usability to enable the Engineer to write Demon drivers at a level of abstraction close to the source model, so we provide a wrapper that lets the Engineer write Demon strategies as if the Demonification pass never occurred. The wrapper listens for Demonic connectives introduced by the Demonification pass and automatically plays them according to Angel's strategy, which the wrapper looks up using strategy metadata that is extracted and stored during the Demonification pass for later use at runtime. The wrapper only invokes the Demon driver when true Demonic constructs from the source model are encountered. For the sake of usability, the same wrapper also allows writing a Demon strategy that is agnostic to Kaisar's (Chapter 7) SSA pass. Recall that since the input Kaisar model of constructive VeriPhy is already in SSA form, the raw extracted strategy will have multiple SSA-variants x_i of a single source variable x. The wrapper lets the driver simply read or write x, automatically maintaining a separate value for x which agrees with the latest x_i, so that the Demon driver can be written agnostic of SSA.

8.3.3 Data Structures

We discuss data structures for representation of languages and runtime state.

8.3.3.1 Strategy Language

One of the most important data structures for the execution of strategies is the IR language for the strategies. We discuss different computational interpretations of games discussed throughout the thesis and their relation to different potential designs for strategy languages:

- In the realizability semantics of CGL (Section 4.6), the language of realizers is a programming language for executable strategies. The realizability semantics emphasize imperative changes to program state and a continuation-passing design, where the realizer for a given strategy statement returns another realizer which is used to play any following statements. The well-formedness constraints for realizers are like a simple type system in the sense that well-formedness does not check whether a realizer wins a certain game, only whether it can be used to play the game at all. The realizability semantics do not use an explicit syntactic representation for Demon's strategy, but quantify over all possible Demonic actions.
- In the type-theoretic CdGL semantics (Section 5.6), strategies are direct-style, pure functional programs with a dependent type system. In the dependently-typed system, a strategy only typechecks if it wins the given game with the given postcondition. Constructive strategies for box modalities must win a game regardless of what choice Demon makes, but Demon's strategy does not have an explicit syntactic representation. The big-step operational semantics of CdGL give an algorithm for playing two constructive strategies against each other, which could be used in the case where Demon follows a constructive strategy.

- The strategy fragment of Kaisar (Section 7.2.1) is a syntax for Angelic strategies. It expresses strategies as direct-style imperative programs. Kaisar allows a theorem statement to be inferred from a proof, which is equivalent to type inference for strategy programs. In this sense, Kaisar is more permissive than even the realizability semantics might suggest: a user can start writing a strategy without first deciding *which game* it is a strategy for.
- The proof term semantics of CGL (Chapter 4), unlike the other kinds of semantics defined in the thesis, are not directly concerned with computational interpretations of strategies. Rather, they show how proofs (correspondingly, strategies) can be simplified *statically*, before they are ever executed.

Realizers, dependently-typed terms, and Kaisar strategies are all plausible representations of strategies, but their strengths and weaknesses differ. Dependently-typed terms have strong typing constraints which imply strong correctness guarantees, but would require either using a programming language which supports dependent types (Scala does not) or implementing the dependently-typed realizer language from scratch. If we ever decide to execute incorrect strategies for testing or simulation purposes, dependent typing constraints could also make the execution of such strategies harder rather than easier. The realizability semantics do not require dependent types, but they do use features which we expect are unfamiliar to most potential users. Common proof paradigms such as loop invariant proofs result in infinite coinductive realizers with infinitely-nested anonymous functions in continuation-passing style, suggesting that realizers are ill-suited for use by non-experts. Kaisar strategies differ from realizers in that they are direct-style and provide built-in high-level constructs for looping rather than relying on infinite constructions, both of which make Kaisar strategies more promising for non-expert use. The realizer semantics are meant to be as abstract as possible, a design priority which is reflected in their complexity. Direct-style programming aside, Kaisar strategies are like a concrete syntactic model of the abstract realizability semantics. Given these tradeoffs, our executable IR for Angelic strategies is a variant of Kaisar strategies[3] which remembers only the information that is essential for execution. Because realizability semantics, type-theoretic semantics, and Kaisar strategies do not define an explicit representation for *Demon*'s strategies, this chapter will have to introduce its own representation, which we will do by defining a Scala trait `DemonStrategy`. However, we will present Angel's strategies first.

The Scala datatypes for Angel's strategies are given in Fig. 8.1. The strategy constructors are divided into two traits `AngelStrategy` and `SimpleStrategy`, where `SimpleStrategy` is a subtrait of `AngelStrategy`. The `AngelStrategy` trait includes all constructors for Angelic strategies, while `SimpleStrategy` includes only constructors which are allowed in Demonified strategies, i.e., those constructors where Angel makes no choices.

Constructor names starting with `S` represent the `SimpleStrategy` constructors for Demonic connectives, while names starting with `A` represent Angelic connectives. In Scala,

[3]While our IR representation comes from Kaisar, the CGL realizability semantics and CdGL big-step semantics were also sources of inspiration, specifically for the treatment of Demon and the interpretation algorithm, as we note in the discussions of Demon's strategies and of the interpreter.

```
sealed trait AngelStrategy { val nodeID: Int }
sealed trait SimpleStrategy extends AngelStrategy

case class STest(f: Formula) extends SimpleStrategy
case class SAssign(x: Ident, f: Term) extends SimpleStrategy
case class SAssignAny(x: Ident) extends SimpleStrategy
case class SLoop(s: AngelStrategy) extends SimpleStrategy
case class SCompose(children: List[AngelStrategy])
   extends SimpleStrategy
case class SChoice(l: AngelStrategy, r: AngelStrategy)
   extends SimpleStrategy
case class SODE(ode: ODESystem) extends SimpleStrategy

case class AForLoop(idx: Ident, idx0: Term, conv: Formula,
                    body: AngelStrategy, idxUp: Term,
                    guardDelta:Option[Term])
   extends AngelStrategy
case class ASwitch(branches: List[(Formula, AngelStrategy)])
   extends AngelStrategy
case class AODE(ode: ODESystem, dur: Term) extends AngelStrategy
```

Figure 8.1: Scala datatype for ProofPlex strategy IR.

the syntax `case class ... extends T` represents the declaration of a constructor for datatype `T`. Each node of a strategy contains a (unique, auto-generated) numeric identifier so that error messages can recover the strategy text from an ID.

Constructor `STest(fml)` is a weak-test Demonic test where `fml` is the guard condition. Deterministic assignments `SAssign(x, f)` set x to the value of f and are simultaneously Angelic and Demonic. Nondeterministic Demonic assignments `SAssignAny(x)` assign x a value supplied by Demon. Demonic loops `SLoop(s)` repeatedly ask Demon whether to terminate the loop, and run strategy s at each iteration. Demon is responsible for eventually terminating the loop. [4] Composition `SCompose(children)` is considered both Demonic and Angelic, and each child is executed in sequence. Demonic choice `SChoice(l, r)` asks Demon whether strategy `l` vs. `r` should be played.

Note that the `SimpleStrategy` constructors accept `AngelStrategy` arguments because the `SimpleStrategy` constructors are also useful in an `AngelStrategy`. When a value has type `SimpleStrategy`, it does not mean the entire strategy consists entirely of simple constructors, it means the top-level constructor is simple.

[4]Termination need not be based on an explicit termination metric. For example, it is common for Demons to terminate in response to user input. While Demon should avoid true infinite loops, termination is guaranteed in any case by the second law of thermodynamics.

Angelic `for` loops are represented by

```
AForLoop(idx: Ident, idx0: Term, conv: Formula,
         body: AngelStrategy, idxUp: Term,
         guardDelta: Option[Term])
```

where `idx` is the index variable of the loop, whose initial value is computed by `idx0` and which is updated to the value of `idxUp` at the end of each iteration. The `body` is executed repeatedly so long as `conv` holds. Kaisar is responsible for ensuring Angelic loops terminate by checking that the guard `conv` is eventually false when repeatedly applying the update `idx := idxUp` after each iteration. The `guardDelta` contains the guard comparison delta from the Kaisar proof, if any. If `guardDelta` is empty, simple interval comparisons are used for guards, else the interval width is additionally inflated by the delta. Constructive VeriPhy is only responsible for extracting an executable strategy which satisfies the same termination property. Constructor `ASwitch(branches)` represents a switch statement strategy for an Angelic choice. Each branch's guard condition is tested until the first true one is found, then the corresponding branch is executed. Kaisar is responsible for ensuring that all `switch` proofs have constructively-total guards, while constructive VeriPhy is only responsible for ensuring the code extracted from the proof also has total guards. Angelic ODE's are represented by `AODE(ode, dur)` which expresses that `ode` should evolve for the duration denoted by term `dur`.

While Angel's (syntactic) strategy can contain both Angelic and Demonic connectives, Demon's own strategy is less well-suited for explicit syntactic representations. Unlike Angel, Demon is generally a real or simulated CPS with its own external sensing, actuation, and/or control, so the representation of Demon's strategies must allow for external components. Instead of defining Demon's strategies positively with an explicit syntax, we define them negatively by giving an interface which describes their capabilities. By characterizing Demon's strategies abstractly as an interface, we allow the arbitrary implementation flexibility which is required when interacting with external systems. The negative characterization of Demonic strategies as an interface is comparable to the abstract definition of realizers (Section 4.6), where Demonic constructs are given negative definitions. The Scala trait `DemonStrategy[T]` gives the interface for Demon's strategies whose numeric type is `T`. The trait `DemonStrategy[T]` is given in Fig. 8.2.

Each method is told the `NodeID` of the statement being executed, so that a Demon driver can call back into constructive VeriPhy to receive additional information about a statement. We choose numeric `NodeID`s so that our interface uses the simplest types possible, which would prove helpful for any future work developing a verified, low-level execution backend. The `init` method is called upon the start of execution so that a driver can perform any system-specific initialization. Methods `readLoop`, `readChoice`, and `readAssign` respectively ask the driver whether to continue a loop, which branch to take in a choice, or what value to assign to a variable. The `read` family of methods model the negative, Demonic realizers: when a nondeterministic construct is executed during Demon's turn, we ask Demon to resolve nondeterminism. The `writeAssign` method is called whenever a *deterministic* assignment, including any Angelic assignment,

```
trait DemonStrategy[T] {
  type NodeID = Int
  def init(): Unit = ()
  def readLoop(id: NodeID): Boolean
  def readChoice(id: NodeID): Boolean
  def readAssign(id: NodeID, x: Ident): T
  def writeAssign(id: NodeID, x: Ident, f: T): Unit
}
```

Figure 8.2: Scala trait for Demonic strategies.

is executed. The `writeAssign` method asks nothing of Demon, but serves as a hook for any external action Demon wishes to take based on the assignment. Typically, Demon uses `writeAssign` to learn when an actuated variable (such as acceleration) has been assigned and to perform the actuation. Recall that constructive VeriPhy reduces ODEs to assignments, nondeterministic assignments, and tests. Thus, Demon uses `readAssign` to return the final state of a Demonic ODE or `writeAssign` to learn the final state of an Angelic ODE that is represented by assignments which implement its solution. Because real CPSs are written in a wide variety of languages, the user-supplied implementation of `DemonStrategy` can and often will be a thin wrapper which offloads sensing, actuation, and optional control to an implementation in another language.

We are now equipped to revisit a remark from Section 7.2.2 in Chapter 7, where we hypothesized that there is likely little difficulty in switching among code generated for different strategies of the same game. The Engineer's driver code interacts with VeriPhy through the `DemonStrategy` trait, so when we assert that switching between two strategies is easy, we mean that both strategies would interact with `DemonStrategy` in similar ways. Two strategies which play the same game will typically read and write the same variables in the same places and will typically feature similar branching and loop structure in similar places, thus have similar interactions with `DemonStrategy`. We have not attempted to show that this is always the case, particularly because the refinement checker of Section 7.2.2 does permit certain differences between strategies and games, such as certain differences in branching structure and the addition of ghost assignments. While it is possible in principle that such differences would preclude full, automatic interchangeability between different strategies of the same game, we nonetheless find it conceptually helpful to think of games as reusable interfaces for strategies and expect a nontrivial degree of interchangeability in practice.

8.3.3.2 States and Numeric Variables

Strategy execution maintains a runtime `state` which maps every variable `Identifier` to its value, which is always a `number`. Each `number` is a rational interval. Whenever Angel or Demon assigns a variable, the new value of the variable is reflected in the state.

```
type number = (Rational, Rational)
type state = Map[Ident, number]
```

Constructive VeriPhy uses[5] intervals whose endpoints are rational. The rational number type we use is provided by the Scala library `spire` (Osheim & Switzer, 2011–)[6]. The `spire` library was chosen because it provides a variety of numeric types in a common hierarchy. All of the `spire` types discussed here provide more flexibility than the fixed-precision arithmetic of classical VeriPhy, as desired. However, each type has its advantages and disadvantages:

- Rational numbers were chosen because they are relatively simple and specifically because they are available in HOL4. Their availability in HOL4 means that a rational-based execution backend can in principle be verified more easily in future work. The downside of rational numbers is that (square and nth-) roots cannot be evaluated exactly, requiring rounding. The resulting rounding means that either rational intervals must be used or some unsoundness resulting from rounding must be accepted. To ensure soundness, we use interval arithmetic where we round down when computing the lower endpoint and round up when computing the upper endpoint of the interval.
- The `BigDecimal` type in `spire` has the same advantages and disadvantages that it does in vanilla Scala. Like a `Rational`, a `BigDecimal` can increase its precision as needed, but `BigDecimal` faces inexactness challenges not only for roots (as does `Rational`) but division as well. Computations on `BigDecimal` are only used internally to compute the approximate root of a `Rational`, as the `Rational` type does not have its own root function. The `BigDecimal` root function allows the caller to specify a rounding mode, so we round down for lower bounds and round up for upper bounds.
- Computable real numbers would provide a close connection between the constructive VeriPhy implementation and the CdGL foundations. In principle, computable real numbers would simplify transfer of totality guarantees for constructive case analyses because inexact comparisons in Kaisar are already total over constructive reals. However, totality is ensured by sampling inputs to arbitrarily high precision, which is not realistic for inputs drawn from sensors whose precision is finite.

  A first downside of computable real numbers is that their performance is often worse than other common numeric types. A second downside is that computable real numbers are typically represented as functions. On the one hand, our present implementation could easily be interfaced with computable real numbers because Scala has first-class functions. However, if we ever develop a back-end which compiles strategies to a low-level programming language, we might find that function-based data structures are difficult to represent in low-level languages. Likewise, to write a Demon

[5]Our implementation defines a trait `Numeric` so that different numeric types can be swapped out. While we have implemented the trait for both single rational numbers and rational intervals, we recommend use of rational intervals specifically.

[6]To reduce the size of our executable files, we do not link in the entire library. Instead, we manually copied just the rational number type. The `spire` library is released under the MIT license, which permits such derivative works.

driver which interfaces with low-level code, we would need to convert computable real numbers to the low-level code's (most likely inexact) number format.

- Real-algebraic numbers occupy a middle-ground: exact roots are supported and an explicit, non-functional representation is available. However, real-algebraic numbers were not chosen because they may still pose similar integration challenges to computable reals and because the associated performance penalties are also unclear.

We briefly reflect on the relationship between our present use of rational interval arithmetic in constructive VeriPhy, the use of *integer* interval arithmetic in classical VeriPhy, and the use of rationals and reals in CGL and CdGL, respectively. In doing so we also reflect on the role of exact and inexact arithmetic comparisons. As did classical VeriPhy, we conservatively approximate the term semantics of the source language: while computable reals *could* give us an exact match for CdGL arithmetic, rationals were chosen for their simplicity. If we were interested in synthesis of CGL rather than CdGL, a *single* rational number would suffice to exactly model arithmetic. Instead, an *interval* gives a sound, conservative (incomplete) approximation of a computable real. Computations over computable reals are like interval arithmetic computations where the consumer of a function requests the precision of the output interval and the function samples all inputs at high enough precision to achieve the requested output precision. Comparison operations likewise provide comparison deltas which are used to query terms at sufficient precision to make the comparison soundly. In contrast, we do not allow the consumer to dynamically request a precision, so we do not affix comparison deltas to comparison operations, rather we compare intervals. While comparison deltas, which were the focus of significant attention in CdGL (Chapter 5) and Kaisar (Chapter 7), are erased in constructive VeriPhy, it is still useful in practice to know that all comparisons in the source model were satisfied by a nonzero comparison delta. In practice, if the terms compared in a comparison are found to overlap (thus causing the comparison result to be unknown), we would respond by increasing the precision of all inexact operations or increasing comparison precision by decreasing comparison deltas. The fact that the source model admitted inexact comparison implies that it is typically possible in practice to eliminate overlapping comparisons in constructive VeriPhy by sufficiently increasing precision.

The `state` type is simply a `Map`, so we wrap it in a class `Environment` in order to provide smart accessors which are useful in the SSA wrapper (Section 8.3.2.3). The `Environment` class also implements the evaluation of terms and formulas in methods `Environment.eval` and `Environment.holds`, respectively. Method `eval` accepts a term as its argument and returns a number (i.e., interval) representing the value(s) of the term, while `holds` accepts a formula and returns a truth value. Because interval arithmetic is conservative, `holds` returns a ternary truth value, just like the interval semantics for formulas used in classical VeriPhy (Section 3.4). We do not write the definitions of `holds` and `eval` here because those definitions are almost identical to the interval semantics from Section 3.4. The only differences are that `holds` and `eval` use rationals rather than integers and that our representation does not have (does not need) explicit infinities, so `holds` and `eval` do not need to implement edge cases for infinities. As in classical VeriPhy, tests only pass when the test condition is definitely-true, rather than

unknown. The evaluation of formulas uses the fact that strategies are weak-test, meaning that quantifiers and modalities never need to be evaluated in `holds`.

8.3.4 Execution

Now that strategies and states have been defined, we present the execution algorithm. The big-step operational semantics play of CdGL Section 5.6 provides a good analogy for our execution algorithm, despite the fact that our data structures are not dependently-typed and the fact that the play semantics do not use a different strategy data structure for Demon than for Angel. Execution proceeds by recursion on the structure of Angel's strategy, just as play follows the structure of a game. When Angel's strategy requires Demonic input, the Demon strategy is consulted, and when Angel performs an assignment, Demon is notified.

Strategy execution is provided by a function `Play.apply()`[7], which is presented in Fig. 8.3. If execution terminates normally, `Play.apply` returns `Unit` and the results of execution are accessed through the `Environment`. The `Environment` is an abstract interface for the `state`, which typically starts out as the empty state. An exception is raised if a test condition fails or if an unsupported strategy connective or formula is encountered. In practice, unsupported connectives should never occur and unsupported formulas should only occur in the rare case that a verified model features division by zero or negative roots. It is the responsibility of constructive VeriPhy to check for unsupported tests and eliminate Angelic strategy connectives before calling `Play.apply`. Failed tests, in contrast, are often expected: many Angel strategies win their games by backing Demon into a corner where Demon is forced to fail a test, making Angel win.

Because not all exceptions are equal causes for concern, we distinguish exceptions for tests (`TestFailure`), unexpected Angelic strategies (`UnsupportedStrategy`), and expressions that cannot be evaluated (`NoValue`). All exceptions contain the `nodeID` of the statement which caused the exception, so that the corresponding statement can be reported in error messages.

```
case class TestFailure(nodeID: Int) extends Exception
case class UnsupportedStrategy(nodeID: Int) extends Exception
case class NoValue(nodeID: Int) extends Exception
```

Having introduced the exception hierarchy, we are ready to present the execution algorithm implementation (Fig. 8.3).

A Demonic test (`STest(f)`) is executed by testing if `f` holds, raising `TestFailure` if `f` cannot be shown. A deterministic assignment (`SAssign(x, f)`) is executed by evaluating `f`, announcing the result to Demon, and storing the result in `x`. An exception is raised in error cases such as division by zero or rooting a negative number. Demonic nondeterministic assignments (`SAssignAny(x)`) ask Demon for the value, which is stored in the environment. Demonic loops (`SLoop(s)`) repeat the loop body until `Demon` says to stop. Each recursive call to the loop body will update the environment in-place. Sequential

[7]In Scala, implementing a method named `obj.apply` allows `obj` to be invoked with functional syntax `obj(args)`. In this case, `object Play` is best understood as a globally-visible function.

```
object Play { ...
  def apply(env: Environment, as: AngelStrategy,
            ds: DemonStrategy[number]): Unit = {
    as match {
      case STest(f) =>
        try { if (!env.holds(f)) throw TestFailure(as.nodeID) }
        catch { case v: NoValue => throw NoValue(as.nodeID) }
      case SAssign(x, f) =>
        try {
          val v = env.eval(f)
          ds.writeAssign(as.nodeID, x, v)
          env.set(x, v)
        } catch { case v: NoValue => throw NoValue(as.nodeID)}
      case SAssignAny(x) =>
        val v = ds.readAssign(as.nodeID, x)
        env.set(x, v)
      case SLoop(s) =>
        while(ds.readLoop(as.nodeID)) { apply(env, s, ds)}
      case SCompose(children) =>
        children.foreach(x => apply(env, x, ds))
      case SChoice(l, r) =>
        if (ds.readChoice(as.nodeID)) apply(env, l, ds)
        else apply(env, r, ds)
      case _ => throw UnsupportedStrategy(as.nodeID) }}}
```

Figure 8.3: Execution algorithm for strategies.

compositions `SCompose(children)` execute each child, updating the environment in-place. Demonic choice (`SChoice(l, r)`) asks Demon which branch to take.

8.4 Synthesis Considerations for Modeling

While code can be synthesized from any (weak-test) proof, not all synthesized code is equally useful. Thus, the development process often involves revising a model and proof in order to improve the quality of synthesized code. One qualitative goal of constructive VeriPhy is to ensure that the process of developing and revising models for synthesis is no more difficult than necessary. We give a qualitative comparison of our experiences with model revision and code quality in classical VeriPhy vs. constructive VeriPhy as well as proof recommendations that maximize code quality. We give this comparison in the context of driving models, but it applies to any model.

Recall that a single game can have multiple winning strategies, each of which corresponds to Angelic control code and Demonic monitoring code. While VeriPhy always generates code which is faithful to the given strategy, some strategies are preferable to others. A good Angelic strategy always makes progress toward its goal. By one metric, we may want Angel to be as aggressive as possible, though other metrics such as (fuel) efficiency are also of interest. A good Demonic strategy, on the other hand, is as permissive as possible without violating safety. More permissive Demonic strategies correspond to more permissive monitors which exhibit fewer spurious alarms at runtime.

The most systematic way to ensure a controller makes progress is to generate the controller from a reach-avoid proof rather than a simple safety proof. Reach-avoid proofs contain proofs of progress, a property which transfers automatically to the strategy specified by the proof. While the use of interval arithmetic *can* still cause progress violations, the use of arbitrary-precision interval arithmetic allows the frequency of those violations to be minimized. The author of a model can often find a successful controller without any special analysis, especially for simple applications. For example, when accelerating at $acc := A$ for $A > 0$, it is intuitive that accelerating toward the destination yields progress.

While the design of good Demonic proofs is less systematic, bad Demonic proofs can often be found systematically. Solution-based ODE proofs typically result in poor monitors, because those monitors will fail if physics does not obey the exact solution of the ODE. Because no model nor sensor is perfect, real CPSs will not obey ODE solutions exactly, nor will sophisticated simulations. For this reason, ODEs should be proved with invariants when synthesis is desired. The invariants should be inequalities, because equational invariants are just as restrictive as solution equations.

Inequational ODE invariants can be found in several ways:

- Many models use controllers which are less aggressive than the theoretical optimum, because less aggressive controllers sometimes have simpler proofs. In this case, inequational invariants often arise naturally. For example, the driving models in (Mitsch et al., 2017) conservatively approximate the distance to an obstacle with an ∞-norm, naturally leading to an invariant saying the true distance traversed is at most the distance predicted for straight-line travel.

- When inequational invariants do not arise naturally, the Logic-User can revise the model to include tolerances. The models in (Mitsch et al., 2017) include tolerances such as bounded actuator disturbance and sensing error, while the 2D waypoint-following model of Chapter 3 allows the vehicle to track its desired path within a distance tolerance. The original equational invariants can be made inequational in the resulting model by rephrasing the invariants in terms of the tolerances.

While building and verifying a model with tolerances is nontrivial, it is worthwhile even before considering its implications for synthesis. If modeling and verification are to be relevant to real-world safety, models must reflect real systems. VeriPhy's aversion to equational invariants is not a quirk of our synthesis approach, but a reflection of the fact that equational assumptions over sensed variables are not realistic. The fact that equational proofs generate failing monitors reflects a strength, not a weakness of monitor synthesis: because a synthesized monitor tells us when reality and models disagree, it provides a systematic way to detect when our modeling assumptions are too strict for reality.

While both classical VeriPhy and constructive VeriPhy leave the Logic-User responsible for writing a synthesizable proof, constructive VeriPhy makes the process easier in several ways. The ready availability of reach-avoid proofs allows the Logic-User to more easily assess the quality of controllers early, before synthesis. Moreover, constructive VeriPhy allows arbitrary model shapes rather than fixed templates, meaning the Logic-User is less likely to run into arbitrary format restrictions while revising a model.

While the Logic-User's choice of Angel and Demon proofs have an important impact on modeling, the impact of arithmetic representations is also significant. In particular, constructive VeriPhy's choice of rational arithmetic yields one of its clearest improvements in usability over classical VeriPhy. Because classical VeriPhy employs 32-bit integers with no rounding or shifting, it is exceedingly easy to encounter overflow errors. This is the case even when compared to other fixed-point arithmetic systems that allow for nonzero fractional parts: proper fixed-point multiplication downshifts according to the number of fractional bits, while integer multiplication does not. Our experience with 2D driving in classical VeriPhy was that units of measurement must be carefully experimentally determined even for models where terms are no more than quadratic. If the precisions used in Section 3.7.3 are significantly increased, overflows occur, while precision cannot be significantly decreased without rendering the monitors unusable. Thus, 32-bit integer arithmetic posed a significant limitation even on models of moderate complexity and could make classical VeriPhy unusable in the literal sense on models with higher-degree polynomials.

Constructive VeriPhy does actually establish a precision limit at compile-time, in preparation for future integration with low-level verified backends. However, the precision limits in constructive VeriPhy are qualitatively different: the limit can be arbitrarily high and the default precision was significantly higher than needed in our models. A more subtle but important point is that because rational arithmetic allows fractions, the author does not have to manually choose individual precisions for each variable at design time, which can simplify the model by reducing or eliminating the need for manual unit conversions. If the result of an arithmetic operation ever exceeds the chosen precision and causes an error, the solution is far simpler in constructive VeriPhy: the global size limit can simply be raised,

rather than experimentally tweaking variable precision in order to pack all multiplication results into 32 bits.

8.5 Results Comparison

Constructive VeriPhy's goals are both subjective and objective. In Section 8.4, we subjectively discussed the qualitative improvements in development process provided by constructive VeriPhy. In this section, we give an empirical evaluation which compares the runtime behavior of sandboxes synthesized by both classical VeriPhy and constructive VeriPhy.

We reproduce the evaluations of Chapter 3 on constructive VeriPhy and compare the results across versions. We reproduce both the Raspberry Pi-based hardware evaluation (Section 8.5.1) and the AirSim-based simulation (Section 8.5.2). Kaisar proofs for each system are fed as inputs to constructive VeriPhy in order to synthesize the code used in each evaluation. The source of each Kaisar proof is Appendix D.1. See Section 7.5 in Chapter 7 for discussion of the Kaisar proofs from Appendix D.1; in this chapter, the Kaisar models and proofs are discussed only briefly to help understand the synthesis experiments.

The foremost goal of the present evaluation is to demonstrate the key functionality of constructive VeriPhy by applying it to the same case studies as in Chapter 3 and additionally (in the GoPiGo example) expanding the case study to include Angelic constructs with liveness proofs from which concrete control code is extracted. Any improvement according to the evaluation metrics used in Chapter 3, such as monitor failure rates, is a bonus rather than our primary goal.

8.5.1 GoPiGo Results

The GoPiGo evaluation uses several variants of the Kaisar port of the 1D driving model of Chapter 3, whose listings are in Appendix D.1 and whose Kaisar proofs are summarized in Table 7.1 of Chapter 7:

- The *Demonic Choice* model is a direct port of the Chapter 3 model. That is, the controller model is a Demonic choice where the first branch allows Demon to choose the velocity v (which is identical to linear speed in our setting of forward 1D motion) and the second branch fully, immediately brakes by choosing $v = 0$. In this model and those that follow, the plant models relative, linear motion at speed v, where the speed v does not change continuously, only discretely. The duration of the plant in each system loop iteration is controlled by Demon but is bounded above by a constant. Demon controls the duration of the system loop.
- In the *Angelic Sandbox* model, Demon suggests a velocity v first, then Angel uses a `switch` statement to check whether the suggested velocity v is safe. That is, she checks a control monitor condition which, when true, implies the vehicle cannot crash during the current loop iteration. If the suggested velocity is not safe, Angel fully, immediately brakes by choosing $v = 0$. Demon controls the loop duration.
- The *Timed Angelic Control* uses a `switch` statement with no input from Demon: Angel decides whether to drive at maximum speed ($v = V$ where V is a system

parametr for maximum speed) or brake ($v = 0$). The system loop is an Angelic `for` loop which runs for number of iterations for a total time of up to 10 seconds, which is the same duration limit we used in experiments where Demon controls the loop.

- The *Reach-Avoid* model differs from the Timed Angelic Control model in that the `for` loop is guarded by distance to the obstacle: the first time that Angel decides to brake for safety, the loop stops. Both safety and liveness bounds are proved.
- The *Reach-Avoid with Disturbance* model was not used in this evaluation because it is sufficiently similar to the Reach-Avoid model that similar insights are provided by experiments using either of the two.

The reason we used a series of related models was so that we can both reproduce the functionality of classical VeriPhy and test the new functionality provided in this chapter. As in Section 7.5, a series of increasingly complicated models also allows us to assess the ease or difficulty of incremental changes, which often occur in practice.

The Demonic Choice model allows us to assess how well constructive VeriPhy directly reproduces the functionality of classical VeriPhy. Whereas classical VeriPhy always converts every controller to a sandbox, constructive VeriPhy expects the user to explicitly ask for a sandbox if they want it, and also allows non-sandbox models to be executed. The Demonic Choice model thus assumes that Demon makes safe choices and reports whether Demon makes an unsafe choice, but does not terminate execution when Demon makes an unsafe choice. In practice, we may wish to terminate execution or invoke a fallback upon an unsafe decision, but in the context of an evaluation, recording the unsafe decisions provided useful experimental data. The Angelic Choice model explicitly models an Angelic sandbox, thus implementing the same sandbox logic expected in classical VeriPhy. The Timed Angelic Control model demonstrates a concrete Angelic controller including concrete Angelic control of loop duration. It uses a fixed duration in order to more closely match the behavior of preceding models. The Reach-Avoid model stops when the robot reaches the obstacle, proving a lower bound on the progress made by each loop iteration. Because reach-avoid proofs are a key proof paradigm in CdGL and Kaisar, we wrote this model to show that reach-avoid proofs are supported in constructive VeriPhy too.

We present the data from the experiments first, then discuss the lessons learned from the experiments in Section 8.5.3. We use the same symbols from Chapter 3: **C†** for spikes in the controller, **C↯** for controller monitor failures, **P↯** for plant monitor failures, and **Ob0** for the obstacle stopping. Each of the models was evaluated using the same controller configurations discussed in Chapter 3:

- A *correct controller* was employed which detects when it is too close to the *static obstacle* and brakes for safety. This case is plotted in blue with circle markers.
- A *buggy controller* was employed which ignores the static obstacle. It initially follows the requested speed, but commands maximum speed once the distance to the obstacle is less than 50cm. The sandbox logic must override the commanded maximum speed for safety. This case is plotted in orange-red with $\times$ markers.
- The correct controller is used when the obstacle is *approaching*, which can violate the plant model's safety assumptions. This case is plotted in yellow with + markers.

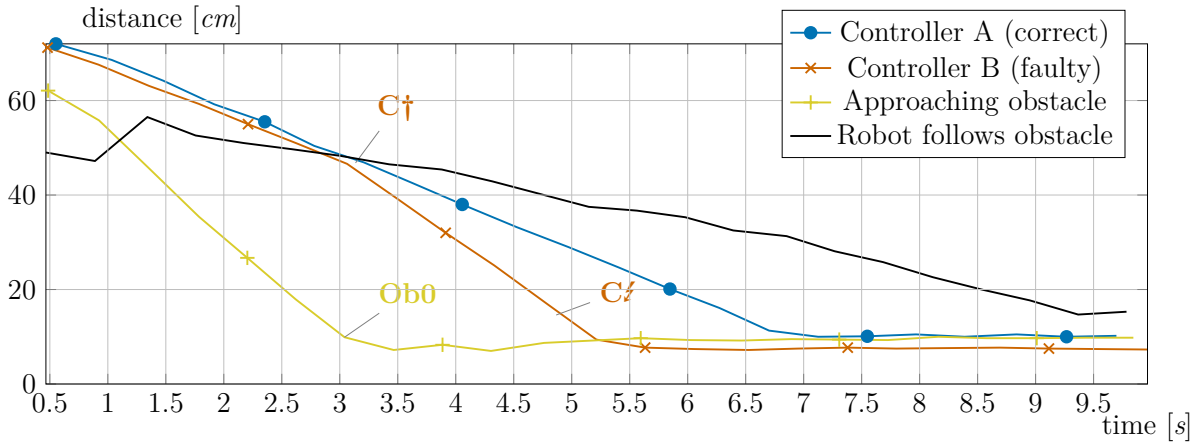


Figure 8.4: Robot experiments with Demonic model.

- The correct controller is applied when the obstacle is *receding*, which makes the model strictly safer but can violate progress assumptions in models that prove liveness. This case is plotted in black with no markers.

Because the approaching and receding obstacles were implemented by the experimenter manually moving an object in front of the robot, we expected (and observed) significant variance in the distance plots for the approaching and receding cases. For the same reason, the plots shown here for the approaching and receding obstacles differ drastically from their counterparts in Chapter 3 where the experimenter moved the obstacle with at a different speed and to a different extent.

In each case, the initial distance between the robot and obstacle is 75cm. The experiments were performed on a floor made of hard tile. We tested the robot with speeds of 10 cm/s, 15 cm/s, 20 cm/s, and 25 cm/s. We present graphs with results from the 10 cm/s test cases. We chose to present the 10 cm/s case in our graphs for the simple reason that, though the results for each speed would tell the same story, the graphs are easiest to read for slower-moving tests, where each graph contains more data points because the robot takes longer to reach its destination. Each graph shows the distance of the robot to the obstacle over time, with important system events annotated.

In Fig. 8.4, we present the results of the Demonic model controlled at a target speed of 10 cm/s. As in Chapter 3, we see that the robot moves forward steadily until it nears the obstacle. The robot stops at a distance of approximately 10 cm, which is about twice the stopping distance from Chapter 3. The greater stopping distance is a result of the greater time between control cycles: we allow up to 0.6 seconds between control cycles due to the long sensing times we witnessed[8]. When the controller knows there may be a longer interval between control decisions, it starts braking earlier, which is an essential aspect of ensuring safety.

[8]We are unsure why we witnessed higher sensing times than those reported in prior work. Sensing time is highly dependent on the time required to start a Python process on the GoPiGo, which could easily vary between software versions, though we have no direct evidence to that effect.

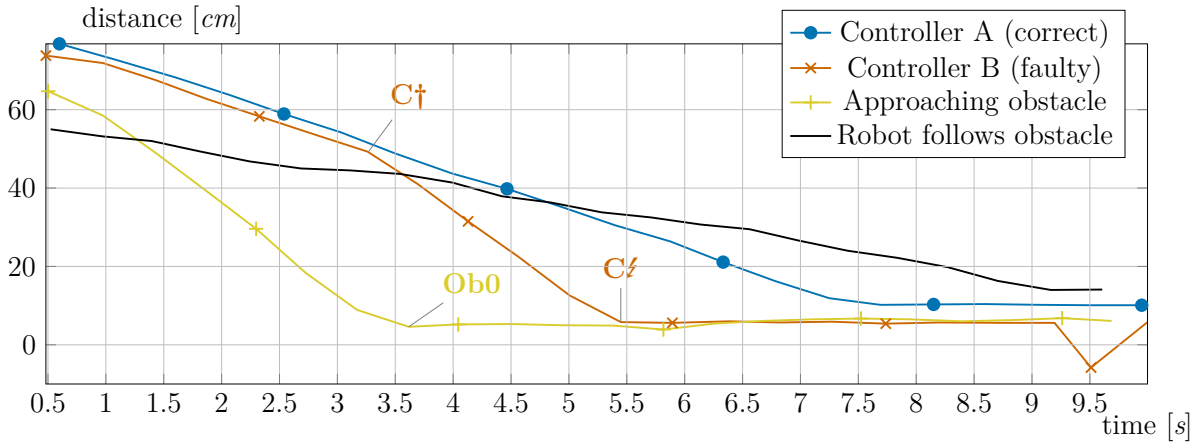


Figure 8.5: Robot experiments with Angelic Sandbox model.

In the orange-red plot, **C†** indicates the point at which the buggy controller begins commanding maximum speed, after which a steeper slope is observed, while **C↯** indicates the point at which the sandbox commands braking. Note the point **C↯** occurs a timestep before a change in velocity is observed. Because a similar delay of one timestep was not observed in other tests, we suspect but cannot prove that the delay was the result of actuation error. In the yellow plot, **Ob0** indicates the point at which the obstacle stopped moving. The experimenter stopped moving the obstacle when the robot stopped moving, because the distance sensor is known to exhibit bugs[9] when touched directly or when distance is extremely low. In the black plot, a brief initial increase in distance is observed as the experimenter varied the speed of the moving obstacle. The slow downward slope in the black plot represents that the robot was driving at the commanded speed while the obstacle moved away at a *slower* speed, resulting in a low but positive relative velocity.

In the Demonic sandbox test, we manually instrumented the Demon driver to perform VeriPhy-style sandboxing logic because in constructive VeriPhy, Angelic constructs rather than Demonic ones are used to model sandboxing explicitly. This instrumentation would not be used in practice, but was used for experimental purposes specifically to enable a close comparison of the two VeriPhy implementations.

The Angelic Sandbox model results are depicted in Fig. 8.5. The sandboxing behavior is the same as in the Demonic sandbox, though there are several incidental differences between the plots. The orange-red line briefly goes negative due to blip in the sensor data. The blip does not represent actual movement.

The Timed Angelic Control case is depicted in Fig. 8.6. The blue and black plots are similar to previous tests. The orange-red plot does not have have a control spike as a result of an implementation trick which was used in this test. Recall that the Timed Angelic Control model uses deterministic assignments for velocity rather than a Demonic nondeterministic assignment. We could have communicated the driving velocity by introducing an

[9]The sensor will report a negative distance and may or may not require power cycling before reporting correct distances again.

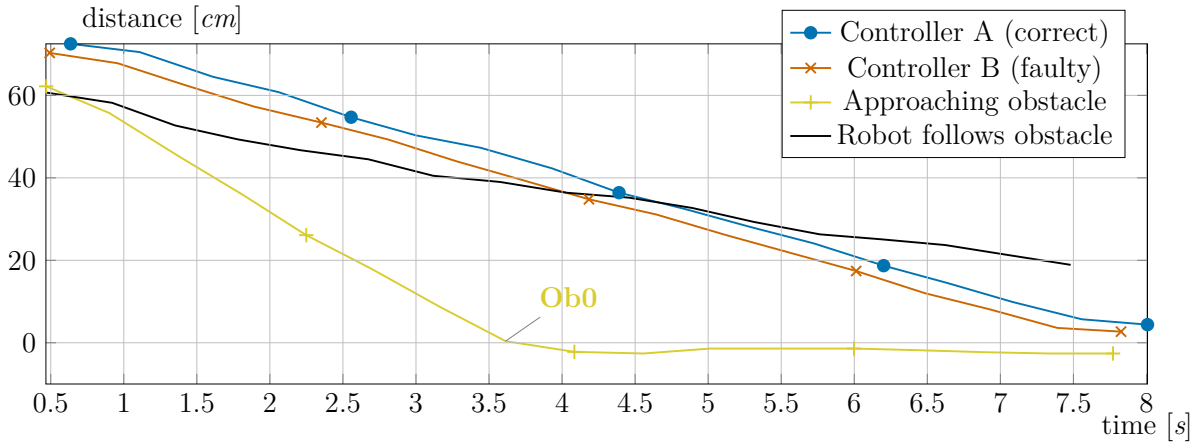


Figure 8.6: Robot experiments with Timed Angelic Control model.

additional variable, but for the sake of convenience we opted to set the *maximum velocity* to the driving velocity during initialization, so that the controller only ever drives at the so-called maximum rate or brakes. Despite the difference when compared to prior graphs, the test case serves its same purpose of ensuring that the sandbox logic safely overrides the buggy control decision. The yellow line demonstrates buggy behavior by the GoPiGo distance sensor: when the distance is very low, negative distances are reported. In this case, the sensor recovered after the obstacle was removed, but that is not always the case. Sensor bugs aside, an important takeaway from the yellow plot is that safety cannot always be ensured when plant assumptions are violated: a moving obstacle can crash into the vehicle even when the vehicle is stationary.

One might wonder why the durations of the lines vary and why none of them are 10 seconds long. The durations of the lines reveal the subtle meaning of `for` loops in Kaisar: our system initializes a variable called `time` to 0, then increments it by the maximum control cycle duration until it reaches 10 seconds. Because the maximum duration is a constant, the loop executes a fixed number of times. Every repetition of the loop takes less than the maximum time budget, causing the final duration to be less than 10 seconds. Natural variation in loop body duration results in variation in the duration of each test case. One can also implement a loop whose termination is based on actual elapsed time rather than the maximum time budget. The argument for such a loop would be similar to the following Reach-Avoid test case because it would rely on a *lower* bound on duration that ensures progress in each loop iteration.

The Reach-Avoid results are depicted in Fig. 8.7. A visual difference between the Reach-Avoid plot and previous plots is noticeable: all plots end as soon as the goal is nearly reached, sometimes within 4 seconds. The early stops are a result of the fact that Angel terminates the system loop as soon as brakes are engaged. The end of the yellow line is particularly abrupt because the obstacle moved quickly toward it, causing the goal to be reached. All of the indicated plant faults ($\mathbf{P}\lightning$) are expected because the plant monitor includes a lower bound on distance traveled which was used to prove liveness. The black line records the case where the experimenter moved the obstacle in a uneven fashion; the

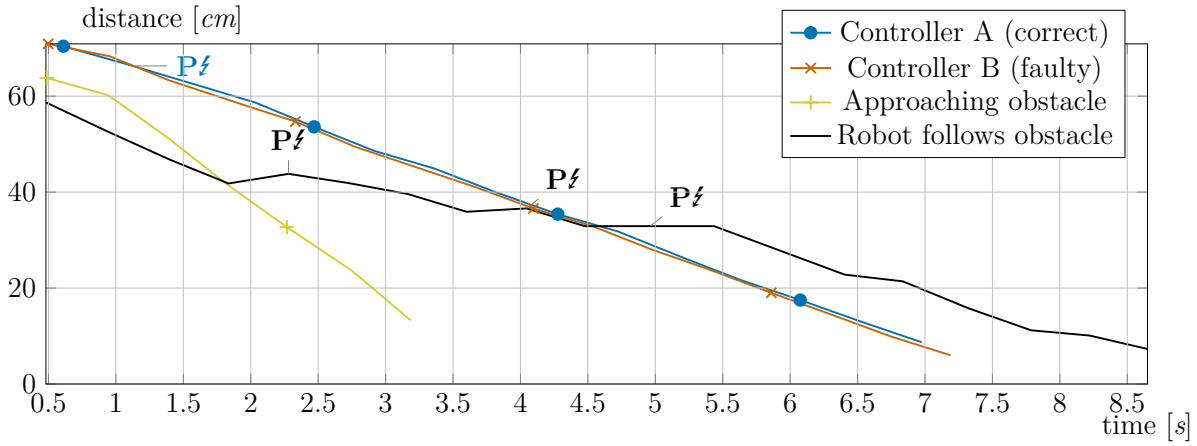


Figure 8.7: Robot experiments with Reach-Avoid model.

plant faults in the black line are due to the fact that the obstacle is moving away, which invalidates our modeling assumption of a static obstacle, thus invalidating the liveness part of the reach-avoid proof. In the blue line, the sole plant fault occurs because the robot is slow to accelerate and does not move far in its first timestep. In the model, the lower bound on true distance traveled is a fraction of the distance predicted by the ideal dynamics and the coefficient on the bound is configurable. In these tests, we used a coefficient of $\frac{1}{4}$, i.e., the monitor fails if the true distance is less that $\frac{1}{4}$ of the ideal distance.

The orange-red line does not have a control spike because we used the same trick described in the Timed Angelic Control graph: a single velocity is used throughout the test. We do not show a control fault or the point where the obstacle stops moving because those points are simply the endpoints of the respective plot lines.

The reach-avoid results suggest a general tradeoff in the use of Demonic vs. Angelic loops: while an Angelic loop can prove progress toward its goal, we may not always wish to terminate the controller as soon as the goal is reached. In the receding obstacle case (black line), it is possible that the obstacle recedes after the robot has braked, presenting an opportunity to resume driving. In this test case, the experimenter did continue to move the obstacle, but the controller terminated the first time it neared the obstacle and thus did not follow it again. If one wishes to have progress guarantees but also wishes to allow the controller to resume when an obstacle moves, a simple solution is to use a nested loop: an outer loop can terminate the system based on duration or user input, while the inner loop can make progress until a goal is reached.

In addition to evaluating our 1D models on the GoPiGo, we evaluated them using the same custom 1D simulation used in the Chapter 3 evaluation. The simulated evaluation process was primarily used with the Demonic sandbox model to debug basic functionality before performing an evaluation on the GoPiGo. We do not plot the results here because the simulation reproduced prior results exactly.

Table 8.1: Average speed, monitor failure rates, plant violation rates, for AirSim and human driver in Rectangle, Turns, and Clover for patrol missions.

	Avg. Speed (m/s)				
World	BB	PD1	PD2	PD3	Human
Rect	7.29	6.41	7.19	12.0	**16.1**
Turns	7.24	6.75	7.09	9.14	**9.58**
Clover	15.4	25.3	25.6	26.2	**30.3**

	Ctrl Fail.				
World	BB	PD1	**PD2**	PD3	Human
Rect	0%	0.69 %	**0.79%**	6.55%	10.1%
Turns	2.67%	0.87%	**0.07%**	9.57%	17.5%
Clover	7.89%	0.26%	**0%**	0.21%	0%

	Plant Fail.				
World	BB	PD1	PD2	PD3	Human
Rect	0%	0.69 %	**0.79%**	6.14%	0.04%
Turns	2.33%	0.74%	**0%**	9.46%	13.8%
Clover	1.17%	0%	**0%**	0%	0%

8.5.2 AirSim Results

We evaluated constructive VeriPhy using the same AirSim test environments as the classical VeriPhy evaluation from Section 3.7.3. The results of the evaluation are presented in Table 8.1 using the same table format as Section 3.7.3. Recall that the experiments use three driving environments (Rect, Turns, and Clover) in order to test a variety of turning maneuvers, with the Turns environment featuring the tightest, lowest-speed turns and the Clover environment featuring the widest, highest-speed turns. The experiments were repeated using a bang-bang controller (BB), three proportional-derivative controllers (PD1, PD2, and PD3) which are each tuned with different coefficients for the proportional and derivative terms of their control laws, and manual piloting by the author (Human). Bang-bang control and proportional-derivative control are both standard control algorithms, but using both in the experiments let us demonstrate constructive VeriPhy's behavior when used with various controller implementations. The experiments used sandbox control code which was synthesized from the Kaisar port of the 2D model Chapter 3. For more on the Kaisar model and proof, see the"RA-L" model in Section 7.5 from Chapter 7.

The comparison of AirSim results between classical VeriPhy and constructive VeriPhy is not one-to-one because different versions of AirSim and Unreal Engine[10] were used and mi-

[10]This chapter uses AirSim version 1.3.1, which is based on Unreal Engine version 4.24. In contrast, Chapter 3 uses AirSim 1.2.0, which is based on Unreal Engine 4.18.

nor enhancements[11] were made to our own control algorithm since the original experiments. Moreover, for the sake of convenience, the statistics in Table 8.1 were computed as part of the code we added to AirSim, and are computed using floating-point numbers. While the constructive VeriPhy monitor uses rational rather than floating point computations, we expect the floating-point computations to agree closely with the rational computations because the rational number data structures in constructive VeriPhy, in contrast to classical VeriPhy, support arbitrarily high precision. When converting from floating point to rational numbers, we uniformly use a denominator of 1000 (e.g., recording distances in millimeters), but could just as easily use any denominator which fits in a 32-bit word. The 32-bit limitation is not part of constructive VeriPhy but is a part of a protocol we developed for communicating between AirSim and constructive VeriPhy, which could be relaxed if a higher precision were found necessary. There are two reasons the arithmetic precision issues experienced in classical VeriPhy were not experienced in constructive VeriPhy when using 32-bit words: firstly, precision in classical VeriPhy was not a problem for the values of individual variables, just *results* of computations, which are arbitrary-precision in constructive VeriPhy; secondly, when numerator and denominator are each 32 bits, the total precision is greater than 32 bits. As before, VeriPhy and AirSim communicate by passing the values of variables across a bidirectional pipe using the Windows named pipe API.

Compared to Section 3.7.3, our results sometimes report higher error rates and at other times report lower error rates. The medium-speed PD controller (PD2) has the best error rates. It exhibited no plant violations on the Turns and Clover environments, which is significant because plant violations, in contrast to control violations, must be avoided if provable safety is to be guaranteed. Its plant failure rate on the Rect model is an order of magnitude smaller than the lowest rate (8.23%) achieved by the low-speed PD controller on the same level in the prior evaluation. Its control monitor failure rate is slightly lower than before and its average velocity (inversely proportional to completion time) is comparable or better. The better completion time in the Turns level is likely owed to the improved control algorithm which reduces oversteering during the tight, low-velocity turns required by that environment. Minor data bugs were also found in the plan of the Turns level during this evaluation, so the removal of those bugs could also change the average velocity: for example, the turning radius may have increased as a result of these changes, which can increase turning velocity in practice. The Clover level in particular exhibited a drastic improvement in plant monitor failure rates. Our original hypothesis (Section 3.7.3) was that high-speed drifting behaviors caused model compliance to fail, but those behaviors are also present in the new evaluation, falsifying that hypothesis. It is possible that the past monitor failures were due to limited arithmetic precision: it is certainly the case that the Clover level has the highest velocities and greatest distances between waypoints, meaning it was at the greatest risk of arithmetic limitations in classical VeriPhy. While we do not have a single conclusive reason for increased system performance, possible contributing factors are the use of high-precision arithmetic and the use of revised control algorithms.

[11]Specifically, old versions of the controller tended to oversteer at the boundary between path segments because their distance heuristics spuriously reported a high distance from the intended path. The revised heuristics report a less conservative distance between car and path which reduces oversteering at path segment boundaries.

Because the plant failure rates are zero in the Turns and Clover environments and low in the Rect environment, it is tempting to conclude that safety is entirely guaranteed for the PD2 controller in the Turns and Clover environments and nearly guaranteed in practice. It is important to recognize however that the safety property assessed in this experiment (and Section 3.7.3) is a subtle one which should not be construed as absolute safety. We enforce the choice of accelerations which agree with a given speed limit and monitor the compliance of curved motion to a given plan within a given tolerance. When the plan is a safe one, the system will be safe, but the choice of plan is safety-critical.

To assess the significance of trusted planners, we briefly recapitulate the implementations of plans in our experimental setup. Because planning and sensing are not the primary focuses of the work, each environment is accompanied by a handwritten path, whose safety has been inspected manually. However, the planner dynamically adapts the curvature of the current path segment to provide the car with a feasible path to return to the desired waypoint, should it begin to stray. Dynamically planning significantly improves monitor compliance even in simple cases: if a car is attempting to follow a straight line but is pointing at a slight angle, replacing the straight line with a gentle curve makes system safety invariants easier to satisfy. In extreme cases where the vehicle has actually left the intended path, the replanner will instruct the vehicle to follow a new path to the correct waypoint, rather than stopping execution because the vehicle has violated a safety invariant. Thus, it is possible for our monitor to accept a system execution where the vehicle has left the handwritten path so long as the vehicle complies with trajectories proposed by the replanner. The implications for safety are significant: our low failure rates do not imply that a hard-coded path is followed precisely, rather they show that the controller succeeds in following a path which is chosen cooperatively by the replanner.

It would be interesting to run the experiments without the replanner and compare the results. In general, the use of the replanner leads to a weaker and more subtle notion of safety, but greatly improves liveness. Without the use of a replanner, it is not only possible but common for the system to fail liveness: if the vehicle brakes to a stop while facing the wrong direction, the controller monitor will continue to fire and the vehicle will never accelerate again. The fallback controller causes the system to brake but takes no corrective steering action, so it is important to avoid steering violations through some other mechanism. In our case, we avoided steering violations by ensuring the planner always proposes a steerable path.

The relatively high control monitor failure rates for the human pilot is most likely due to the fact that the pilot applied the maximum acceleration value throughout the experiment. The control monitor overrides the maximum acceleration when necessary to obey the speed limits set by each environment. The human pilot had a higher average velocity in this evaluation than in Section 3.7.3, but the difference could be explained by natural variation in pilot behavior between different experiment runs.

In principle, the more flexible arithmetic of constructive VeriPhy may allow slightly more aggressive control because it can reduce spurious monitor violations. While constructive VeriPhy improved upon the failure rates and completion times of classical VeriPhy, particularly in the PD2 configuration, it is not clear whether or not the improvement owes mainly to arithmetic. While the conclusions drawn from the numeric comparison between

classical VeriPhy and constructive VeriPhy are limited, our qualitative experience developing the experiment is also an important factor in evaluating the usability of constructive VeriPhy. Those experiences are discussed in Section 8.5.3.

8.5.3 Lessons Learned

We discuss our experience using constructive VeriPhy to implement the 1D and 2D evaluations. Specifically, we discuss our anecdotal user experience in order to assess the extent to which constructive VeriPhy has resolved the classical VeriPhy as well as any practical limitations in constructive VeriPhy which manifested during the evaluation process. Those limitations, when they arise, highlight the value of proposed future work such as the compilation of high-level synthesized code to machine code.

The discussion should be treated as anecdotal because they are the experiences of constructive VeriPhy's developer. It is important to recognize that the experience of a tool's developer is not reflective of a typical user. A tool's developer has a more intimate familiarity with it than does a typical user, yet will also experience hurdles that most users do not, because they have interacted with the tool during its development. In this sense, the user experience discussed below simultaneously reflects a less difficult and more difficult experience than a typical user would have. Though it is important to recognize that the reported user experiences are influenced by interaction with development versions of constructive VeriPhy, our goal is not to discuss our experience developing and debugging constructive VeriPhy. On the other hand, we are interested in our experience developing and debugging a CPS implementation which incorporates code synthesized by constructive VeriPhy, since the development and debugging of such systems is a standard task for users of constructive VeriPhy and is thus a task which we wish to make easier.

We begin by discussing the extent to which constructive VeriPhy met its goals of resolving limitations observed in classical VeriPhy. Limited arithmetic precision was an important limitation in classical VeriPhy and was exercised in the 2D driving example (Section 8.5.2). By providing rational arithmetic with flexible precision, constructive VeriPhy allowed the implementation to use whichever physical units of measure the user finds most convenient, whereas classical VeriPhy required the user to carefully choose units of measure that fit within the available precision.

Another important goal of the constructive VeriPhy implementation was to provide readable and traceable error messages. The GoPiGo evaluation was performed using an early version of constructive VeriPhy with basic error messages: when a plant monitor fails, the failing formula and entire program state are displayed. Our error messages did not display source code locations for failing tests or simplify the state by eliminating irrelevant SSA variables. For our relatively short models, these basic error messages were already helpful, but this experience led us to implement a more polished error-reporting mechanism for subsequent use, for example by adding including source line numbers to improve traceability of failing formulas to source model positions. Based on our experience, these error messages are an important tool which can help users of constructive VeriPhy debug their system implementations. Because classical VeriPhy does not have access to source line numbers, reporting them in error messages would be harder in classical VeriPhy.

Practical use of classical VeriPhy often encountered the limitation that models are restricted to a specific format: the system must be a single Demonic loop with a controller followed by a single ODE system, where the controller is a top-level choice containing the control branches. Classical VeriPhy would always insert a fallback case into the controller. While we did not go out of our way to test exotic modeling paradigms, our tests demonstrated that constructive VeriPhy supports various model shapes which arose naturally in our experiments. In our models, both the looping constructs and controller branches varied between Angelic and Demonic versions. In the Angelic Sandbox model, Demon supplies a suggested speed before the controller, which is an example of a common behavior that falls outside the fixed format from classical VeriPhy. The model used for the AirSim evaluation had nested case analyses and multiple instances of its ODE on different branches. The proofs differ in the number of explicit proof assertions and their positions. None of these variations required fundamental changes to the VeriPhy implementation, aside from revealing typical, unremarkable bugs from the initial prototype of the VeriPhy implementation. The Reach-Avoid model synthesized a live, concrete controller, a feature which was missing from classical VeriPhy.

While constructive VeriPhy does not require the use of the classical VeriPhy-style FFI interface for external interaction, we decided to build a wrapper for the FFI interface in constructive VeriPhy in order to maximize code reuse from Chapter 3 and maximize consistency between the evaluations of both VeriPhy implementations. The wrapper ultimately constituted a significant fraction of the design and debugging effort for the experiments because it had to reconcile constructive VeriPhy's `DemonStrategy` interface, where variables are read and assigned one at a time, with classical VeriPhy's FFI interface, where variables are read and assigned in blocks. Because classical VeriPhy maintains lists of the controller and sensor variables but constructive VeriPhy does not, the wrapper takes lists of the controller and sensor variables as arguments. We implemented a Scala class which accepts a shared library that implements a bulk FFI-based interface and translates it to the strategy interface. The translation relies on buffering: when one sensor value is needed, all current sensor values are read and stored for future access. Assignments to controller variables are likewise buffered, not actuated immediately. Buffering is implemented by distinguishing a program variable which, when assigned, triggers the actuation of all buffered writes to controller variables and triggers emptying of the buffer. Specifically, we use a time variable (e.g. t) as the trigger for actuation because time-triggered models typically initialize a time variable (e.g., t) to 0 immediately before running the plant, and because it is standard to perform actuation immediately before running the plant. Our wrapper allowed us to reuse code from the classical VeriPhy experiments with minimal change, but as with classical VeriPhy, the FFI interface assumes there are two distinct stages in the (single) main loop of the model: control and sensing. Because constructive VeriPhy is meant to support a wide variety of model shapes, the FFI interface wrapper is not intended for use in *all* applications of constructive VeriPhy. The takeaway of the FFI wrapper is that constructive VeriPhy provides significant support for adapting classical VeriPhy-based implementations, though we do not expect the adaptation process to be entirely automatic, since the wrapper was built with our own models in mind and may not support different models out-of-the-box. Users of constructive VeriPhy are encouraged to

use either block-based or variable-based interfaces as they see fit.

We reflect on how our choice of implementation language and libraries impacted the usage of constructive VeriPhy. CPSs are often implemented using low-level languages such as C where memory can be managed manually and where it is often possible to write controllers which satisfy tight constraints on CPU, power, memory, and disk usage. Our FFI-based interface did integrate a low-level controller written in C, which was compiled to a shared library and then loaded dynamically. However, the main function of the VeriPhy interpreter is written in Scala and executed on the JVM rather than implemented in C.

During development, performance issues *were* encountered on the GoPiGo. The initial VeriPhy executable (i.e., `.jar` archive) included the full KeYmaera X implementation and all of its dependencies, with a total file size of ≈51 MiB. The first control cycle took over 3 seconds, which caused a plant monitor violation and caused the robot not to drive at all. We suspect but have not proved that this high startup time is due to JIT compilation or dynamic initialization which occurred not at the beginning of the main function but when first invoking the `spire` library.

After encountering high startup times, the VeriPhy implementation was revised to remove a dependency on the full `spire` library (now using only its rational number type) as well as all KeYmaera X code and dependencies that were not required for the main VeriPhy execution function. The resulting executable has a compressed size of ≈13 MiB and uncompressed size of ≈35 MiB. A large fraction of the size is due to the Scala standard library, whose uncompressed size in the executable is ≈22 MiB. The (uncompressed) size of the JNA library for JVM-to-C interoperability was an additional ≈4 MiB, so we suspect that the executable size could not be reduced much further without removing the standard library, C interoperation, or unused parts thereof.

The smaller executable did not exhibit a noticeable startup delay. As a precautionary measure, we also added the ability to warm-start the experiments by performing dummy calls to the FFI library before starting the VeriPhy interpreter, which would eliminate any hypothetical costs associated with loading of the shared library.

The practicality of our present implementation varies greatly between different CPS platforms. Microcontrollers with minuscule amounts of memory would have no hope of hosting a JVM, let alone executing a 13 MiB executable. Single-board computers such as the Raspberry Pi, however, have sufficient resources to run our executables so long as dependencies on large libraries are avoided.

Regardless of the fact that we successfully ran experiments on the Raspberry Pi, it remains our goal in future work to provide a compiler backend for constructive VeriPhy which would result in much smaller executables with more predictable memory usage and running time. A compilation backend would make the use of constructive VeriPhy on highly resource-constrained systems much more likely to succeed.

The development of a compilation backend would also serve to eliminate some development inconveniences we encountered which are common to the use of FFI's and shared libraries. Eliminating those inconveniences is valuable because future users of constructive VeriPhy would likely encounter the same inconveniences in their own applications. Our C functions interact with the JVM by passing values through an array, so C code must be written with the name and order of variables in mind. By generating C stubs as part of the

compilation process, one can eliminate the potential for integration bugs where arrays are packed and unpacked into incorrect variables, bugs which we did experience during development. A compilation-based approach would also simplify the use of debugging messages in the driver implementation: the use of I/O in shared libraries is often discouraged if not necessarily impossible, so our driver took the more complicated approach of storing debugging messages in a buffer where the Scala wrapper could query them and display them to the driver developer. Static linking of libraries by the compiler would also reduce the likelihood that a driver developer accidentally attempts to use a library which is compiled with an incorrect architecture or calling convention. While the elimination of dynamic linking errors is not our priority, it is always beneficial to make an entire class of development mistakes more difficult.

Further experience was gained during the development of the 2D AirSim-based evaluation. The development effort was dominated by development of an inter-process communication protocol between AirSim (C++) and constructive VeriPhy as well as debugging[12] of the replanner and steering controllers in AirSim. We find it promising that the development bottlenecks were not specific to VeriPhy: for example, we did not have to manage limited arithmetic precision and had an easier time interpreting monitor failures. The constructive VeriPhy error reporting code preprocessed monitor failures to identify the key information: when the monitor is a conjunction of invariants, constructive VeriPhy reports specifically which invariants failed. Detailed error reporting helpde because failures of different invariants often have different causes: the "car is on the path" invariant can fail when replanning is disabled, while the "waypoint has positive y coordinate" invariant can fail if the planner fails to switch to the next waypoint when a previous waypoint is reached.

In summary, our evaluations went beyond showing that constructive VeriPhy met its core goals of extracting concrete control code and support a flexible input language without the fixed input format restrictions of classical VeriPhy. Supporting features such as rational arithmetic and detailed error messages respectively supported the user in developing systems which have fewer arithmetic bugs to begin with and in identifying bugs more easily. The FFI wrapper aided in adapting applications of classical VeriPhy to use constructive VeriPhy. A notable practical limitation that remains is the use of an interpreter for strategies rather than a compiler that produces low-level code. While our evaluation showed that interpretation could be made to work on our chosen platforms, it also showed that compilation can promise greater performance and simpler interaction with FFIs, as expected, thus motivating future work. Because classical VeriPhy successfully used compilation in its backend, that future work is expected to be feasible.

[12]While the implementation of the car controller was debugged during the previous evaluation, there exist several copies of the code with varying levels of correctness, so debugging effort was nonetheless expended to identify the correct implementation.

Chapter 9

Conclusion

In this thesis, we advanced the state of the art in end-to-end verification of hybrid systems in multiple ways, to address the competing needs of our Logician, Engineer, and Logic-User:

- We placed dL on its firmest foundation yet with an Isabelle/HOL formalization of soundness, culminating in a verified proof term checker which has been tested on KeYmaera X proofs exceeding 10^5 steps.
- We developed classical VeriPhy, which consumes dL models and proofs to generate sandbox controllers with correctness guarantees down to machine-code level. Classical VeriPhy consumes classical dL proofs whose soundness it now trusts more due to the Isabelle/HOL soundness formalization. A further Isabelle/HOL formalization of fixed-point arithmetic was developed to justify the arithmetic compilation pass of VeriPhy; that formalization reuses the dL expression and semantics definitions from the former formalization to show soundness of the interval semantics with respect to real-valued dL semantics. VeriPhy was evaluated on 1D and 2D ground robotics case studies, which revealed limitations regarding types of code generated, arithmetic computation, debuggability, and robustness to complex models. The Logician is satisfied with VeriPhy's solid foundations, but both the Engineer and Logic-User face significant hurdles in practice.
- Constructive game logic was introduced as a logic for discrete games (CGL) and hybrid games (CdGL). It was motivated by the observation that many of the Engineer's and Logic-User's struggles centered around the fact that synthesis was only supported for restrictive modeling and proof formats, as well as the desire to synthesize whitebox controllers in addition to sandboxes. Constructivity provides a foundation for exhaustive general-case synthesis procedures, while games promise a particularly convenient setting for combined synthesis of monitoring and control. Games additionally often provide models which are more concise, thus more convenient for the Logic-User, because models can express controllers abstractly and delegate their definition to the proof.

  Constructive Game Logic features a Curry-Howard interpretation of games, which says that constructive proofs are computable winning strategies. The Curry-Howard interpretation of games provides a rigorous basis for synthesizing code which plays

the computable winning strategy, without being restricted to the fixed formats used in classical VeriPhy.

A refinement calculus manifests the relationship between strategies and proved games by recovering hybrid systems from hybrid games' winning strategies. The ability to extract systems from proved games suggests a loosely-coupled design architecture for tools that process game proofs: once a proved game is transformed into a system, the rest of the implementation can operate over hybrid systems, which are a simpler language than hybrid games.

- We introduce Kaisar to slay the twin dragons of unmaintainable proofs and ill-structured inputs for synthesis. The constructive, natural deduction calculus of CdGL provides a theoretical basis for proof structuring, which in Kaisar is presented through proofs annotated not only by invariants but by full-powered first-order structured proofs. Kaisar's structuring principles support the Logic-User by providing clear visual correspondence between proof structure and model structure and by streamlining reasoning across changing program states. Behind the scenes, ghost reasoning is used to rigorously implement reasoning across changing program states.
- We introduce constructive VeriPhy, which is compared to classical VeriPhy below.
- Constructive VeriPhy can synthesize whitebox controllers, not just blackbox controllers based on monitors.
- Constructive VeriPhy uses arbitrary-precision rational arithmetic instead of fixed-precision integer arithmetic, greatly reducing the risk of conservative arithmetic causing monitor failures.
- Constructive VeriPhy can synthesize code from any Kaisar proof. In contrast to the fixed format of classical VeriPhy, it supports synthesis from proofs which feature (for example) nested case analyses, multiple loops, and multiple ODE systems. In this sense, constructive VeriPhy provides the more robust user experience.
- The Kaisar language is designed to provide greater traceability (i.e., ease of tracking the relation between model statement and corresponding proof step) compared to the hybrid systems proof languages that precede it and also emphasizes features such as block structuring and lexical scope which are expected to provide greater usability in large-scale proofs. Because modeling and proof in Kaisar are a key step of the (constructive) VeriPhy workflow, the workflow overall is made more convenient for large-scale proof. Because constructive VeriPhy consumes a Kaisar proof, runtime failures in a constructive VeriPhy monitor can be traced all the way back to the Kaisar proof, for example by printing source location information in monitor failure messages.
- As of this writing, constructive VeriPhy does *not* rigorously prove the transfer of safety and liveness to generated code, let alone with the exhaustive chain of artifacts that classical VeriPhy does. Such an artifact chain would include machine-checked semantic proofs about CdGL, whose difficulty was discussed in the chapters about the foundations of CdGL. The semantics of CdGL use a combination of features which is difficult to find in modern theorem-provers: inductive families with non-syntactic

well-foundedness arguments, infinite towers of predicational universes, and large eliminations which not only use universe polymorphism but *polymorphic recursion* over the universe. While the difficulty of formalizing CdGL in the future should not be understated, we have identified the formalization obstacles which future work would need to overcome.

- A key step in the end-to-end argument of classical VeriPhy is the use of a CakeML-based verified compilation backend. Constructive VeriPhy does not currently have such a backend. Instead, it generates a low-level IR, then executes it in an interpreter written in Scala. In future work, the creation of a verified compilation backend for constructive VeriPhy is expected to pose far fewer challenges than a formalization of CdGL semantics, but is outside the scope of the thesis because it would require restructuring, generalizing, and reproving (in HOL4) the compilation backend of classical VeriPhy to support free-format programs rather than fixed-format ones.

While we are hopeful that mechanizations of CdGL semantics and especially verified compilation backends could make the Logician even happier in the future, our point is not that the constructive implementation alone would make them happy in the future. Rather, the Logician is happy now because they are able to choose freely between classical and constructive VeriPhy, which were built with different tradeoffs in mind. Classical VeriPhy demonstrated that a full chain of proof artifacts is possible, but it also demonstrated the high level of effort required to apply the tool in complex settings. If the Logician wants the absolute strongest chain of evidence, they must invest the corresponding effort. Constructive VeriPhy showed how to support a broader class of systems while adding support for live whitebox controllers and providing a more robust implementation with lower expected effort. Constructive VeriPhy prioritizes the Logic-User and Engineer over the Logician, but does not disregard the Logician completely; after all, constructive VeriPhy justifies its correctness on the basis of our foundational CdGL developments. The Logician, Logic-User, and Engineer are meant to represent extremes, and for those of us who lie in between, constructive VeriPhy makes important strides for our inner Logic-User and Engineer without discarding our inner Logician.

Possible future work falls into several categories:

- Moving the endpoints: both versions of VeriPhy assume sensing and actuation are correct. Verification of real sensors and actuators is valuable, orthogonal work.
- Closing the gaps: constructive VeriPhy makes life significantly better for the Logic-User and Engineer compared to classical VeriPhy, but classical VeriPhy provides a far cleaner story for the Logician. It is valuable to provide rigorous proofs and full formal artifacts in constructive VeriPhy to best satisfy the needs of all three characters simultaneously.
- Expanding the problem domain: Many CPSs include stochastic components or wish to ensure security properties, the latter of which the author has explored elsewhere (Bohrer & Platzer, 2018). Code extraction for stochastic systems would be useful because stochastic systems are commonplace and need to be correct. At the same time, correct code extraction for these systems would introduce novel theoreti-

cal challenges because, just as VeriPhy required an understanding of the foundations of hybrid systems and hybrid games, correct extraction of code for stochastic systems would require an understanding of stochastic foundations. For example, one would need to show that a randomized program faithfully implements the randomness of the model. Security-preserving code extraction introduces its own interesting theoretical challenges as well. For example, it would require the use of proof techniques beyond those of Chapter 3 because some security properties are *hyperproperties*, properties of *sets* of program executions. Additionally, a security-preserving code extraction algorithm, compared to those of VeriPhy, would need to expend extra effort to ensure that any nondeterminism in an input model is still present in the synthesized code. This need arises from a phenomenon called the *refinement paradox*, referring to the fact that a secure, nondeterministic program can be made less secure merely by making it more deterministic. While only a paradox in the colloquial sense, the interaction between nondeterminism and security indicates that security-preserving code extraction from nondeterministic models would be a nontrivial undertaking that would yield fundamental technical contributions.

- Applications: A wide variety of CPS applications have been verified in dL, and many others wait to be verified. Verifying such models, synthesizing code and integrating the code with production systems can provide a powerful correctness safety net for practical use.

This thesis addresses topics whose definitions are ever-changing and at times subjective: practical and end-to-end verification. However, this high-level thesis statement is by design, because the thesis links together diverse topics and viewpoints: chapters range from foundational to applied, classical to constructive, and from systems to games. It is important to note the contributions of individual chapters are often of independent interest beyond the context of end-to-end CPS verification. In particular, we hope that our constructive foundations for program logics may bear fruit beyond CdGL and we hope that Kaisar's combination of annotations, first-order proving, and labeled reasoning may prove a useful paradigm in domains other than CPS.

Conversely, while no one thesis can conclusively solve a broadly-defined topic such as practical and end-to-end verification, a holistic perspective on the thesis's contributions is worthwhile. Correctness for CPSs is difficult in part because many levels of abstraction are in play: one member of a development team may focus their attention on high-level dynamic considerations, another on low-level control code, and yet another on sensing and actuation. In such a rapidly evolving field, the range of technical specialties and the size of codebases both continue to grow. Rapid growth can pose a challenge to the Logician and Logic-User, whose capacity to verify software may struggle to keep pace with rapid technical advancement.

A good parting message for the thesis is that while it may be a long time before the Logician and Logic-User see total victory, they also have no reason to despair. In foundational work, choosing the right abstractions is half the battle, and abstraction is likewise fundamental to coping with rapid technological advancement. Once we chose the abstraction of games and developed CdGL, we could manage complexity by refining the

high-level abstraction to a lower-level abstraction: systems. Likewise, Kaisar provides the abstractions of persistent contexts and label-based reasoning, which it lowers to CdGL-like proofs. The key to VeriPhy's effectiveness is that monitoring allows it to abstract away the implementation of a controller and only consider the control decisions reached.

Good abstractions do more than hide the complexity of modern systems, however. Good abstractions help us notice the challenges that remain, while providing a modular structure that remains open to future solutions for today's challenges. VeriPhy's plant monitors may fail when physical behavior violates model assumptions, but they also help us to notice and address the limitations of our models as early as possible. The very process of writing a formal safety guarantee for VeriPhy immediately suggested the importance of correct sensing and actuation, but by stating those correctness assumptions formally, we lay out future work that can build seamlessly on the present work. The process of developing CdGL for pure strategies of two-player perfect information games immediately reminds us of the value of more general game concepts, but once a given CPS has been modeled in today's abstractions, we gain a fuller picture of what future abstractions can or cannot hope to add.

End-to-end verification for CPS is by no means a closed topic, but logical foundations do not need to close the topic in order to play an indispensable role. If this thesis shows anything, it shows that foundations, large-scale proof, and implementation will continue to serve as one another's catalysts for the foreseeable future, meaning that none of the three should dare to go without the others.

Appendix A

Appendices to Chapter 4

A.1 Proof Calculus

$$(\langle\cup\rangle\text{E})\quad \frac{\Gamma \vdash A : \langle\alpha\cup\beta\rangle\phi \quad \Gamma, \ell : \langle\alpha\rangle\phi \vdash B : \psi \quad \Gamma, r : \langle\beta\rangle\phi \vdash C : \psi}{\Gamma \vdash \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : \psi}$$

$$(\langle\cup\rangle\text{I1})\quad \frac{\Gamma \vdash M : \langle\alpha\rangle\phi}{\Gamma \vdash \langle \ell \cdot M\rangle : \langle\alpha\cup\beta\rangle\phi}$$

$$(\langle\cup\rangle\text{I2})\quad \frac{\Gamma \vdash M : \langle\beta\rangle\phi}{\Gamma \vdash \langle r \cdot M\rangle : \langle\alpha\cup\beta\rangle\phi}$$

$$([\cup]\text{I})\quad \frac{\Gamma \vdash M : [\alpha]\phi \quad \Gamma \vdash N : [\beta]\phi}{\Gamma \vdash [M, N] : [\alpha\cup\beta]\phi}$$

$$(\langle?\rangle\text{I})\quad \frac{\Gamma \vdash M : \phi \quad \Gamma \vdash N : \psi}{\Gamma \vdash \langle M, N\rangle : \langle?\phi\rangle\psi}$$

$$([?]\text{I})\quad \frac{\Gamma, p : \phi \vdash M : \psi}{\Gamma \vdash (\lambda p : \phi.\ M) : [?\phi]\psi}$$

$$([?]\text{E})\quad \frac{\Gamma \vdash M : [?\phi]\psi \quad \Gamma \vdash N : \phi}{\Gamma \vdash (M\ N) : \psi}$$

$$([\cup]\text{E1})\quad \frac{\Gamma \vdash M : [\alpha\cup\beta]\phi}{\Gamma \vdash [\pi_L M] : [\alpha]\phi}$$

$$([\cup]\text{E2})\quad \frac{\Gamma \vdash M : [\alpha\cup\beta]\phi}{\Gamma \vdash [\pi_R M] : [\beta]\phi}$$

$$(\text{hyp})\quad \frac{}{\Gamma, p : \phi \vdash p : \phi}$$

$$(\langle?\rangle\text{E1})\quad \frac{\Gamma \vdash M : \langle?\phi\rangle\psi}{\Gamma \vdash \langle \pi_L M\rangle : \phi}$$

$$(\langle?\rangle\text{E2})\quad \frac{\Gamma \vdash M : \langle?\phi\rangle\psi}{\Gamma \vdash \langle \pi_R M\rangle : \psi}$$

Figure A.1: CGL proof calculus: propositional rules.

In rule $\langle*\rangle$I, the "formula" $\mathcal{M}_0 = \mathcal{M} \succ \mathbf{0}$ is not a proper CdGL formula but a shorthand for the proper CdGL formula $\mathcal{M}_0 = \mathcal{M} \wedge \mathcal{M} \succ \mathbf{0}$.

$$(\langle*\rangle\mathrm{C})\quad \frac{\Gamma \vdash A : \langle\alpha^*\rangle\phi \quad \Gamma, s : \phi \vdash B : \psi \quad \Gamma, g : \langle\alpha\rangle\langle\alpha^*\rangle\phi \vdash C : \psi}{\Gamma \vdash \langle \mathrm{case}_*\ A \text{ of } s \Rightarrow\ B \mid g \Rightarrow C\rangle : \psi}$$

$$(\mathrm{M})\quad \frac{\Gamma \vdash M : \langle\alpha\rangle\phi \quad \Gamma_{\frac{\vec{y}}{\mathrm{BV}(\alpha)}}, p : \phi \vdash N : \psi}{\Gamma \vdash M \circ_p N : \langle\alpha\rangle\psi}\ {}^{1}$$

[1]Variables $\vec{y}$ are fresh and $|\vec{y}| = |\mathrm{BV}(\alpha)|$

$$([*]\mathrm{E})\quad \frac{\Gamma \vdash M : [\alpha^*]\phi}{\Gamma \vdash [\text{unroll } M] : \phi \wedge [\alpha][\alpha^*]\phi}$$

$$(\langle*\rangle\mathrm{S})\quad \frac{\Gamma \vdash M : \phi}{\Gamma \vdash \langle\text{stop } M\rangle : \langle\alpha^*\rangle\phi}$$

$$(\langle*\rangle\mathrm{G})\quad \frac{\Gamma \vdash M : \langle\alpha\rangle\langle\alpha^*\rangle\phi}{\Gamma \vdash \langle\text{go } M\rangle : \langle\alpha^*\rangle\phi}$$

$$(\langle\!\langle:=\rangle\!\rangle\mathrm{I})\quad \frac{\Gamma_{\frac{y}{x}}, p : (x = f_{\frac{y}{x}}) \vdash M : \phi}{\Gamma \vdash \langle\!\langle x := f_{\frac{y}{x}} \text{ in } p.\ M\rangle\!\rangle : \langle\!\langle x := f\rangle\!\rangle\phi}\ {}^{1}$$

$$(\langle:*\rangle\mathrm{E})\quad \frac{\Gamma \vdash M : \langle x := *\rangle\phi \quad \Gamma_{\frac{y}{x}}, p : \phi \vdash N : \psi}{\Gamma \vdash \mathrm{unpack}(M, py.\ N) : \psi}\ {}^{2}$$

$$([:*]\mathrm{I})\quad \frac{\Gamma_{\frac{y}{x}} \vdash M : \phi}{\Gamma \vdash (\lambda x : \mathbb{Q}.\ M) : [x := *]\phi}\ {}^{3}$$

$$(\langle:*\rangle\mathrm{I})\quad \frac{\Gamma_{\frac{y}{x}}, p : (x = f_{\frac{y}{x}}) \vdash M : \phi}{\Gamma \vdash \langle f_{\frac{y}{x}} :* p.\ M\rangle : \langle x := *\rangle\phi}\ {}^{4}$$

$$(\mathrm{split})\quad \Gamma \vdash \mathrm{split}\ [f \sim g] : f \leq g \vee f > g$$

$$(\langle\!\langle d\rangle\!\rangle\mathrm{I})\quad \frac{\Gamma \vdash M : \langle\!\langle\alpha\rangle\!\rangle\phi}{\Gamma \vdash \langle\!\langle\text{yield } M\rangle\!\rangle : \langle\!\langle\alpha^d\rangle\!\rangle\phi}$$

$$([*]\mathrm{R})\quad \frac{\Gamma \vdash M : \phi \wedge [\alpha][\alpha^*]\phi}{\Gamma \vdash [\text{roll } M] : [\alpha^*]\phi}$$

$$(\langle\!\langle;\rangle\!\rangle\mathrm{I})\quad \frac{\Gamma \vdash M : \langle\!\langle\alpha\rangle\!\rangle\langle\!\langle\beta\rangle\!\rangle\phi}{\Gamma \vdash \langle\!\langle\iota\ M\rangle\!\rangle : \langle\!\langle\alpha;\beta\rangle\!\rangle\phi}$$

$$([:*]\mathrm{E})\quad \frac{\Gamma \vdash M : [x := *]\phi}{\Gamma \vdash (M\ f) : \phi^f_x}\ {}^{5}$$

$$(\mathrm{iG})\quad \frac{\Gamma, p : x = f \vdash M : \phi}{\Gamma \vdash \mathrm{Ghost}[x = f](p.\ M) : \phi}\ {}^{6}$$

$$(\mathrm{FO})\quad \frac{\Gamma \vdash M : \rho}{\Gamma \vdash \mathsf{FO}[\phi](M) : \phi}\ {}^{7}$$

$$(\mathrm{Dec})\quad \frac{\Gamma \vdash M : \rho}{\Gamma \vdash (\mathsf{Dec}[\phi \vee \psi](M)) : \phi \vee \psi}\ {}^{8}$$

[1]y fresh in Γ and $\langle\!\langle x := f\rangle\!\rangle\phi$
[2]y fresh in $\Gamma, \langle x := *\rangle\phi, \psi$ and $x \notin \mathrm{FV}(\psi)$
[3]y fresh in Γ and $[x := *]\phi$
[4]y fresh in $\Gamma, f, \langle x := *\rangle\phi$ and p fresh in Γ
[5]ϕ^f_x admiss.
[6]x fresh except free in M, p fresh
[7]exists $\mathsf{b}, \{\mathsf{b}\} \times \mathcal{S} \subseteq [\![\rho \to \phi]\!]$, ρ, ϕ first-order
[8]exists $\mathsf{b}, \{\mathsf{b}\} \times \mathcal{S} \subseteq [\![\rho \to \phi \vee \psi]\!]$, ρ, ϕ, ψ first-order

Figure A.2: CGL proof calculus: first-order games, first-order arithmetic.

$$(\langle * \rangle \mathrm{I}) \quad \frac{\Gamma \vdash A : \varphi \quad p : \varphi, q : \mathcal{M}_0 = \mathcal{M} \succ \mathbf{0} \vdash B : \langle \alpha \rangle (\varphi \wedge \mathcal{M}_0 \succ \mathcal{M}) \quad p : \varphi, q : \mathbf{0} \succcurlyeq \mathcal{M} \vdash C : \phi}{\Gamma \vdash \mathrm{for}(p : \varphi(\mathcal{M}) = A; q; B)\,\{\alpha\}\,C : \langle \alpha^* \rangle \phi}\ {}^{1}$$

$$([*]\mathrm{I}) \quad \frac{\Gamma \vdash M : \psi \quad p : \psi \vdash N : [\alpha]\psi \quad p : \psi \vdash O : \phi}{\Gamma \vdash (M \text{ rep } p : \psi.\ N \text{ in } O) : [\alpha^*]\phi}$$

$$(\mathrm{FP}) \quad \frac{\Gamma \vdash A : \langle \alpha^* \rangle \phi \quad s : \phi \vdash B : \psi \quad g : \langle \alpha \rangle \psi \vdash C : \psi}{\Gamma \vdash FP(A, s.\ B, g.\ C) : \psi}$$

[1] $\mathcal{M}_0$ fresh, $\mathcal{M}$ effectively-well-founded

Figure A.3: CGL proof calculus: loops.

A.2 Full Operational Semantics

We recite the full operational semantics on proof terms here for reference. We begin by reciting the language of proof terms.

Definition A.1 (Proof terms)**.** We define the grammar of proof terms M, N, O (sometimes A, B, C, D, E, F) here, before describing the meaning of each proof term when we describe its corresponding proof rule:

$$\begin{aligned} M, N ::= {} & \lambda x : \mathbb{Q}.\ M \mid M\ f \mid \lambda p : \phi.\ M \mid M\ N \mid \langle \ell \cdot M \rangle \mid \langle r \cdot M \rangle \\ & \mid \langle \text{case}_*\ A \text{ of } s \Rightarrow\ B \mid g \Rightarrow C \rangle \mid \langle \text{case}\ A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C \rangle \\ & \mid \text{for}(p : \varphi(\mathcal{M}) = A; q; B)\ \{\alpha\}\ C \mid FP(A, s.\ B, g.\ C) \\ & \mid M \text{ rep } p : \psi.\ N \text{ in } O \mid [\text{unroll } M] \mid \langle \text{stop } M \rangle \mid \langle \text{go } M \rangle \\ & \mid \langle f \tfrac{y}{x} :* p.\ M \rangle \mid \langle\!\langle \pi_L M \rangle\!\rangle \mid \langle\!\langle \pi_R M \rangle\!\rangle \mid \langle\!\langle M, N \rangle\!\rangle \mid \langle\!\langle \iota\ M \rangle\!\rangle \mid \langle\!\langle \text{yield } M \rangle\!\rangle \\ & \mid \langle\!\langle x := f \tfrac{y}{x} \text{ in } p.\ M \rangle\!\rangle \mid \text{unpack}(M, py.\ N) \mid M \circ_p N \mid \mathsf{FO}[\phi](M) \mid p \end{aligned}$$

where p, q, ℓ, r, s, and g are *proof variables*, that is, variables that range over proof terms of a given proposition. In contrast to the assignable *program variables*, the proof variables are given their meaning by substitution and are scoped lexically, not globally.

We proceed with the β rules. The first-order simplification rules (e.g., FO$\forall\beta$) enable a simpler notion of normal forms. First-order non-syntactic proofs can be used to provide a proof of arbitrary first-order facts. Our rules show that even non-syntactic proofs of first-order facts support normalization which reduces them to syntactic combinations of non-syntactic proofs of atomic propositions. In Fig. A.5, when rule stop$\circ$ is applied to program α^* then $\vec{a} = \text{BV}(\alpha)$ and $\vec{a}'$ is the vector of corresponding ghost variables. The renaming in $[*]\circ$ is likewise. In rule $[\cup]\circ$ for game $\alpha \cup \beta$, the vectors $\vec{a}$ and $\vec{b}$ do not contain *all* variables of $\text{BV}(\alpha)$ and $\text{BV}(\beta)$, only those which are exclusive to each branch, i.e., $\text{BV}(\alpha) \setminus \text{BV}(\beta)$ and $\text{BV}(\beta) \setminus \text{BV}(\alpha)$, respectively. Vectors $\vec{a}'$ and $\vec{b}'$ are again vectors of ghost variables corresponding to $\vec{a}$ and $\vec{b}$. For more information, see Section 4.9.

$(\lambda\phi\beta)$ $(\lambda p : \phi.\ M)\ N \mapsto [N/p]M$ $\qquad$ $(\pi_L\beta)$ $\langle\!\langle \pi_L \langle\!\langle M, N \rangle\!\rangle \rangle\!\rangle \mapsto M$

$(\lambda\beta)$ $(\lambda x : \mathbb{Q}.\ M)\ f \mapsto M^f_x$ $\qquad$ $(\pi_R\beta)$ $\langle\!\langle \pi_R \langle\!\langle M, N \rangle\!\rangle \rangle\!\rangle \mapsto N$

$(\text{case}\beta\text{L})$ $\langle\!\langle \text{case } \langle\!\langle \ell \cdot A \rangle\!\rangle \text{ of } p \Rightarrow\ B \mid q \Rightarrow\ C \rangle\!\rangle \mapsto [A/p]B$

$(\text{case}\beta\text{R})$ $\langle\!\langle \text{case } \langle\!\langle r \cdot A \rangle\!\rangle \text{ of } p \Rightarrow\ B \mid q \Rightarrow\ C \rangle\!\rangle \mapsto [A/q]C$

$(\text{unroll}\beta)$ $[\text{unroll } [\text{roll } M]] \mapsto M$

$(\text{FO}\forall\beta)$ $\mathsf{FO}[\forall x\, \phi](M) \mapsto (\lambda x : \mathbb{Q}.\ \mathsf{FO}[\phi](M))$

$(\text{FO}\wedge\beta)$ $\mathsf{FO}[\phi \wedge \psi](M) \mapsto \langle\!\langle \mathsf{FO}[\phi](M), \mathsf{FO}[\psi](M) \rangle\!\rangle$

$(\text{FO}\exists\beta)$ $\mathsf{FO}[\exists x\, \phi](M) \mapsto \langle f\tfrac{y}{x} :* p.\ \mathsf{FO}[\phi]([\mathsf{FO}[f\tfrac{y}{x} = f\tfrac{y}{x}]()/p](M_x^{\left(f\frac{y}{x}\right)}))\rangle$

$(\text{FO}\vee\beta)$ $\mathsf{FO}[\phi \vee \psi](M) \mapsto \langle \text{case } \pi_0\mathsf{b} \text{ of } p \Rightarrow \langle \ell \cdot \mathsf{FO}[\phi](M, p)\rangle \mid q \Rightarrow \langle r \cdot \mathsf{FO}[\psi](M, q)\rangle\rangle$ [1]

$(\text{unpack}\beta)$ $\text{unpack}(\langle f\tfrac{y}{x} :* q.\ M\rangle, py.\ N) \mapsto (\text{Ghost}[x = f\tfrac{y}{x}](q.\ [M/p]N))$

$(\text{FP}\beta)$ $FP(A, s.\ B, g.\ C) \mapsto \langle \text{case}_*\ A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow [(r \circ_t FP(t, s.\ B, g.\ C))/g]C\rangle$

$(\text{rep}\beta)$ $(M \text{ rep } p : \psi.\ N \text{ in } O) \mapsto [\text{roll } \langle M, ([M/p]N) \circ_q (q \text{ rep } p : \psi.\ N \text{ in } O)\rangle]$

$(\text{for}\beta)$ $\text{for}(p : \varphi(\mathcal{M}) = A; q; B)\ \{\alpha\}\ C \mapsto$

$\langle \text{case split } [\mathcal{M} \sim 0] \text{ of}$
$\quad \ell \Rightarrow\ \langle \text{stop } [(A, \ell)/(p, q)]C\rangle$
$\mid r \Rightarrow\ \text{Ghost}[\mathcal{M}_0 = \mathcal{M}](rr.\ \langle \text{go } (([A, \langle rr, r\rangle/p, q]B) \circ_t (\text{for}(p : \varphi(\mathcal{M}) = \langle \pi_L t\rangle; q; B)\ \{\alpha\}C))\rangle)\rangle$

[1] Here, b is the realizer corresponding to M. Recall that rule FO is non-effective by design, hence the reflection of semantic constructs (b) into a syntactic rule.

Figure A.4: Operational semantics: β-rules.

$$
\begin{array}{rlrl}
(\lambda\circ) & (\lambda x : \mathbb{Q}.\ M)\circ_q N \mapsto (\lambda x : \mathbb{Q}.\ ([M/q]N)) & ([\iota]\circ) & [\iota\ M]\circ_p N \mapsto [\iota\ (M\circ_q(q\circ_p N))] \\
(\lambda\phi\circ) & (\lambda p : \phi.\ M)\circ_q N \mapsto (\lambda p : \phi.\ ([M/q]N)) & (\langle\iota\rangle\circ) & \langle\iota\ M\rangle\circ_p N \mapsto \langle\iota\ (M\circ_q(q\circ_p N))\rangle \\
([^d]\circ) & [\text{yield } M]\circ_p N \mapsto [\text{yield } (M\circ_p N)] & (\langle\ell\cdot\rangle\circ) & \langle\ell\cdot M\rangle\circ_p N \mapsto \langle\ell\cdot(M\circ_p N)\rangle \\
(\langle^d\rangle\circ) & \langle\text{yield } M\rangle\circ_p N \mapsto \langle\text{yield } (M\circ_p N)\rangle & (\langle r\cdot\rangle\circ) & \langle r\cdot M\rangle\circ_p N \mapsto \langle r\cdot(M\circ_p N)\rangle \\
([\cup]\circ) & [A,B]\circ_q N \mapsto [A\circ_p(N\,\frac{\vec{a}}{\vec{a'}}), B\circ_p(N\,\frac{\vec{b}}{\vec{b'}})] & (\langle\cup\rangle\circ) & \langle A,B\rangle\circ_p N \mapsto \langle A,[B/p]N\rangle \\
(\langle{:}*\rangle\circ) & \langle f\,\frac{y}{x} :* q.\ M\rangle\circ_p N \mapsto \langle f\,\frac{y}{x} :* q.\ [M/p]N\rangle & & \\
(\langle{:}{=}\rangle\circ) & \langle x := f\,\frac{y}{x} \text{ in } q.\ M\rangle\circ_p N \mapsto \langle x := f\,\frac{y}{x} \text{ in } q.\ [M/p]N\rangle & & \\
([:{=}]\circ) & [x := f\,\frac{y}{x} \text{ in } q.\ M]\circ_p N \mapsto [x := f\,\frac{y}{x} \text{ in } q.\ [M/p]N] & & \\
(\text{case}\circ) & \llbracket\text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rrbracket\circ_p D \mapsto \langle\text{case } A \text{ of } \ell \Rightarrow\ B\circ_p D \mid r \Rightarrow\ C\circ_p D\rangle & & \\
([*]\circ) & [\text{roll } M]\circ_p N \mapsto [\text{roll } \langle(\langle\pi_L M\rangle)\circ_p(N\,\frac{\vec{a}}{\vec{a'}}), (\langle\pi_R M\rangle)\circ_t(t\circ_p N)\rangle] & & \\
(\text{stop}\circ) & \langle\text{stop } M\rangle\circ_p N \mapsto \langle\text{stop } ([M/p](N\,\frac{\vec{a}}{\vec{a'}}))\rangle & & \\
(\text{go}\circ) & \langle\text{go } M\rangle\circ_p N \mapsto \langle\text{go } (M\circ_q q\circ_p N)\rangle & &
\end{array}
$$

Figure A.5: Operational semantics: monotonicity rules.

$$(\llbracket\pi_L\rrbracket\mathrm{C}) \quad \llbracket\pi_L\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rrbracket \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \llbracket\pi_L B\rrbracket \mid r \Rightarrow\ \llbracket\pi_L C\rrbracket\rangle$$

$$(\llbracket\pi_R\rrbracket\mathrm{C}) \quad \llbracket\pi_R\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rrbracket \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \llbracket\pi_R B\rrbracket \mid r \Rightarrow\ \llbracket\pi_R C\rrbracket\rangle$$

$$([\cup]\mathrm{C1}) \quad [\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle, N] \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ [B, N] \mid r \Rightarrow\ [C, N]\rangle$$

$$([\cup]\mathrm{C2}) \quad [M, \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle] \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ [M, B] \mid r \Rightarrow\ [M, C]\rangle$$

$$(\langle\cup\rangle\mathrm{C1}) \quad \langle\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle, N\rangle \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \langle B, N\rangle \mid r \Rightarrow\ \langle C, N\rangle\rangle$$

$$(\langle\cup\rangle\mathrm{C2}) \quad \langle M, \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rangle \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \langle M, B\rangle \mid r \Rightarrow\ \langle M, C\rangle\rangle$$

$$\begin{aligned}(\mathrm{stopC}) \quad & \langle \text{stop } \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rangle \\ \mapsto & \langle \text{case } A \text{ of } \ell \Rightarrow\ \langle \text{stop } B\rangle \mid r \Rightarrow\ \langle \text{stop } C\rangle\rangle\end{aligned}$$

$$(\mathrm{goC}) \quad \langle \text{go } \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rangle \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \langle \text{go } B\rangle \mid r \Rightarrow\ \langle \text{go } C\rangle\rangle$$

$$(\langle\ell\cdot\rangle\mathrm{C}) \quad \langle \ell \cdot \langle \text{case } A \text{ of } p \Rightarrow\ B \mid q \Rightarrow\ C\rangle\rangle \mapsto \langle \text{case } A \text{ of } p \Rightarrow\ \langle \ell \cdot B\rangle \mid q \Rightarrow\ \langle \ell \cdot C\rangle\rangle$$

$$(\langle r\cdot\rangle\mathrm{C}) \quad \langle r \cdot \langle \text{case } A \text{ of } p \Rightarrow\ B \mid q \Rightarrow\ C\rangle\rangle \mapsto \langle \text{case } A \text{ of } p \Rightarrow\ \langle r \cdot B\rangle \mid q \Rightarrow\ \langle r \cdot C\rangle\rangle$$

$$\begin{aligned}(\mathrm{caseC}^*) \quad & \langle \text{case}_* \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \text{ of } s \Rightarrow\ D \mid g \Rightarrow E\rangle \\ \mapsto & \langle \text{case } A \text{ of} \\ & \quad \ell \Rightarrow\ \langle \text{case}_* B \text{ of } s \Rightarrow\ D \mid g \Rightarrow E\rangle \\ & \mid r \Rightarrow\ \langle \text{case}_* C \text{ of } s \Rightarrow\ D \mid g \Rightarrow E\rangle\rangle\end{aligned}$$

$$\begin{aligned}(\mathrm{caseC}) \quad & \langle \text{case } \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \text{ of } \ell \Rightarrow\ D \mid r \Rightarrow\ E\rangle \\ \mapsto & \langle \text{case } A \text{ of} \\ & \quad \ell \Rightarrow\ \langle \text{case } B \text{ of } \ell \Rightarrow\ D \mid r \Rightarrow\ E\rangle \\ & \mid r \Rightarrow\ \langle \text{case } C \text{ of } \ell \Rightarrow\ D \mid r \Rightarrow\ E\rangle\rangle\end{aligned}$$

Figure A.6: Operational semantics: commuting conversion rules.

$$\text{(unrollC)}\quad \begin{array}{l}[\text{unroll } \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle] \\ \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ [\text{unroll } B] \mid r \Rightarrow\ [\text{unroll } C]\rangle\end{array}$$

$$\text{(repC)}\quad \begin{array}{l}(\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \text{ rep } p : \psi.\ N \text{ in } O) \\ \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ (B \text{ rep } p : \psi.\ N \text{ in } O) \mid r \Rightarrow\ (C \text{ rep } p : \psi.\ N \text{ in } O)\rangle\end{array}$$

$$\text{(forC)}\quad \begin{array}{l}\text{for}(p : \varphi(\mathcal{M}) = \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle; q; N)\,\{\alpha\}\,O \\ \mapsto \langle \text{case } A \text{ of} \\ \quad \ell \Rightarrow\ \text{for}(p : \varphi(\mathcal{M}) = B; q; N)\,\{\alpha\}\,O \\ \quad \mid r \Rightarrow\ \text{for}(p : \varphi(\mathcal{M}) = C; q; N)\,\{\alpha\}\,O\rangle\end{array}$$

$$\text{(FPC)}\quad \begin{array}{l}FP(\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle, s.\ D, g.\ E) \\ \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ FP(B, s.\ D, g.\ E) \mid r \Rightarrow\ FP(C, s.\ D, g.\ E)\rangle\end{array}$$

$$\text{(}\langle\iota\rangle\text{C)}\quad \langle \iota\ \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rangle \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \langle \iota\ B\rangle \mid r \Rightarrow\ \langle \iota\ C\rangle\rangle$$

$$\text{([}\iota\text{]C)}\quad [\iota\ \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle] \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ [\iota\ B] \mid r \Rightarrow\ [\iota\ C]\rangle$$

$$\text{(}\langle^d\rangle\text{C)}\quad \begin{array}{l}\langle \text{yield } \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rangle \\ \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \langle \text{yield } B\rangle \mid r \Rightarrow\ \langle \text{yield } C\rangle\rangle\end{array}$$

$$\text{([}^d\text{]C)}\quad \begin{array}{l}[\text{yield } \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle] \\ \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ [\text{yield } B] \mid r \Rightarrow\ [\text{yield } C]\rangle\end{array}$$

$$\text{(app1C)}\quad \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\ N \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ B\ N \mid r \Rightarrow\ C\ N\rangle$$

$$\text{(app2C)}\quad M\ \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ M\ B \mid r \Rightarrow\ M\ C\rangle$$

$$\text{(appC)}\quad \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\ f \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ B\ f \mid r \Rightarrow\ C\ f\rangle$$

$$\text{(}\circ\text{C)}\quad \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \circ_p N \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ (B \circ_p N) \mid r \Rightarrow\ (C \circ_p N)\rangle$$

$$\text{(}\langle{:}*\rangle\text{C)}\quad \begin{array}{l}\text{unpack}(\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle, py.\ N) \\ \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \text{unpack}(B, py.\ N) \mid r \Rightarrow\ \text{unpack}(C, py.\ N)\rangle\end{array}$$

Figure A.7: Operational semantics: commuting conversion rules (contd.)

$$(\langle\!\langle \pi_L \rangle\!\rangle \mathrm{S}) \quad \frac{M \mapsto M'}{\langle\!\langle \pi_L M \rangle\!\rangle \mapsto \langle\!\langle \pi_L M' \rangle\!\rangle} \qquad (\circ \mathrm{S}) \quad \frac{M \mapsto M'}{M \circ_p N \mapsto M' \circ_p N}$$

$$(\langle\!\langle \pi_R \rangle\!\rangle \mathrm{S}) \quad \frac{M \mapsto M'}{\langle\!\langle \pi_R M \rangle\!\rangle \mapsto \langle\!\langle \pi_R M' \rangle\!\rangle} \qquad (\langle\!\langle \ell \cdot \rangle\!\rangle \mathrm{S}) \quad \frac{M \mapsto M'}{\langle\!\langle \ell \cdot M \rangle\!\rangle \mapsto \langle\!\langle \ell \cdot M' \rangle\!\rangle}$$

$$(\mathrm{repS}) \quad \frac{M \mapsto M'}{(M \text{ rep } p : \psi.\ N \text{ in } O) \mapsto (M' \text{ rep } p : \psi.\ N \text{ in } O)} \qquad (\langle\!\langle r \cdot \rangle\!\rangle \mathrm{S}) \quad \frac{M \mapsto M'}{\langle\!\langle r \cdot M \rangle\!\rangle \mapsto \langle\!\langle r \cdot M' \rangle\!\rangle}$$

$$(\mathrm{unroll}) \quad \frac{M \mapsto M'}{[\text{unroll } M] \mapsto [\text{unroll } M']} \qquad ([\cup]\mathrm{S1}) \quad \frac{M \mapsto M'}{[M, N] \mapsto [M', N]}$$

$$([:*]\mathrm{S}) \quad \frac{M \mapsto M'}{M\ f \mapsto M'\ f} \qquad (\langle\cup\rangle\mathrm{S1}) \quad \frac{M \mapsto M'}{\langle M, N \rangle \mapsto \langle M', N \rangle}$$

$$([\iota]\mathrm{S}) \quad \frac{M \mapsto M'}{[\iota\ M] \mapsto [\iota\ M']} \qquad ([\cup]\mathrm{S2}) \quad \frac{M \text{ normal} \quad N \mapsto N'}{[M, N] \mapsto [M, N']}$$

$$(\langle\iota\rangle\mathrm{S}) \quad \frac{M \mapsto M'}{\langle \iota\ M \rangle \mapsto \langle \iota\ M' \rangle} \qquad (\langle\cup\rangle\mathrm{S2}) \quad \frac{M \text{ normal} \quad N \mapsto N'}{\langle M, N \rangle \mapsto \langle M, N' \rangle}$$

$$([^d]\mathrm{S}) \quad \frac{M \mapsto M'}{[\text{yield } M] \mapsto [\text{yield } M']} \qquad (\mathrm{appS1}) \quad \frac{M \mapsto M'}{M\ N \mapsto M'\ N}$$

$$(\langle^d\rangle\mathrm{S}) \quad \frac{M \mapsto M'}{\langle \text{yield } M \rangle \mapsto \langle \text{yield } M' \rangle} \qquad (\mathrm{appS2}) \quad \frac{M \text{ normal} \quad N \mapsto N'}{M\ N \mapsto M\ N'}$$

$$(\mathrm{forS}) \quad \frac{A \mapsto A'}{\text{for}(p : \varphi(\mathcal{M}) = A; q; B)\ \{\alpha\}\ C \mapsto \text{for}(p : \varphi(\mathcal{M}) = A; q; B)\ \{\alpha\}\ C}$$

$$(\mathrm{FPS}) \quad \frac{A \mapsto A'}{FP(A, s.\ B, g.\ C) \mapsto FP(A', s.\ B, g.\ C)}$$

$$(\mathrm{caseS}) \quad \frac{A \mapsto A'}{\langle\!\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C \rangle\!\rangle \mapsto \langle\!\langle \text{case } A' \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C \rangle\!\rangle}$$

$$(\langle :* \rangle \mathrm{S}) \quad \frac{M \mapsto M'}{\text{unpack}(M, py.\ N) \mapsto \text{unpack}(M', py.\ N)}$$

Figure A.8: Operational semantics: structural rules.

A.3 Example Proofs

Having developed a proof calculus, we are now equipped to verify the example games of Section 4.5. We give proof terms for each player's winning region for Nim in Fig. A.9, likewise for cake-cutting in Fig. A.10. While practical Kaisar proofs (Chapter 7) are much higher-level, the natural deduction proofs in this case are already surprisingly simple. Moreover, the examples serve to demonstrate the typical usage of each construct and to confirm that the rules are applicable to typical practical examples. By convention, in the examples, we assume the proof variable for the fact introduced by assigning to a variable named x is itself also named x.

Proof terms aNim and dNim in Fig. A.9 both give examples of loop proofs, the former being a convergence proof and the latter being an invariant proof.

In aNim, the invariant is $c \bmod 4 \neq 0$, i.e., $(c \bmod 4 = 0) \to \mathit{false}$, so the invariant does not tell Angel which value $c \bmod 4$ assumes, only that it would be a contradiction to take value 0. The inductive step is step, which shows the body preserves the invariant while the metric decreases. Thus, Angel must inspect the state using the law of excluded middle (which is constructive for equality tests on rational numbers) and the "case" keyword. For each case Angel picks the branch that ensures $c \bmod 4 = 0$, using $\langle \ell \cdot \cdot \rangle$ and $\langle r \cdot \cdot \rangle$ to specify which branch. In each branch, an intermediate condition Mid is proved to follow from the case assumptions by first-order reasoning (FO[Mid](...)). Proof terms such as $\langle c := f_{\overline{c}} \text{ in } \cdot \rangle$ are used to introduce assignments and the proof term constructor $\langle \iota \cdot \rangle$ introduces sequential compositions. After building a "case" proof term which proves the intermediate condition on each branch, the rule M is used to show that the invariant follows from the intermediate condition. The use of rule M reduces the number of cases from 9 total to 3 for Angel and 3 for Demon: Under the assumption (named m) that $c > 0 \wedge c \bmod 4 = 0$, then all possible Demon choices restore the invariant, which we prove by FO (corr. automation in a machine-checked proof). In the proof of the invariant, lambda expressions prove tests, e.g., $(\lambda \mathsf{test} : c > 0. \cdot)$ proves the Demonic test $?c > 0$. Angelic tests are proved by pairs where the components respectively prove the test condition and postcondition.

The proof term dNim proves a Demonic repetition formula, so it proves an invariant argument but need not show that any termination metric decreases. The strategy for the body is similar to that of aNim in the sense that it is a mirroring strategy. A notable difference is that the monotonicity step of dNim uses Demonic reasoning in the proof of its first premise and Angelic reasoning in the second, which is opposite of aNim because opposite players move first in each game.

The intermediate condition $(c \bmod 4 = 2 \vee c \bmod 4 = 3 \vee c \bmod 4 = 0)$ uses a constructive disjunction to represent the fact that Demon explicitly tells Angel which branch he took, which is then used to choose Angel's branch. If we knew $c \bmod 4 \in \mathbb{N}$, an alternate proof approach would be to write $c \bmod 4 \neq 1$ in the intermediate condition, forgetting Demon's announcement of the branch taken. Because rational comparison is decidable, Angel could perform her own case analysis to decide which branch *she* should take. Natural numbers are not built in to CGL but are definable using the same technique used for dL (Platzer, 2008a), so the alternate approach is feasible in principle. Our chosen approach is simpler because it does not require defining natural numbers, rather we have discussed the alternate proof

approach as a way of reflecting on the additional strategic information represented by our disjunctive formula vs. a single disequality.

In aCake, Angel splits the cake evenly, which is her optimal strategy. Even assuming Demon is entitled to nonconstructive strategies, whatever slice he picks must be one of the two, both of which have size 0.5. In dCake, Demon splits the cake according to an arbitrary strategy, then Angel inspects the value of x to decide which slice to take. If $x \geq 0.5$ then Angel takes x, else Angel takes y. Angel has an optimal strategy because rationals can be compared exactly. If we operated over constructive reals, Angel would not have exact optimality because reals could not be compared exactly. She would, however, be allowed to get arbitrarily close to optimal.

$$
\begin{array}{l}
\mathsf{Mid} \equiv c > 0 \wedge c \bmod 4 = 1 \quad \mathcal{M} \equiv c - 1 \operatorname{div} 4 \quad \mathbf{0} \equiv 0 \quad \varphi \equiv (c \bmod 4 \in \{0,2,3\}) \\
\mathsf{aNim} \equiv \lambda nz : c > 0.\ \lambda mod_0 : (c \bmod 4 \in \{0,2,3\}). \\
\qquad \mathrm{for}(cnv : \varphi(\mathcal{M}) = \langle mod_0, nz\rangle; mod; \mathsf{step})\{\textsc{Nim}\}(\mathsf{FO}[c \in \{2,3,4\}](nz, mod)) \\
\mathsf{step} \equiv \langle \iota\ \langle \iota\ \langle \text{case } \mathsf{Dec}[(c \bmod 4 = 3)]() \text{ of} \\
\qquad \ell_1 \Rightarrow\ \langle r \cdot \langle \ell \cdot \langle c := f_{\bar{c}} \text{ in } \ \mathsf{FO}[\mathsf{Mid}](cnv, mod, \ell_1, c)\rangle\rangle\rangle \\
\qquad \mid r_1 \Rightarrow\ \langle \text{case } \mathsf{Dec}[(c \bmod 4 = 2)]() \text{ of} \\
\qquad\quad \ell_2 \Rightarrow\ \langle \ell \cdot \langle c := f_{\bar{c}} \text{ in } \ \mathsf{FO}[\mathsf{Mid}](cnv, mod, r_1, \ell_2, c)\rangle\rangle \\
\qquad\quad \mid r_2 \Rightarrow\ \langle r \cdot \langle r \cdot \langle c := f_{\bar{c}} \text{ in } \ \mathsf{FO}[\mathsf{Mid}](cnv, mod, r_1, r_2, c)\rangle\rangle\rangle\rangle\rangle \\
\quad {}^{\circ}\mathsf{m:Mid} \\
\quad \langle \mathsf{FO}[c > 0](\mathsf{m}), \langle \text{yield } [\iota \\
\quad [[c := f_{\bar{c}} \text{ in } \ (\lambda \mathsf{test} : c > 0.\ \mathsf{FO}[\varphi \wedge \mathcal{M}_0 \succ \mathcal{M}](\mathsf{m}, c))] \\
\quad , [[c := f_{\bar{c}} \text{ in } \ (\lambda \mathsf{test} : c > 0.\ \mathsf{FO}[\varphi \wedge \mathcal{M}_0 \succ \mathcal{M}](\mathsf{m}, c))] \\
\quad , [c := f_{\bar{c}} \text{ in } \ (\lambda \mathsf{test} : c > 0.\ \mathsf{FO}[\varphi \wedge \mathcal{M}_0 \succ \mathcal{M}](\mathsf{m}, c))]]]\rangle\rangle\rangle\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathsf{Mid} \equiv c \geq 0 \wedge (c \bmod 4 = 2 \vee c \bmod 4 = 3 \vee c \bmod 4 = 0) \\
\varphi \equiv (c > 0 \wedge c \bmod 4 = 1) \\
\mathsf{dNim} \equiv \lambda nz : c > 0.\ \lambda mod_0 : (c \bmod 4 = 1). \\
\qquad \langle nz, mod\rangle \text{ rep } cnv : \varphi.\ \mathsf{step} \text{ in } cnv \\
\mathsf{step} \equiv [\iota\ ([\iota \\
\qquad [[c := f_{\bar{c}} \text{ in } \ (\lambda \mathsf{test} : c > 0.\ \mathsf{FO}[\mathsf{Mid}](cnv, c, \mathsf{test}))] \\
\qquad , [[c := f_{\bar{c}} \text{ in } \ (\lambda \mathsf{test} : c > 0.\ \mathsf{FO}[\mathsf{Mid}](cnv, c, \mathsf{test}))] \\
\qquad , [c := f_{\bar{c}} \text{ in } \ (\lambda \mathsf{test} : c > 0.\ \mathsf{FO}[\mathsf{Mid}](cnv, c, \mathsf{test}))]]]) \\
\quad {}^{\circ}\mathsf{m:Mid} \\
\quad \langle \text{yield } \langle \text{case } \mathsf{m} \text{ of} \\
\qquad \ell_1 \Rightarrow\ \langle \ell \cdot \langle c := f_{\bar{c}} \text{ in } \ \mathsf{FO}[\varphi](\mathsf{m}, \ell_1, c)\rangle\rangle \\
\qquad \mid r_1 \Rightarrow\ \langle \text{case } r_1 \text{ of} \\
\qquad\quad \ell_2 \Rightarrow\ \langle r \cdot \langle \ell \cdot \langle c := f_{\bar{c}} \text{ in } \ \mathsf{QE}_1\rangle\rangle\rangle \\
\qquad\quad \mid r_2 \Rightarrow\ \langle r \cdot \langle r \cdot \langle c := f_{\bar{c}} \text{ in } \ \mathsf{QE}_2\rangle\rangle\rangle\rangle\rangle\rangle \\
\mathsf{QE}_1 \equiv \mathsf{FO}[\varphi](\mathsf{m}, r_1, \ell_2, c) \\
\mathsf{QE}_2 \equiv \mathsf{FO}[\varphi](\mathsf{m}, r_1, r_2, c)
\end{array}
$$

Figure A.9: Proof terms for Nim example.

$$
\begin{aligned}
&\mathsf{Mid} \equiv x = 0.5 \wedge y = 0.5 \\
&\mathsf{aCake} \equiv \quad \langle 0.5_{\overline{x}} :* \langle \mathsf{FO}[0 < x < 1](), \langle y := f_{\overline{y}} \text{ in } \ \langle \mathsf{FO}[\mathsf{Mid}](x,y), \mathsf{FO}[\mathsf{Mid}](x,y)\rangle\rangle\rangle\rangle \\
&\qquad \circ_{\mathsf{m}:\mathsf{Mid}} \\
&\qquad\quad \langle \text{yield } [[\text{yield } \langle \iota \ \langle a := f_{\overline{a}} \text{ in } \ \langle d := f_{\overline{d}} \text{ in } \ \mathsf{FO}[a \geq 0.5](a,x)\rangle\rangle\rangle] \\
&\qquad\qquad , [\text{yield } \langle \iota \ \langle a := f_{\overline{a}} \text{ in } \ \langle d := f_{\overline{d}} \text{ in } \ \mathsf{FO}[a \geq 0.5](a,x)\rangle\rangle\rangle]]\rangle \\
&\mathsf{dCake} \equiv \lambda x : \mathbb{Q}.\ \lambda \mathsf{Range} : (0 \leq x \leq 1). \\
&\qquad\quad ([y := f_{\overline{y}} \text{ in } \ \mathsf{FO}[x + y = 1](\mathsf{Range}, y)]) \\
&\qquad \circ_{\mathsf{m}:(x+y=1)} [\text{yield} \\
&\qquad\quad \langle \text{case (split } [x \sim 0.5]) \text{ of} \\
&\qquad\qquad p \Rightarrow \ \ell \cdot \langle \text{yield } [a := f_{\overline{a}} \text{ in } \ [d := f_{\overline{d}} \text{ in } \ \mathsf{FO}[d \geq 0.5](p,d,m,\mathsf{Range})]]\rangle \\
&\qquad\quad \mid q \Rightarrow \ r \cdot \langle \text{yield } [a := f_{\overline{a}} \text{ in } \ [d := f_{\overline{d}} \text{ in } \ \mathsf{FO}[d \geq 0.5](q,d)]]\rangle\rangle]
\end{aligned}
$$

Figure A.10: Proof terms for cake-cutting example.

A.4 Theory Proofs

We give proofs regarding semantics, soundness, and progress and preservation. We first give a preliminary result on effective well-foundedness.

A.4.1 Preliminaries

Lemma 4.5 (Effectively Well-Founded Implies Effectively Descending)**.** *Let $\succ$ be an effectively well-founded relation on some set X. Then $\succ$ satisfies the effective descending chain condition on X.*

Proof. Fix a set X and an effectively well-founded ordering $\succ$ on X. The induction principle on effectively-well founded (Hofmann et al., 2006) orderings says for every set $Y \subseteq X$ that in order to conclude $X = Y$, it suffices to prove that

$$\forall x\, (\forall y\ \ (x \succ y \to y \in Y) \to x \in Y)$$

The main proof is by induction, so we start by defining our choice of set Y. Let $\mathsf{Seq}(S)$ denote the set of sequences drawn from S (for any set S). We index sequences starting from zero, written S_i for the ith element. Let $\mathsf{DecSeq}(v, X)$ denote the set of decreasing sequences drawn from X with initial element v. Let $\mathit{Dom}(S) \subseteq \mathbb{N}$ be the domain of sequence S. As is standard in constructive mathematics, sequences must have decidable domain membership tests, so that domain membership (for a given index i) admits the law of the excluded middle. Formally, we define

$$\mathsf{DecSeq}(v, X) \equiv \{S \in \mathsf{Seq}(X) \mid S_0 = v \text{ and } (\forall i\, j \in \mathit{Dom}(S)\ i < j \to S_i \succ S_j))\}$$

meaning that for any pair of indices $i < j$ in the domain of the sequence, the value at location i is always greater than that at location j. Let $\mathsf{FiniteSeq}$ be the set of finite sequences of elements of X. Formally, we define

$$\mathsf{FiniteSeq} \equiv \{S \in \mathsf{Seq}(X) \mid \exists i : \mathbb{N}\ i \notin \mathit{Dom}(S)\}$$

meaning there exists a natural number outside its domain[1].

To show the effective descending chain condition, we prove by induction on the initial value that all nonempty descending sequences are finite, that is, we define $Y = \{v \in X \mid \mathsf{DecSeq}(v, X) \subseteq \mathsf{FiniteSeq}\}$. The induction will show $Y = X$. Because the inclusion $Y \subseteq X$ holds by construction, it suffices to show the inductive step.

To prove the inductive step, fix some $x \in X$ and assume for all $y \in X$ that if $x \succ y$ then $y \in Y$, which is the inductive hypothesis. Then show $x \in Y$. To show $x \in Y$, fix arbitrary $S \in \mathsf{DecSeq}(x, X)$. Because S is a sequence, either $\{0, 1\} \subseteq \mathit{Dom}(S)$ or not, constructively. In the latter case, $S \in \mathsf{FiniteSeq}$ by letting $i = 0$ or $i = 1$ to witness $\exists i : \mathbb{N}\ i \notin \mathit{Dom}(S)$. In the former case, there exist elements S_0 and S_1 and a sequence $\hat{S}$ such that $S = S_0, S_1, \hat{S}$. Because $S \in \mathsf{DecSeq}(v, X)$ we have $S_0 = x$ and $S_0 \succ S_1$ and $S_1, \hat{S} \in \mathsf{DecSeq}(S_1, X)$. The

[1]Because S is a sequence, it is implied that all $j > i$ are also outside its domain.

IH applies on $S_1, \hat{S}$ because $S_0 \succ S_1$, resulting in $S_1, \hat{S} \in \mathsf{FiniteSeq}$. The inductive case holds by concluding $S_0, S_1, \hat{S} \in \mathsf{FiniteSeq}$ because adding one element to a finite sequence results in a finite sequence. Formally, the inductive hypothesis yields $i \in \mathbb{N}$ such that $i \notin Dom(S_1, \hat{S})$, then $i+1 \notin Dom(S_0, S_1, \hat{S})$ witnesses finiteness of S as desired.

We now have $X = Y$, from which we prove every descending sequence is finite. Let S be any descending sequence of elements of X under $\succ$. Either $0 \in Dom(S)$ or not. If not, S is the empty sequence, which is finite. Else, the element S_0 exists and $S \in \mathsf{DecSeq}(S_0, X)$ by construction of $\mathsf{DecSeq}(S_0, X)$. Since $S_0 \in X$ by assumption then by $X = Y$ we have $\mathsf{DecSeq}(S_0, X) \subseteq \mathsf{FiniteSeq}$ so that $S \in \mathsf{FiniteSeq}$ by transitivity, as desired. □

A.4.2 Static Semantics

We repeat (Fig. A.11) the definitions of free variable $\mathrm{FV}(e)$, bound variable $\mathrm{BV}(\alpha)$, and must-bound variable $\mathrm{MBV}(\alpha)$ computations for formulas and games which will be used throughout the proofs. Free variables are those which might influence the expression. Bound variables are those which could be modified during a game, and must-bound variables are those which are modified on every path of a game. The free variables of a term are all the variables mentioned in the term. The definitions are standard (Platzer, 2018b). These definitions also apply to CdGL as used in Chapter 5 and Chapter 6, though the bound and must-bound variable computations in CdGL have additional cases for systems of ODEs, which are present in CdGL but not CGL. In contrast to CGL, CdGL treats terms semantically, not syntactically, thus the free variables of terms (e.g., $\mathrm{FV}(f)$) should be read as semantic free variables when reading this definition in the context of CdGL.

A.4.3 Realizers

Realizers $\mathsf{b}, \mathsf{c}, \mathsf{d}$ (variable name rz is occasionally used as well) encode strategies for playing a game or (equivalently) witnessing a formula. Because different components of a strategy must be evaluated in different states, execution of realizers proceeds lazily: execution of a single game operator may evaluate one component of a realizer in the current state, then return an unevaluated realizer to be evaluated in another yet-unknown state.

While reasoning about lazy semantics is technically subtle, it is often the case that local reasoning steps only need to consider components of a realizer which are locally being forced to a value. For that reason, our lemmas about realizers will employ an eager *forcing* semantics of realizers: $[\![\mathsf{b}]\!]\omega$ is the realizer value that results from replacing the free variables x of b with $\omega(x)$ and evaluating. The gap between eager and lazy semantics is bridged wherever realizer lemmas are applied by locally applying lemmas when the CGL semantics do force evaluation of a realizer.

A.4.4 Repetition

In classical (and constructive) GL, the challenge in defining the semantics of repetition games α^* is that the number of iterations, while finite, can depend on both players' actions and is thus not known in advance, while a DL-like semantics of α^*, such as defining the

$$
\begin{array}{rl}
\mathrm{FV}(f \sim g) & = \mathrm{FV}(f) \cup \mathrm{FV}(g) \\
\mathrm{FV}(\langle\alpha\rangle\phi) & = \mathrm{FV}(\alpha) \cup (\mathrm{FV}(\phi) \setminus \mathrm{MBV}(\alpha)) \\
\mathrm{FV}([\alpha]\phi) & = \mathrm{FV}(\alpha) \cup (\mathrm{FV}(\phi) \setminus \mathrm{MBV}(\alpha)) \\
\mathrm{FV}(?\phi) & = \mathrm{FV}(\phi) \\
\mathrm{FV}(x := f) & = \mathrm{FV}(f) \\
\mathrm{FV}(x := *) & = \emptyset \\
\mathrm{FV}(\alpha;\beta) & = \mathrm{FV}(\alpha) \cup (\mathrm{FV}(\beta) \setminus \mathrm{MBV}(\alpha)) \\
\mathrm{FV}(\alpha \cup \beta) & = \mathrm{FV}(\alpha) \cup \mathrm{FV}(\beta) \\
\mathrm{FV}(\alpha^*) & = \mathrm{FV}(\alpha) \\
\mathrm{FV}(\alpha^d) & = \mathrm{FV}(\alpha) \\
\hline
\mathrm{BV}(?\phi) & = \emptyset \\
\mathrm{BV}(x := f) & = \{x\} \\
\mathrm{BV}(x := *) & = \{x\} \\
\mathrm{BV}(\alpha;\beta) & = \mathrm{BV}(\alpha) \cup \mathrm{BV}(\beta) \\
\mathrm{BV}(\alpha \cup \beta) & = \mathrm{BV}(\alpha) \cup \mathrm{BV}(\beta) \\
\mathrm{BV}(\alpha^*) & = \mathrm{BV}(\alpha) \\
\mathrm{BV}(\alpha^d) & = \mathrm{BV}(\alpha) \\
\hline
\mathrm{MBV}(?\phi) & = \emptyset \\
\mathrm{MBV}(x := f) & = \{x\} \\
\mathrm{MBV}(x := *) & = \{x\} \\
\mathrm{MBV}(\alpha;\beta) & = \mathrm{MBV}(\alpha) \cup \mathrm{MBV}(\beta) \\
\mathrm{MBV}(\alpha \cup \beta) & = \mathrm{MBV}(\alpha) \cap \mathrm{MBV}(\beta) \\
\mathrm{MBV}(\alpha^*) & = \emptyset \\
\mathrm{MBV}(\alpha^d) & = \mathrm{MBV}(\alpha)
\end{array}
$$

Figure A.11: CGL static semantics.

loop α^* as the union of all its finite iterations, would give an advance-notice semantics. Classical GL provides the no-advance-notice semantics as a fixed point (Parikh, 1983), and we adopt a fixed-point semantics as well. In CGL, we also give an iterative semantics which agrees with the fixed-point semantics, which notably means playing our iterative semantics is *not* equivalent to playing the game which chooses between all finite repetitions of the body at the start of gameplay.

Our iterative semantics is reminiscent of an inflationary semantics for dGL (Platzer, 2015a), but has key differences. The dGL semantics works backwards to determine an *initial* winning region as a function of a *final* goal region. The dGL semantics has a large closure ordinal, meaning that there exist games where it only converges after a large infinite ordinal number of iterations. Our semantics works forward: starting with both initial states *and* realizers which determine the strategy to be played in each state, it determines the final region. In stark contrast, our forward-chaining semantics is Scott-continuous, thus has a low closure ordinal, at most ω. Our fixed-point construction thus converges in at most ω steps and our iterative semantics can be characterized by a countable union rather than transfinite induction over large ordinals as is necessary in the corresponding dGL

semantics (Platzer, 2015a).

While the main topic of this subsection is the discussion of repetition, our results for repetition rely on Scott-continuity which depends on monotonicity of the CGL semantics. For that reason, we present the statements and proofs of monotonicity and Scott-continuity here. The monotonicity lemma is distinct from the monotonicity *rule* of CGL. The monotonicity *rule* is discussed in Appendix A.4.5.

Lemma 4.6 (Monotonicity)**.** *Assume $X \subseteq Y$ and let ϕ be an arbitrary CGL formula. In each respective claim, let $\mathsf{b} \in \langle\alpha\rangle\phi\,\mathcal{R}\mathbf{z}$ or $\mathsf{b} \in [\alpha]\phi\,\mathcal{R}\mathbf{z}$ for every realizer b in Y so that the realizers in Y have the shapes expected by the semantic functions, e.g., realizers of Angelic tests are pairs.*

- $X\langle\!\langle\alpha\rangle\!\rangle \subseteq Y\langle\!\langle\alpha\rangle\!\rangle$
- $X[\![\alpha]\!] \subseteq Y[\![\alpha]\!]$

Proof. By simultaneous induction on α for the Angel and Demon claims.

Angel cases:

Case $x := f$: We have that $X\langle\!\langle x := f\rangle\!\rangle = \{(\mathsf{b}, \omega[x \mapsto [\![f]\!]\omega]) \mid (\mathsf{b},\omega) \in X\} \subseteq \{(\mathsf{b}, \omega[x \mapsto [\![f]\!]\omega]) \mid (\mathsf{b},\omega) \in Y\} = Y\langle\!\langle x := f\rangle\!\rangle$.

Case $x := *$: Have $X\langle\!\langle x := *\rangle\!\rangle = \{(\pi_1\mathsf{b}, \omega[x \mapsto \pi_0\mathsf{b}\,\omega]) \mid (\mathsf{b},\omega) \in X\} \subseteq \{(\pi_1\mathsf{b}, \omega[x \mapsto \pi_0\mathsf{b}\,\omega]) \mid (\mathsf{b},\omega) \in Y\} = Y\langle\!\langle x := *\rangle\!\rangle$.

Case $?\phi$: Have

$$\begin{aligned} X\langle\!\langle ?\phi\rangle\!\rangle &= \{(\pi_1\mathsf{b},\omega) \mid (\mathsf{b},\omega) \in X \text{ and } (\pi_0\mathsf{b},\omega) \in [\![\phi]\!]\} \\ &\subseteq \{(\pi_1\mathsf{b},\omega) \mid (\mathsf{b},\omega) \in Y \text{ and } (\pi_0\mathsf{b},\omega) \in [\![\phi]\!]\} = Y\langle\!\langle ?\phi\rangle\!\rangle \end{aligned}$$

Case $\alpha \cup \beta$: Have $X\langle\!\langle \alpha \cup \beta\rangle\!\rangle = X_{\langle 0\rangle}\langle\!\langle\alpha\rangle\!\rangle \cup X_{\langle 1\rangle}\langle\!\langle\beta\rangle\!\rangle \subseteq Y_{\langle 0\rangle}\langle\!\langle\alpha\rangle\!\rangle \cup Y_{\langle 1\rangle}\langle\!\langle\beta\rangle\!\rangle = Y\langle\!\langle \alpha \cup \beta\rangle\!\rangle$.

Case $\alpha;\beta$: Have $X\langle\!\langle \alpha;\beta\rangle\!\rangle = X\langle\!\langle\alpha\rangle\!\rangle\langle\!\langle\beta\rangle\!\rangle \subseteq Y\langle\!\langle\alpha\rangle\!\rangle\langle\!\langle\beta\rangle\!\rangle = Y\langle\!\langle \alpha;\beta\rangle\!\rangle$ since by the second IH have $X\langle\!\langle\beta\rangle\!\rangle \subseteq Y\langle\!\langle\beta\rangle\!\rangle$.

Case α^*: Have $X\langle\!\langle\alpha^*\rangle\!\rangle = \bigcap\{Z_{\langle 0\rangle} \mid X \cup Z_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle \subseteq Z\} \subseteq_{(*)} \bigcap\{Z_{\langle 0\rangle} \mid Y \cup Z_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle \subseteq Z\} = Y\langle\!\langle\alpha^*\rangle\!\rangle$. To show the starred step, consider any set Z in the fixed-point construction. Since we assumed $X \subseteq Y$, then $X \cup Z_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle \subseteq Y \cup Z_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle$ for each Z, i.e., each term of the intersection. The starred step then holds because an intersection of subsets is the subset of the intersection of the supersets.

Case α^d: Have $X\langle\!\langle\alpha^d\rangle\!\rangle = X[\![\alpha]\!] \subseteq Y[\![\alpha]\!] = Y\langle\!\langle\alpha^d\rangle\!\rangle$.

Demon cases:

Case $x := f$: We have that $X[\![x := f]\!] = \{(\mathsf{b}, \omega[x \mapsto [\![f]\!]\omega]) \mid (\mathsf{b},\omega) \in X\} \subseteq \{(\mathsf{b}, \omega[x \mapsto [\![f]\!]\omega]) \mid (\mathsf{b},\omega) \in Y\} = Y[\![x := f]\!]$.

Case $x := *$: Have

$$\begin{aligned} & X[\![x := *]\!] \\ &= \{(\mathsf{b}\,v, \omega[x \mapsto v]) \mid (\mathsf{b},\omega) \in X, \text{ for some } v \in \mathbb{Q}\} \\ &\subseteq \{(\mathsf{b}\,v, \omega[x \mapsto v]) \mid (\mathsf{b},\omega) \in Y, \text{ for some } v \in \mathbb{Q}\} \\ &= Y[\![x := *]\!] \end{aligned}$$

Case $?\phi$: Have $X[\![?\phi]\!] = \{(\mathsf{b}\,\mathsf{c}, \omega) \mid (\mathsf{b},\omega) \in X, (\mathsf{c},\omega) \in [\![\phi]\!]\} \subseteq \{(\mathsf{b}\,\mathsf{c}, \omega) \mid (\mathsf{b},\omega) \in Y, (\mathsf{c},\omega) \in [\![\phi]\!]\} = Y[\![?\phi]\!]$.

Case $\alpha \cup \beta$: Have $X[\![\alpha \cup \beta]\!] = X_{[0]}[\![\alpha]\!] \cup X_{[1]}[\![\beta]\!] = Y_{[0]}[\![\alpha]\!] \cup Y_{[1]}[\![\beta]\!] = Y[\![\alpha \cup \beta]\!]$.

Case $\alpha;\beta$: Have $X[\![\alpha;\beta]\!] = X[\![\alpha]\!][\![\beta]\!] \subseteq Y[\![\alpha]\!][\![\beta]\!] = Y[\![\alpha;\beta]\!]$, since by the second IH $X[\![\beta]\!] \subseteq Y[\![\beta]\!]$.

Case α^*: Have $X[\![\alpha^*]\!] = \bigcap\{Z_{[0]} \mid X \cup Z_{[1]}[\![\alpha]\!] \subseteq Z\} \subseteq_{(*)} \bigcap\{Z_{[0]} \mid Y \cup Z_{[1]}[\![\alpha]\!] \subseteq Z\} = Y[\![\alpha^*]\!]$. To show the starred step, consider any set Z in the fixed-point construction. Since we assumed $X \subseteq Y$, then $X \cup Z_{[1]}[\![\alpha]\!] \subseteq Y \cup Z_{[1]}[\![\alpha]\!]$ for each Z, i.e., each term of the intersection. The starred step then holds because an intersection of subsets is the subset of the intersection of the supersets.

Case α^d: Have $X[\![\alpha^d]\!] = X\langle\!\langle\alpha\rangle\!\rangle \subseteq Y\langle\!\langle\alpha\rangle\!\rangle = Y[\![\alpha^d]\!]$. □

Lemma 4.7 (Scott-continuity)**.** *Let α be a game and ϕ a formula. Let $\{X_i\}$ be a (non-empty) family of regions where i ranges over some index set J.*

In each claim and for all $i \in J$, respectively let $\mathsf{b} \in \langle\alpha\rangle\phi\,\mathcal{R}\mathbf{z}$ or $\mathsf{b} \in [\alpha]\phi\,\mathcal{R}\mathbf{z}$ for all realizers b in each X_i, to ensure the semantics of α are well-defined. Then

$$\bigcup_{i\in J}(X_i\langle\!\langle\alpha\rangle\!\rangle) = (\bigcup_{i\in J} X_i)\langle\!\langle\alpha\rangle\!\rangle$$
$$\bigcup_{i\in J}(X_i[\![\alpha]\!]) = (\bigcup_{i\in J} X_i)[\![\alpha]\!]$$

In addition, the Angelic and Demonic projection operators are all Scott-continuous.

Proof. Firstly, Scott-continuity off the Angelic and Demonic projection operators holds directly by expanding their definitions.

Next, we show Scott-continuity of the game semantics. In each claim, the set equality is shown by showing each set includes the other. The forward direction of each claim holds by Lemma 4.6 because for each $i \in J$ it is the case that $X_i \subseteq \bigcup_{i\in J} X_i$. We prove the converse directions of each claim by simultaneous induction on games for Angel and Demon. That is, the simultaneous induction proves

$$(\bigcup_{i\in J} X_i)\langle\!\langle\alpha\rangle\!\rangle \subseteq \bigcup_{i\in J}(X_i\langle\!\langle\alpha\rangle\!\rangle)$$

for Angel's claim and

$$(\bigcup_{i\in J} X_i)[\![\alpha]\!] \subseteq \bigcup_{i\in J}(X_i[\![\alpha]\!])$$

for Demon's claim. In the proof of each case, we can assume without loss of generality that $\{\top,\bot\} \cap \bigcup_{i\in J} X_i = \emptyset$ because when $\top$ or $\bot$ does belong to some X_i, it immediately belongs to both the left-hand and right-hand sides of the conclusion by definition of the game semantics, regardless of the structure of α. Our proof must still consider cases where $\top$ or $\bot$ appears in the *output* of the game semantics because some test fails. The cases for tests, assignments, and loops are proved pointwise (assume an element of the left-hand side and show it is an element of the right-hand side), so they handle $\top$ and $\bot$ as special cases. The cases for sequential composition, discrete choice, and duality are proved setwise so the case where the output set contains $\top$ or $\bot$ is handled uniformly without any special

case. The cases for loops also employ an inner induction on the fixed-point definition of the semantics of loops.

Angel cases:

Case $?\phi$:

Consider case $(\mathsf{b},\omega) \in \big(\bigcup_{i\in J} X_i\big)\langle\langle ?\phi\rangle\rangle$, then $\mathsf{b} = \pi_0\mathsf{c}$ for some c such that $(\pi_0\mathsf{c},\omega) \in [\![\phi]\!]$ and $(\mathsf{c},\omega) \in \big(\bigcup_{i\in J} X_i\big)$ then $\mathsf{b} = \pi_0\mathsf{c}$ for some $i \in J$ and realizer c where $\pi_0\mathsf{c},\omega) \in [\![\phi]\!]$ and $(\mathsf{c},\omega) \in X_i$ then for some $i \in J$ have $(\mathsf{b},\omega) \in X_i\langle\langle ?\phi\rangle\rangle$ then $(\mathsf{b},\omega) \in \bigcup_{i\in J}\big(X_i\langle\langle ?\phi\rangle\rangle\big)$.

Consider the case $\bot \in \big(\bigcup_{i\in J} X_i\big)\langle\langle ?\phi\rangle\rangle$, then $(\pi_0\mathsf{b},\omega) \notin [\![\phi]\!]$ for all $(\mathsf{b},\omega) \in \big(\bigcup_{i\in J} X_i\big)$, thus there exists $i \in J$ such that $(\pi_0\mathsf{b},\omega) \notin [\![\phi]\!]$ for all $(\mathsf{b},\omega) \in X_i$, so there exists $i \in J$ such that $\bot \in X_i\langle\langle ?\phi\rangle\rangle$ and thus $\bot \in \bigcup_{i\in J} X_i\langle\langle ?\phi\rangle\rangle$ as desired.

Under the assumption $\top \notin \big(\bigcup_{i\in J} X_i\big)$, the case $\top \in \big(\bigcup_{i\in J} X_i\big)\langle\langle ?\phi\rangle\rangle$ cannot occur. By the definition of the semantics of tests, the output is always a proper possibility or $\bot$.

Case $x := f$:

Assume $(\mathsf{b},\omega) \in \big(\bigcup_{i\in J} X_i\big)\langle\langle x := f\rangle\rangle$, then $\omega = \nu[x \mapsto [\![f]\!]\omega]$ for some $(\mathsf{c},\nu) \in \big(\bigcup_{i\in J} X_i\big)$, then for some $i \in J$ and $(\mathsf{c},\nu) \in X_i$, have $\omega = \nu[x \mapsto [\![f]\!]\omega]$, then for some $i \in J, (\mathsf{b},\omega) \in X_i\langle\langle x := f\rangle\rangle$, then $(\mathsf{b},\omega) \in \bigcup_{i\in J}\big(X_i\langle\langle x := f\rangle\rangle\big)$.

Under the assumption $\{\top,\bot\} \cap \big(\bigcup_{i\in J} X_i\big) = \emptyset$, the cases $\top \in \big(\bigcup_{i\in J} X_i\big)\langle\langle x := f\rangle\rangle$ and $\bot \in \big(\bigcup_{i\in J} X_i\big)\langle\langle x := f\rangle\rangle$ cannot occur. By the definition of the semantics of assignments, the output is always a proper possibility.

Case $x := *$:

Assume $(\mathsf{b},\omega) \in \big(\bigcup_{i\in J} X_i\big)\langle\langle x := *\rangle\rangle$, then $(\mathsf{b},\omega) = (\pi_1\mathsf{c}, \nu[x \mapsto [\![\pi_0\mathsf{c}]\!]\nu]$ for some $(\mathsf{c},\omega) \in \big(\bigcup_{i\in J} X_i\big)$, then for some $i \in J, (\mathsf{b},\omega) = (\pi_1\mathsf{c}, \nu[x \mapsto [\![\pi_0\mathsf{c}]\!]\nu]$ for some $(\mathsf{c},\omega) \in X_i$, then for some $i \in J, (\mathsf{b},\omega) \in X_i\langle\langle x := *\rangle\rangle$, then $(\mathsf{b},\omega) \in \bigcup_{i\in J}\big(X_i\langle\langle x := *\rangle\rangle\big)$.

Under the assumption $\{\top,\bot\} \cap \big(\bigcup_{i\in J} X_i\big) = \emptyset$, the cases $\top \in \big(\bigcup_{i\in J} X_i\big)\langle\langle x := *\rangle\rangle$ and $\bot \in \big(\bigcup_{i\in J} X_i\big)\langle\langle x := *\rangle\rangle$ cannot occur. By the definition of the semantics of non-deterministic assignments, the output is always a proper possibility.

Case $\alpha;\beta$:

Have $\big(\bigcup_{i\in J} X_i\big)\langle\langle \alpha;\beta\rangle\rangle, = \big(\big(\bigcup_{i\in J} X_i\big)\langle\langle\alpha\rangle\rangle\big)\langle\langle\beta\rangle\rangle \subseteq \big(\bigcup_{i\in J}\big(X_i\langle\langle\alpha\rangle\rangle\big)\big)\langle\langle\beta\rangle\rangle$ by the inductive hypothesis on α and by Lemma 4.6, then by the inductive hypothesis on β have $\big(\bigcup_{i\in J}\big(X_i\langle\langle\alpha\rangle\rangle\big)\big)\langle\langle\beta\rangle\rangle \subseteq \bigcup_{i\in J}\big(\big(X_i\langle\langle\alpha\rangle\rangle\big)\langle\langle\beta\rangle\rangle\big) = \bigcup_{i\in J}\big(X_i\langle\langle\alpha;\beta\rangle\rangle\big)$. As a result, the case holds by transitivity.

Case $\alpha \cup \beta$:

Have

$$\begin{aligned}
&\big(\bigcup_{i\in J} X_i\big)\langle\!\langle \alpha \cup \beta\rangle\!\rangle \\
&= \big(\bigcup_{i\in J} X_i\big)_{\langle 0\rangle}\langle\!\langle \alpha\rangle\!\rangle \cup \big(\bigcup_{i\in J} X_i\big)_{\langle 1\rangle}\langle\!\langle \beta\rangle\!\rangle \\
&\subseteq \big(\bigcup_{i\in J} (X_i)_{\langle 0\rangle}\big)\langle\!\langle \alpha\rangle\!\rangle \cup \big(\bigcup_{i\in J} (X_i)_{\langle 1\rangle}\big)\langle\!\langle \beta\rangle\!\rangle \\
&\subseteq \bigcup_{i\in J}\big(((X_i)_{\langle 0\rangle})\langle\!\langle \alpha\rangle\!\rangle\big) \cup \bigcup_{i\in J}\big(((X_i)_{\langle 1\rangle})\langle\!\langle \beta\rangle\!\rangle\big) \\
&= \bigcup_{i\in J}\big(((X_i)_{\langle 0\rangle})\langle\!\langle \alpha\rangle\!\rangle \cup ((X_i)_{\langle 1\rangle})\langle\!\langle \beta\rangle\!\rangle\big) \\
&= \bigcup_{i\in J}\big((X_i)\langle\!\langle \alpha \cup \beta\rangle\!\rangle\big)
\end{aligned}$$

where the first $\subseteq$ step holds by the Scott-continuity claim for projections and the second $\subseteq$ step holds by the IHs.

Case α^*:

Consider the case $(\mathsf{b},\omega) \in \big(\bigcup_{i\in J} X_i\big)\langle\!\langle \alpha^*\rangle\!\rangle$, then $(\mathsf{b},\omega) \in \bigcap\{Z_{\langle 0\rangle} \subseteq \mathsf{Poss} \mid \big(\bigcup_{i\in J} X_i\big) \cup (Z_{\langle 1\rangle}\langle\!\langle \alpha\rangle\!\rangle) \subseteq Z\}$. We now show $(\mathsf{b},\omega) \in \bigcup_{i\in J}\big(\bigcap\{Z_{\langle 0\rangle} \subseteq \mathsf{Poss} \mid X_i \cup (Z_{\langle 1\rangle}\langle\!\langle \alpha\rangle\!\rangle) \subseteq Z\}\big)$ by induction on construction of the fixed-point solution Z. In the base case $(\mathsf{b},\omega) \in \big(\bigcup_{i\in J} X_i\big)_{\langle 0\rangle}$ then Scott-continuity of $\cdot_{\langle 0\rangle}$ yields $(\mathsf{b},\omega) \in \bigcup_{i\in J}(X_i)_{\langle 0\rangle}$ so that the base case of the fixed point definition applies and $(\mathsf{b},\omega) \in \bigcup_{i\in J}\big(\bigcap\{Z_{\langle 0\rangle} \subseteq \mathsf{Poss} \mid \big(\bigcup_{i\in J} X_i\big) \cup (Z_{\langle 1\rangle}\langle\!\langle \alpha\rangle\!\rangle) \subseteq Z\}\big)$.

Now consider the inductive case $(\mathsf{b},\omega) \in (Z_{\langle 1\rangle}\langle\!\langle \alpha\rangle\!\rangle)_{\langle 0\rangle}$ for some Z. The inner IH says the claim of the inner induction holds for that Z, i.e., it says that for all $(\mathsf{c},\nu) \in Z$ we have $(\mathsf{c},\nu) \in \bigcup_{i\in J}\big(\bigcap\{Z_{\langle 0\rangle} \subseteq \mathsf{Poss} \mid \big(\bigcup_{i\in J} X_i\big) \cup (Z_{\langle 1\rangle}\langle\!\langle \alpha\rangle\!\rangle) \subseteq Z\}\big)$. Abbreviate $\hat{Z} = \bigcup_{i\in J}\big(\bigcap\{Z_{\langle 0\rangle} \subseteq \mathsf{Poss} \mid \big(\bigcup_{i\in J} X_i\big) \cup (Z_{\langle 1\rangle}\langle\!\langle \alpha\rangle\!\rangle) \subseteq Z\}\big)$ and $Z_i = \bigcap\{Z_{\langle 0\rangle} \subseteq \mathsf{Poss} \mid X_i \cup (Z_{\langle 1\rangle}\langle\!\langle \alpha\rangle\!\rangle) \subseteq Z\}$ for each $i \in J$. Then by monotonicity of $\cdot_{\langle 0\rangle}$ and $\cdot_{\langle 1\rangle}$ and the game semantics (Lemma 4.6) we have $(\mathsf{b},\omega) \in (\hat{Z}_{\langle 1\rangle}\langle\!\langle \alpha\rangle\!\rangle)_{\langle 0\rangle}$ and by the outer IH and by Scott-continuity of $\cdot_{\langle 0\rangle}$ and $\cdot_{\langle 1\rangle}$ have $(\mathsf{b},\omega) \in \bigcup_{i\in J}((Z_i)_{\langle 1\rangle}\langle\!\langle \alpha\rangle\!\rangle)_{\langle 0\rangle}$. To complete the subcase, note for each $i \in J$ that $(Z_i)_{\langle 1\rangle}\langle\!\langle \alpha\rangle\!\rangle \subseteq Z_i$ by (the right conjunct of) the definition of the fixed point, then finally by monotonicity of $\cdot_{\langle 0\rangle}$ we have $((Z_i)_{\langle 1\rangle}\langle\!\langle \alpha\rangle\!\rangle)_{\langle 0\rangle} \subseteq (Z_i)_{\langle 0\rangle}$, i.e., $(\mathsf{b},\omega) \in \bigcup_{i\in J} Z_i$, which, by expanding the definition of Z_i, is the conclusion of the case. This completes the inner induction, yielding $(\mathsf{b},\omega) \in \bigcup_{i\in J}\big(\bigcap\{Z_{\langle 0\rangle} \subseteq \mathsf{Poss} \mid X_i \cup (Z_{\langle 1\rangle}\langle\!\langle \alpha\rangle\!\rangle) \subseteq Z\}\big)$. Then, collapse the loop semantic definition to get $(\mathsf{b},\omega) \in \bigcup_{i\in J}\big(X_i\langle\!\langle \alpha^*\rangle\!\rangle\big)$ as desired.

Next consider the case $\top \in \big(\bigcup_{i\in J} X_i\big)\langle\!\langle \alpha^*\rangle\!\rangle$. Then $\top \in \bigcap\{Z_{\langle 0\rangle} \subseteq \mathsf{Poss} \mid \big(\bigcup_{i\in J} X_i\big) \cup (Z_{\langle 1\rangle}\langle\!\langle \alpha\rangle\!\rangle) \subseteq Z\}$. Because $\cdot_{\langle 0\rangle}$ never introduces a new occurrence of $\top$, we have $\top \in \bigcap\{Z_{\langle 0\rangle} \subseteq \mathsf{Poss} \mid \big(\bigcup_{i\in J} X_i\big) \cup (Z_{\langle 1\rangle}\langle\!\langle \alpha\rangle\!\rangle) \subseteq Z\}$

Abbreviate $\hat{Z} = \bigcup_{i\in J}\big(\bigcap\{Z \subseteq \mathsf{Poss} \mid \big(\bigcup_{i\in J} X_i\big) \cup (Z_{\langle 1\rangle}\langle\!\langle \alpha\rangle\!\rangle) \subseteq Z\}\big)$ and $Z_i = \bigcap\{Z \subseteq \mathsf{Poss} \mid X_i \cup (Z_{\langle 1\rangle}\langle\!\langle \alpha\rangle\!\rangle) \subseteq Z\}$ for each $i \in J$. This differs from their definition in the main subcase only by an application of $\cdot_{\langle 0\rangle}$.

Proceed by induction on membership of $\top$ in the fixed point-solution to prove $\top \in \bigcup_{i\in J} Z_i$. The base case holds vacuously because the case assumption contradicts our assumption $\{\top,\bot\} \cap \big(\bigcup_{i\in J} X_i\big) = \emptyset$. In the inductive case, fix Z and assume an inductive

hypothesis: if $\top \in Z$ then $\top \in \bigcup_{i\in J} Z_i$. The case assumption is that $\top \in Z_{\langle 1\rangle}\langle\!\langle \alpha\rangle\!\rangle$. Then by monotonicity of $\cdot_{\langle 1\rangle}$ and the game semantics (Lemma 4.6) we have $\top \in (\hat{Z}_{\langle 1\rangle}\langle\!\langle \alpha\rangle\!\rangle)$ and by the outer IH and by Scott-continuity of $\cdot_{\langle 1\rangle}$ have $\top \in \bigcup_{i\in J}((Z_i)_{\langle 1\rangle}\langle\!\langle \alpha\rangle\!\rangle)$. To complete the proof of the subcase, note $(Z_i)_{\langle 1\rangle}\langle\!\langle \alpha\rangle\!\rangle \subseteq Z_i = X_i\langle\!\langle \alpha^*\rangle\!\rangle$ for each $i \in J$, which was already proved in the main subcase.

The case for $\bot$ is symmetric.

Case α^d:

Have $\big(\bigcup_{i\in J} X_i\big)\langle\!\langle \alpha^d\rangle\!\rangle = \big(\bigcup_{i\in J} X_i\big)[\![\alpha]\!] \subseteq \bigcup_{i\in J}\big(X_i[\![\alpha]\!]\big) = \bigcup_{i\in J}\big(X_i\langle\!\langle \alpha^d\rangle\!\rangle\big)$ where the $\subseteq$ step is by the simultaneous IH.

Demon cases:

Case $?\phi$:

First consider the case $(\mathsf{b},\omega) \in \big(\bigcup_{i\in J} X_i\big)[\![?\phi]\!]$, then $\mathsf{b} = \mathsf{c}\,\mathsf{d}$ for some c,d such that $(\mathsf{d},\omega) \in [\![\phi]\!]$ and $(\mathsf{c},\omega) \in \big(\bigcup_{i\in J} X_i\big)$, then exists $i \in J$ such that $\mathsf{b} = \mathsf{c}\,\mathsf{d}$ for some c,d such that $(\mathsf{d},\omega) \in [\![\phi]\!]$ and $(\mathsf{c},\omega) \in X_i$, then exists $i \in J$ such that $(\mathsf{b},\omega) \in X_i[\![?\phi]\!]$, that is, $(\mathsf{b},\omega) \in \bigcup_{i\in J}\big(X_i[\![?\phi]\!]\big)$ as desired.

Consider the case $\top \in \big(\bigcup_{i\in J} X_i\big)[\![?\phi]\!]$, then there exists $(\mathsf{b},\omega) \in \big(\bigcup_{i\in J} X_i\big)$ where there is no c such that $(\mathsf{c},\omega) \in [\![\phi]\!]$, thus there exists $i \in J$ and $(\mathsf{b},\omega) \in X_i$ where there is no c such that $(\mathsf{c},\omega) \in [\![\phi]\!]$, thus there exists $i \in J$ such that $\top \in X_i[\![?\phi]\!]$ and finally $\top \in \bigcup_{i\in J}\big(X_i[\![?\phi]\!]\big)$.

Under the assumption $\bot \notin \big(\bigcup_{i\in J} X_i\big)$, the case $\bot \in \big(\bigcup_{i\in J} X_i\big)[\![?\phi]\!]$ cannot occur. By the definition of the semantics of tests, the output is always a proper possibility or $\top$.

Case $x := f$:

Assume $(\mathsf{b},\omega) \in \big(\bigcup_{i\in J} X_i\big)[\![x := f]\!]$, then $\omega = \nu[x \mapsto [\![f]\!]\omega]$ for some $(\mathsf{c},\nu) \in \big(\bigcup_{i\in J} X_i\big)$, then for some $i \in J$ and some $(\mathsf{c},\nu) \in X_i$, have $\omega = \nu[x \mapsto [\![f]\!]\omega]$, then for some $i \in J, (\mathsf{b},\omega) \in X_i[\![x := f]\!]$, then $(\mathsf{b},\omega) \in \bigcup_{i\in J}\big(X_i[\![x := f]\!]\big)$.

Under the assumption $\{\top,\bot\} \cap \big(\bigcup_{i\in J} X_i\big) = \emptyset$, the cases $\top \in \big(\bigcup_{i\in J} X_i\big)[\![x := f]\!]$ and $\bot \in \big(\bigcup_{i\in J} X_i\big)[\![x := f]\!]$ cannot occur. By the definition of the semantics of assignments, the output is always a proper possibility.

Case $x := *$:

Assume $(\mathsf{b},\omega) \in \big(\bigcup_{i\in J} X_i\big)[\![x := *]\!]$, then $\omega = \nu[x \mapsto v]$ for some $v \in \mathbb{Q}$ and $(\mathsf{c},\nu) \in \big(\bigcup_{i\in J} X_i\big)$, then for some $i \in J, \omega = \nu[x \mapsto v]$ for some $v \in \mathbb{Q}$ and $(\mathsf{c},\nu) \in X_i$, then for some $i \in J, (\mathsf{b},\omega) \in X_i[\![x := *]\!]$, then $(\mathsf{b},\omega) \in \bigcup_{i\in J}\big(X_i[\![x := *]\!]\big)$.

Under the assumption $\{\top,\bot\} \cap \big(\bigcup_{i\in J} X_i\big) = \emptyset$, the cases $\top \in \big(\bigcup_{i\in J} X_i\big)[\![x := *]\!]$ and $\bot \in \big(\bigcup_{i\in J} X_i\big)[\![x := *]\!]$ cannot occur. By the definition of the semantics of non-deterministic assignments, the output is always a proper possibility.

Case $\alpha;\beta$:

Have $\big(\bigcup_{i\in J} X_i\big)[\![\alpha;\beta]\!], = \big(\big(\bigcup_{i\in J} X_i\big)[\![\alpha]\!]\big)[\![\beta]\!] \subseteq \big(\bigcup_{i\in J}\big(X_i[\![\alpha]\!]\big)\big)[\![\beta]\!]$ by the IH on α and by Lemma 4.6, then by the IH on β have $\big(\bigcup_{i\in J}\big(X_i[\![\alpha]\!]\big)\big)[\![\beta]\!] \subseteq \bigcup_{i\in J}\big(\big(X_i[\![\alpha]\!]\big)[\![\beta]\!]\big) = \bigcup_{i\in J}\big(X_i[\![\alpha;\beta]\!]\big)$ so the case holds by transitivity.

Case $\alpha \cup \beta$:

Have

$$\begin{aligned}
&\big(\bigcup_{i\in J} X_i\big)[\![\alpha\cup\beta]\!]\\
&= \big(\bigcup_{i\in J} X_i\big)_{[0]}[\![\alpha]\!] \cup \big(\bigcup_{i\in J} X_i\big)_{[1]}[\![\beta]\!]\\
&\subseteq \big(\bigcup_{i\in J} (X_i)_{[0]}\big)[\![\alpha]\!] \cup \big(\bigcup_{i\in J} (X_i)_{[1]}\big)[\![\beta]\!]\\
&\subseteq \bigcup_{i\in J}\big(((X_i)_{[0]})[\![\alpha]\!]\big) \cup \bigcup_{i\in J}\big(((X_i)_{[1]})[\![\beta]\!]\big)\\
&= \bigcup_{i\in J}\big(((X_i)_{[0]})[\![\alpha]\!] \cup ((X_i)_{[1]})[\![\beta]\!]\big)\\
&= \bigcup_{i\in J}\big(X_i[\![\alpha\cup\beta]\!]\big)
\end{aligned}$$

where the first $\subseteq$ step holds by the Scott-continuity claim for projections and the second $\subseteq$ step holds by the IHs.

Case α^*:

First, we consider the main subcase $(\mathsf{b},\omega) \in \big(\bigcup_{i\in J} X_i\big)[\![\alpha^*]\!]$. In this subcase, we have $(\mathsf{b},\omega) \in \bigcap\{Z_{[0]} \subseteq \mathsf{Poss} \mid \big(\bigcup_{i\in J} X_i\big) \cup (Z_{[1]}[\![\alpha]\!]) \subseteq Z\}$. We now show $(\mathsf{b},\omega) \in \bigcup_{i\in J}\big(\bigcap\{Z_{[0]} \subseteq \mathsf{Poss} \mid X_i \cup (Z_{[1]}[\![\alpha]\!]) \subseteq Z\}\big)$ by induction on construction of the fixed-point solution Z. In the base case $(\mathsf{b},\omega) \in \big(\bigcup_{i\in J} X_i\big)_{[0]}$ then Scott-continuity of $\cdot_{[0]}$ yields $(\mathsf{b},\omega) \in \bigcup_{i\in J}(X_i)_{[0]}$ so that the base case of the fixed point definition applies and $(\mathsf{b},\omega) \in \bigcup_{i\in J}\big(\bigcap\{Z_{[0]} \subseteq \mathsf{Poss} \mid \big(\bigcup_{i\in J} X_i\big) \cup (Z_{[1]}[\![\alpha]\!]) \subseteq Z\}\big)$.

Now consider the inductive case $(\mathsf{b},\omega) \in (Z_{[1]}[\![\alpha]\!])_{[0]}$ for some Z. The inner IH says the claim of the inner induction holds for that Z, i.e., it says that for all $(\mathsf{c},\nu) \in Z$ we have $(\mathsf{c},\nu) \in \bigcup_{i\in J}\big(\bigcap\{Z_{[0]} \subseteq \mathsf{Poss} \mid \big(\bigcup_{i\in J} X_i\big) \cup (Z_{[1]}[\![\alpha]\!]) \subseteq Z\}\big)$. Abbreviate $\hat{Z} = \bigcup_{i\in J}\big(\bigcap\{Z_{[0]} \subseteq \mathsf{Poss} \mid \big(\bigcup_{i\in J} X_i\big) \cup (Z_{[1]}[\![\alpha]\!]) \subseteq Z\}\big)$ and $Z_i = \bigcap\{Z_{[0]} \subseteq \mathsf{Poss} \mid X_i \cup (Z_{[1]}[\![\alpha]\!]) \subseteq Z\}$ for each $i \in J$. Then by monotonicity of $\cdot_{[0]}$ and $\cdot_{[1]}$ and the game semantics (Lemma 4.6) we have $(\mathsf{b},\omega) \in (\hat{Z}_{[1]}[\![\alpha]\!])_{[0]}$ and by the outer IH and by Scott-continuity of $\cdot_{[0]}$ and $\cdot_{[1]}$ have $(\mathsf{b},\omega) \in \bigcup_{i\in J}((Z_i)_{[1]}[\![\alpha]\!])_{[0]}$. To complete the case, note for each $i \in J$ that $(Z_i)_{[1]}[\![\alpha]\!] \subseteq Z_i$ by (the right conjunct of) the definition of the fixed point, then finally by monotonicity of $\cdot_{[0]}$ we have $((Z_i)_{[1]}[\![\alpha]\!])_{[0]} \subseteq (Z_i)_{[0]}$, i.e., $(\mathsf{b},\omega) \in \bigcup_{i\in J} Z_i$, which, by expanding the definition of Z_i, is the conclusion of the case. This completes the inner induction, yielding $(\mathsf{b},\omega) \in \bigcup_{i\in J}\big(\bigcap\{Z_{[0]} \subseteq \mathsf{Poss} \mid X_i \cup (Z_{[1]}[\![\alpha]\!]) \subseteq Z\}\big)$ so that, by collapsing the definition of the loop semantics, $(\mathsf{b},\omega) \in \bigcup_{i\in J}\big(X_i[\![\alpha^*]\!]\big)$ as desired.

Next consider the case $\top \in \big(\bigcup_{i\in J} X_i\big)[\![\alpha^*]\!]$. Then $\top \in \bigcap\{Z_{[0]} \subseteq \mathsf{Poss} \mid \big(\bigcup_{i\in J} X_i\big) \cup (Z_{[1]}[\![\alpha]\!]) \subseteq Z\}$. Because $\cdot_{[0]}$ never introduces a new occurrence of $\top$, we have $\top \in \bigcap\{Z_{[0]} \subseteq \mathsf{Poss} \mid \big(\bigcup_{i\in J} X_i\big) \cup (Z_{[1]}[\![\alpha]\!]) \subseteq Z\}$

Abbreviate $\hat{Z} = \bigcup_{i\in J}\big(\bigcap\{Z \subseteq \mathsf{Poss} \mid \big(\bigcup_{i\in J} X_i\big) \cup (Z_{[1]}[\![\alpha]\!]) \subseteq Z\}\big)$ and $Z_i = \bigcap\{Z \subseteq \mathsf{Poss} \mid X_i \cup (Z_{[1]}[\![\alpha]\!]) \subseteq Z\}$ for each $i \in J$. This differs from their definition in the main subcase only by an application of $\cdot_{[0]}$.

Proceed by induction on membership of $\top$ in the fixed point-solution to prove $\top \in$

$\bigcup_{i\in J} Z_i$. The base case holds vacuously because the case assumption contradicts our assumption $\{\top,\bot\}\cap\big(\bigcup_{i\in J} X_i\big)=\emptyset$. In the inductive case, fix Z and assume an inductive hypothesis: if $\top\in Z$ then $\top\in\bigcup_{i\in J} Z_i$. The case assumption is that $\top\in Z_{[1]}[\![\alpha]\!]$. Then by monotonicity of $\cdot_{[1]}$ and the game semantics (Lemma 4.6) we have $\top\in(\hat{Z}_{[1]}[\![\alpha]\!])$ and by the outer IH and by Scott-continuity of $\cdot_{[1]}$ have $\top\in\bigcup_{i\in J}((Z_i)_{[1]}[\![\alpha]\!])$. To complete the proof of the subcase, note $(Z_i)_{[1]}[\![\alpha]\!]\subseteq Z_i = X_i[\![\alpha^*]\!]$ for each $i\in J$, which was already proved in the main subcase.

The case for $\bot$ is symmetric.

Case α^d:

Have $\big(\bigcup_{i\in J} X_i\big)[\![\alpha^d]\!] = \big(\bigcup_{i\in J} X_i\big)\langle\!\langle\alpha\rangle\!\rangle \subseteq \bigcup_{i\in J}\big(X_i\langle\!\langle\alpha\rangle\!\rangle\big) = \bigcup_{i\in J}\big(X_i[\![\alpha^d]\!]\big)$ where the $\subseteq$ step is by the simultaneous IH.

□

Lemma A.1 (Pointwise partition)**.** *Let α be a game and ϕ a formula. In each case respectively let $\mathsf{b}\in\langle\alpha\rangle\phi\,\mathcal{R}\mathbf{z}$ or $\mathsf{b}\subseteq[\alpha]\phi\,\mathcal{R}\mathbf{z}$ for all realizers b from region X to ensure the semantics of α are well-defined.*

$$X\langle\!\langle\alpha\rangle\!\rangle = \bigcup_{\mathsf{poss}\in X}\{\mathsf{poss}\}\langle\!\langle\alpha\rangle\!\rangle$$
$$X[\![\alpha]\!] = \bigcup_{\mathsf{poss}\in X}\{\mathsf{poss}\}[\![\alpha]\!]$$

Proof. Apply Lemma 4.7, letting $J=X$ and letting the family $X_i=\{\{\mathsf{poss}\}\mid\mathsf{poss}\in X\}$, i.e., the family of singleton sets containing each element of X. The partition property follows immediately by noting $X=\bigcup_{i\in J} X_i$ by construction of J and X_i. □

We give the proofs of lemmas relating iterative and fixed-point semantics. In general, when we wish to give names to facts in our proofs for later reference, we write the names (Fact) or numbers (1) in parentheses.

Definition 4.12 (Iterative CGL semantics)**.** We define the k-step Angelic and Demonic iterations recursively for $k\in\mathbb{N}$, i.e.,

$$X\langle\!\langle\alpha\rangle\!\rangle^0 = X \qquad X\langle\!\langle\alpha\rangle\!\rangle^{k+1} = (X\langle\!\langle\alpha\rangle\!\rangle^k)_{\langle1\rangle}\langle\!\langle\alpha\rangle\!\rangle$$
$$X[\![\alpha]\!]^0 = X \qquad X[\![\alpha]\!]^{k+1} = (X[\![\alpha]\!]^k)_{\langle1\rangle}[\![\alpha]\!]$$

Remark 4.1 (Pre-iteration and post-iteration). Pre-iteration and post-iteration agree in the following sense:

$$(X_{\langle1\rangle}\langle\!\langle\alpha\rangle\!\rangle)\langle\!\langle\alpha\rangle\!\rangle^k = (X\langle\!\langle\alpha\rangle\!\rangle^k)_{\langle1\rangle}\langle\!\langle\alpha\rangle\!\rangle \qquad (X_{[1]}[\![\alpha]\!])[\![\alpha]\!]^k = (X[\![\alpha]\!]^k)_{[1]}[\![\alpha]\!]$$

Proof. Each claim is by induction on k.

Angel case: 0

$$(X_{\langle1\rangle}\langle\!\langle\alpha\rangle\!\rangle)\langle\!\langle\alpha\rangle\!\rangle^0 = X_{\langle1\rangle}\langle\!\langle\alpha\rangle\!\rangle = ((X\langle\!\langle\alpha\rangle\!\rangle^0)_{\langle1\rangle})\langle\!\langle\alpha\rangle\!\rangle$$

Angel case: $k+1$
Assume IH: $(X_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle)\langle\!\langle\alpha\rangle\!\rangle^k = ((X\langle\!\langle\alpha\rangle\!\rangle^k)_{\langle 1\rangle})\langle\!\langle\alpha\rangle\!\rangle$
Show:

$$\begin{aligned}
&(X_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle)\langle\!\langle\alpha\rangle\!\rangle^{k+1}\\
&= ((X_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle)_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle)\langle\!\langle\alpha\rangle\!\rangle^k\\
&\underset{\text{IH}}{=} ((X\langle\!\langle\alpha\rangle\!\rangle)_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle^k)_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle\\
&= (X\langle\!\langle\alpha\rangle\!\rangle^{k+1})_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle
\end{aligned}$$

as desired, completing the induction for Angel.
Demon case: 0

$$(X_{[1]}[\![\alpha]\!])[\![\alpha]\!]^0 = X_{[1]}[\![\alpha]\!] = (X[\![\alpha]\!]^0)_{[1]}[\![\alpha]\!]$$

Demon case: $k+1$
Assume IH: $(X_{[1]}[\![\alpha]\!])[\![\alpha]\!]^k = ((X[\![\alpha]\!]^k)_{[1]})[\![\alpha]\!]$
Show:

$$\begin{aligned}
&(X_{[1]}[\![\alpha]\!])[\![\alpha]\!]^{k+1}\\
&= (((X_{[1]}[\![\alpha]\!])_{[1]})[\![\alpha]\!])[\![\alpha]\!]^k\\
&\underset{\text{IH}}{=} ((X[\![\alpha]\!])_{[1]}[\![\alpha]\!]^k)_{[1]}[\![\alpha]\!]\\
&= (X[\![\alpha]\!]^{k+1})_{[1]}[\![\alpha]\!]
\end{aligned}$$

as desired, completing the induction for Demon. □

Lemma 4.8 (Alternative semantics)**.** *The CGL fixed-point definition of repetition agrees with the iterative definition with closure ordinal ω:*

$$X\langle\!\langle\alpha^*\rangle\!\rangle = \bigcup_{k\in\mathbb{N}} (X\langle\!\langle\alpha\rangle\!\rangle^k)_{\langle 0\rangle} \qquad X[\![\alpha^*]\!] = \bigcup_{k\in\mathbb{N}} (X[\![\alpha]\!]^k)_{[0]}$$

Proof. We give a direct proof for Angel. The proof for Demon is symmetric.

Fix an initial region X Let $f(Z) = X \cup Z_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle$ and write $f^k(Z)$ for the kth iteration of f from initial region Z. By construction of f and definition of $X\langle\!\langle\alpha\rangle\!\rangle^k$ (Def. 4.12) have $f^k(Z) = \bigcup_{i\in[0,k]} X\langle\!\langle\alpha\rangle\!\rangle^k$. By Lemma 4.7, f is Scott-continuous, so by Kleene's fixed point theorem, its least fixed point exists and has closure ordinal at most ω. Write $\mathsf{lfp}(f, Z)$ for the least fixed-point of f from initial region Z.

We reason by equality:

$$
\begin{aligned}
& X\langle\!\langle \alpha^* \rangle\!\rangle \\
&= \bigcap \{ Z_{\langle 0 \rangle} \subseteq \mathsf{Poss} \mid X \cup (Z_{\langle 1 \rangle}\langle\!\langle \alpha \rangle\!\rangle) \subseteq Z \} && \text{By definition} \\
&= \big(\bigcap \{ Z \subseteq \mathsf{Poss} \mid X \cup (Z_{\langle 1 \rangle}\langle\!\langle \alpha \rangle\!\rangle) \subseteq Z \} \big)_{\langle 0 \rangle} && \text{Since } \cdot_{\langle 0 \rangle} \text{ distributes over } \bigcap \\
&= \mathsf{lfp}(f, X)_{\langle 0 \rangle} && \text{Def. of loop semantics as l.f.p.} \\
&= \big(\bigcup_{k \in \mathbb{N}} f^k(X) \big)_{\langle 0 \rangle} && \text{Kleene's fixed point theorem} \\
&= \big(\bigcup_{k \in \mathbb{N}} \big(\bigcup_{j \in [0,k]} X\langle\!\langle \alpha \rangle\!\rangle^j \big) \big)_{\langle 0 \rangle} && \text{Construction of } f \\
&= \big(\bigcup_{k \in \mathbb{N}} X\langle\!\langle \alpha \rangle\!\rangle^k \big)_{\langle 0 \rangle} && \text{Simplify union} \\
&= \bigcup_{k \in \mathbb{N}} (X\langle\!\langle \alpha \rangle\!\rangle^k)_{\langle 0 \rangle} && \text{Since } \cdot_{\langle 0 \rangle} \text{ distributes over } \bigcap
\end{aligned}
$$

thus the claim holds by transitivity. □

A.4.5 Repetition Realizers and Monotonicity

We give definitions of inductive and coinductive realizers for Angelic and Demonic loops. We first discuss monotonicity, because monotonicity reasoning is essential to implement the realizers.

The monotonicity rule M (Section 4.7) says that valid implications remain valid when lowered under the same modality:

$$\text{(M)} \quad \frac{\Gamma \vdash M : \langle\alpha\rangle\phi \quad \Gamma_{\frac{\vec{y}}{\mathrm{BV}(\alpha)}}, p : \phi \vdash N : \psi}{\Gamma \vdash M \circ_p N : \langle\alpha\rangle\psi}\ {}^{1}$$

[1] Variables $\vec{y}$ are fresh and $|\vec{y}| = |\,\mathrm{BV}(\alpha)|$

Monotonicity is a general principle which makes an appearance in the construction of loop realizers because loop realizer execution proceeds by executing the loop body, then potentially repeating the loop recursively; that recursive call is executed in a new state and appears "under" the first execution. As suggested by the operational semantics (Section 4.9), the operational effect of monotonicity is nontrivial: a monotonicity realizer is constructed by traversing the modal program and showing that the valid implication holds for any program. As a helper function for the loop realizers, we define a monotonicity operation on realizers which can construct the realizers corresponding to a monotonicity argument. As in the operational semantics, monotonicity is defined admissibly from existing constructs, but the realizer semantics are also prior to the operational semantics, so we define monotonicity realizers from scratch rather than attempt to reuse the operational treatment for realizability purposes.

Definition A.2 (Monotone realizers)**.** We define two functions $\mathsf{mon}_{\langle\alpha\rangle}\ \mathsf{b}\ \mathsf{c}$ and $\mathsf{mon}_{[\alpha]}\ \mathsf{b}\ \mathsf{c}$. In both functions, b is a $(\phi \to \psi)\,\mathcal{R}\mathbf{z}$ for some formulas ϕ and ψ. The Angelic realizer

$\mathsf{mon}_{\langle\alpha\rangle}\ \mathsf{b}\ \mathsf{c}$ is a $\langle\alpha\rangle\psi\,\mathcal{R}\mathbf{z}$ when c is a $\langle\alpha\rangle\phi\,\mathcal{R}\mathbf{z}$ and the Demonic realizer $\mathsf{mon}_{[\alpha]}\ \mathsf{b}\ \mathsf{c}$ is a $[\alpha]\psi\,\mathcal{R}\mathbf{z}$ when c is a $[\alpha]\phi\,\mathcal{R}\mathbf{z}$.

The definition of monotone realizers is corecursive in realizer c and inductive in game α. That is, the definition is well-founded because every corecursive call is either guarded by a constructor (a pair or λ) or simplifies the game α. Because any chain of subgames of α is finite, non-zero progress is always made after a finite number of corecursive calls.

$$
\begin{aligned}
\mathsf{mon}_{\langle ?\psi\rangle}\ \mathsf{b}\ \mathsf{c} &= (\pi_0\mathsf{c}, \mathsf{b}\ (\pi_1\mathsf{c}))\\
\mathsf{mon}_{\langle x:=f\rangle}\ \mathsf{b}\ \mathsf{c} &= \mathsf{b}\ \mathsf{c}\\
\mathsf{mon}_{\langle x:=*\rangle}\ \mathsf{b}\ \mathsf{c} &= (\pi_0\mathsf{c}, \mathsf{b}\ (\pi_1\mathsf{c}))\\
\mathsf{mon}_{\langle \alpha;\beta\rangle}\ \mathsf{b}\ \mathsf{c} &= \mathsf{mon}_{\langle\alpha\rangle}\ (\Lambda \mathsf{d} : \langle\beta\rangle\phi\,\mathcal{R}\mathbf{z}.\ \mathsf{mon}_{\langle\beta\rangle}\ \mathsf{b}\ \mathsf{d})\ \mathsf{c}\\
\mathsf{mon}_{\langle \alpha\cup\beta\rangle}\ \mathsf{b}\ \mathsf{c} &= \mathtt{if}\ (\pi_0\mathsf{c} = 0)\ (0, \mathsf{mon}_{\langle\alpha\rangle}\ \mathsf{b}\ (\pi_1\mathsf{c}))\ \mathtt{else}\ (1, \mathsf{mon}_{\langle\beta\rangle}\ \mathsf{b}\ (\pi_1\mathsf{c}))\\
\mathsf{mon}_{\langle \alpha^d\rangle}\ \mathsf{b}\ \mathsf{c} &= \mathsf{mon}_{[\alpha]}\ \mathsf{b}\ \mathsf{c}\\
\mathsf{mon}_{\langle \alpha^*\rangle}\ \mathsf{b}\ \mathsf{c} &= \mathtt{if}(\pi_0\mathsf{c} = 0)\\
&\qquad\qquad (0, \mathsf{b}\ (\pi_1\mathsf{c}))\\
&\qquad \mathtt{else}\ (1, \mathsf{mon}_{\langle\alpha\rangle}\ (\Lambda \mathsf{d} : \langle\alpha^*\rangle\phi\,\mathcal{R}\mathbf{z}.\ \mathsf{mon}_{\langle\alpha^*\rangle}\ \mathsf{b}\ \mathsf{d})\ (\pi_1\mathsf{c}))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{mon}_{[?\psi]}\ \mathsf{b}\ \mathsf{c} &= \Lambda \mathsf{d} : \psi\,\mathcal{R}\mathbf{z}.\ \mathsf{b}\ (\mathsf{c}\ \mathsf{d})\\
\mathsf{mon}_{[x:=f]}\ \mathsf{b}\ \mathsf{c} &= \mathsf{b}\ \mathsf{c}\\
\mathsf{mon}_{[x:=*]}\ \mathsf{b}\ \mathsf{c} &= \Lambda q : \mathbb{Q}.\ (\mathsf{b}\ (\mathsf{c}\ q))\\
\mathsf{mon}_{[\alpha;\beta]}\ \mathsf{b}\ \mathsf{c} &= \mathsf{mon}_{[\alpha]}\ (\Lambda \mathsf{d} : [\beta]\phi\,\mathcal{R}\mathbf{z}.\ \mathsf{mon}_{[\beta]}\ \mathsf{b}\ \mathsf{d})\ \mathsf{c}\\
\mathsf{mon}_{[\alpha\cup\beta]}\ \mathsf{b}\ \mathsf{c} &= (\mathsf{mon}_{[\alpha]}\ \mathsf{b}\ (\pi_0\mathsf{c}), \mathsf{mon}_{[\beta]}\ \mathsf{b}\ (\pi_1\mathsf{c}))\\
\mathsf{mon}_{[\alpha^d]}\ \mathsf{b}\ \mathsf{c} &= \mathsf{mon}_{\langle\alpha\rangle}\ \mathsf{b}\ \mathsf{c}\\
\mathsf{mon}_{[\alpha^*]}\ \mathsf{b}\ \mathsf{c} &= (\mathsf{b}\ (\pi_0\mathsf{c}), \mathsf{mon}_{[\alpha]}\ (\Lambda \mathsf{d} : [\alpha^*]\phi\,\mathcal{R}\mathbf{z}.\ \mathsf{mon}_{[\alpha^*]}\ \mathsf{b}\ \mathsf{d})\ (\pi_1\mathsf{c}))
\end{aligned}
$$

We have already given one lemma about monotonicity in CGL: the simple monotonicity lemma (Lemma 4.6). The simple characterization of monotonicity is particularly important in order to show the existence of fixed points in the loop semantics but it only addresses monotonicity of regions which realize the same postcondition formula, which is not enough for the monotonicity rule M because it changes a postcondition. We now give a more powerful notion of monotonicity, Lemma A.2, which characterizes the effect of $\mathsf{mon}_{\langle\alpha\rangle}$ and $\mathsf{mon}_{[\alpha]}$: they monotonically lower valid implications of arbitrary implications $\phi \to \psi$ under modalities. Moreover, the monotonicity realizers are also understood as constructive witnesses for the proof of Lemma A.2.

Lemma A.2 (Monotonicity realizers)**.** *In each case, let R be the set of states reachable by executing α starting from possibility (c,ω), which are respectively $\{\nu \mid \exists \mathsf{d}\ (\mathsf{d},\nu) \in \{(\mathsf{c},\omega)\}[\![\alpha]\!]\}$ and $\{\nu \mid \exists \mathsf{d}\ (\mathsf{d},\nu) \in \{(\mathsf{c},\omega)\}\langle\!\langle\alpha\rangle\!\rangle\}$. Assume some $S \supseteq R$. Assume $\{\mathsf{b}\} \times S \subseteq [\![\phi \to \psi]\!]$, i.e., let b realize $\phi \to \psi$ on set of states $S \supseteq R$. The following claims hold.*

- *If $(\mathsf{c},\omega) \in [\![[\alpha]\phi]\!]$ then $(\mathsf{mon}_{[\alpha]}\ \mathsf{b}\ \mathsf{c},\omega) \in [\![[\alpha]\psi]\!]$.*
- *If $(\mathsf{c},\omega) \in [\![\langle\alpha\rangle\phi]\!]$ then $(\mathsf{mon}_{\langle\alpha\rangle}\ \mathsf{b}\ \mathsf{c},\omega) \in [\![\langle\alpha\rangle\psi]\!]$.*

Proof. By outer induction on α with simultaneous inductions for Angel and Demon, and by inner coinduction on the realizer c. In each case, unpack the definition of $\mathsf{mon}_{\langle\alpha\rangle}$ or $\mathsf{mon}_{[\alpha]}$, and unpack the semantics of α, then apply each (co-)inductive hypothesis. In the definition of $\mathsf{mon}_{\langle\alpha\rangle}$ or $\mathsf{mon}_{[\alpha]}$, the shape of c is preserved and all strategic decisions of α are resolved identically, so all inductive cases of the proof map through homomorphically. In the base cases, realizer b proves the new postcondition ψ by consuming the proof of ϕ. The base cases only ever apply ψ at states in the final region R of α, so it is sufficient to assume that b realizes the implication $\phi \to \psi$ on final states, rather than all states. □

We repeat the statement of simple monotonicity (Lemma 4.6) for the sake of contrast.

Lemma 4.6 (Monotonicity)**.** *Assume $X \subseteq Y$ and let ϕ be an arbitrary CGL formula. In each respective claim, let $\mathsf{b} \in \langle\alpha\rangle\phi\,\mathcal{R}\mathbf{z}$ or $\mathsf{b} \in [\alpha]\phi\,\mathcal{R}\mathbf{z}$ for every realizer b in Y so that the realizers in Y have the shapes expected by the semantic functions, e.g., realizers of Angelic tests are pairs.*

- $X\langle\!\langle\alpha\rangle\!\rangle \subseteq Y\langle\!\langle\alpha\rangle\!\rangle$
- $X[\![\alpha]\!] \subseteq Y[\![\alpha]\!]$

We are now able to define inductive and coinductive realizers using monotonicity:

Definition A.3 (Inductive and coinductive realizers)**.** We give direct definitions of realizers $\mathsf{ind}(\mathsf{b},\,\mathsf{c},\,\mathsf{d})_{\mathcal{M}}$ and $\mathsf{gen}(\mathsf{b},\,\mathsf{c},\,\mathsf{d})$. Realizer $\mathsf{ind}(\mathsf{b},\,\mathsf{c},\,\mathsf{d})_{\mathcal{M}}$ is a $\langle\alpha^*\rangle\phi$-realizer if for termination metric $\mathcal{M}$ and some formula φ we have: b is a φ-realizer, c is a $(\forall\mathcal{M}_0\,(\varphi\wedge(\mathcal{M}_0 = \mathcal{M}\wedge\mathcal{M} \succ \mathbf{0})) \to \langle\alpha\rangle(\varphi\wedge\mathcal{M}_0 \succ \mathcal{M}))$-realizer, and d is a $((\varphi\wedge\mathbf{0} \succcurlyeq \mathcal{M}) \to \phi)$-realizer. Realizer $\mathsf{gen}(\mathsf{b},\,\mathsf{c},\,\mathsf{d})$ is a $[\alpha^*]\phi$-realizer if there exists a formula ψ, such that b is a ψ-realizer, c is a $(\psi \to [\alpha]\psi)$-realizer, and d is a $(\psi \to \phi)$-realizer.

$$
\begin{aligned}
\mathsf{ind}(\mathsf{b},\,\mathsf{c},\,\mathsf{d})_{\mathcal{M}} \equiv\ & \mathtt{if}(\mathbf{0} \succcurlyeq \mathcal{M}) \\
& \qquad (0, \mathsf{d}\ (\mathsf{b}, \epsilon)) \\
& \mathtt{else}\ (1, \mathsf{mon}_{\langle\alpha\rangle} \\
& \qquad (\Lambda \mathsf{rz} : (\varphi \wedge (\hat{\mathcal{M}}_0 = \mathcal{M} \wedge \mathcal{M} \succ \mathbf{0}))\,\mathcal{R}\mathbf{z}.\ \mathsf{ind}(\pi_0\mathsf{rz},\,\mathsf{c},\,\mathsf{d})_{\mathcal{M}}) \\
& \qquad ((\mathsf{c}\ \mathcal{M})(\mathsf{b}, (\epsilon, \epsilon))))) \\
\mathsf{gen}(\mathsf{b},\,\mathsf{c},\,\mathsf{d}) \equiv\ & (\mathsf{d}\ \mathsf{b}, \\
& \qquad \mathsf{mon}_{[\alpha]}\ (\Lambda \mathsf{rz} : \psi\,\mathcal{R}\mathbf{z}.\ \mathsf{gen}(\mathsf{rz},\,\mathsf{c},\,\mathsf{d}))\ (\mathsf{c}\ \mathsf{b}))
\end{aligned}
$$

Where:

- ψ in $\mathsf{gen}(\mathsf{b},\,\mathsf{c},\,\mathsf{d})$ is the loop invariant used in the corresponding loop invariant proof,
- and $\hat{\mathcal{M}}_0$ in $\mathsf{ind}(\mathsf{b},\,\mathsf{c},\,\mathsf{d})_{\mathcal{M}}$ is, in a minor handwave, a numeric literal term which is equal to the value of $\mathcal{M}$ at the initial state of the realizer.

Note that the conditional statement in $\mathsf{ind}(\mathsf{b},\,\mathsf{c},\,\mathsf{d})_{\mathcal{M}}$ branches on the truth of the guard condition $\mathbf{0} \succcurlyeq \mathcal{M}$ to determine whether the loop should stop yet, which it can do because CGL (in contrast to CdGL) assumes that all loop guard conditions are decidable. Because guard conditions are typically composed of decidable comparisons over rationals, the assumption is a small one.

When we prove soundness of the proof calculus (Theorem 4.15), the proof case for each rule will construct a realizer and show the realizer witnesses the conclusion of the rule. The soundness cases for loop introduction rules will reduce to showing that the loop realizers witness the conclusions of the loop introduction rules. The following lemma characterizes the formula witnessed by each loop realizer, so that the soundness cases for loop introduction rules will be proved by appealing to our lemma.

Lemma A.3 (Loop realizers)**.** *Realizers* $\mathsf{ind}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}}$ *and* $\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})$ *are sound implementations of rules* $\langle * \rangle I$ *and* $[*]I$*, respectively.*

The claim for Angelic loops says that (for any invariant formula φ*, postcondition* ϕ*, and well-founded metric* $\mathcal{M}$*)*

- *If* $(\mathsf{b}, \omega) \in [\![\varphi]\!]$
- *and* $(\{\mathsf{c}\} \times \mathcal{S}) \subseteq [\![\forall \mathcal{M}_0\, (\varphi \wedge (\mathcal{M}_0 = \mathcal{M} \wedge \mathcal{M} \succ \mathbf{0}) \rightarrow \langle \alpha \rangle (\varphi \wedge \mathcal{M}_0 \succ \mathcal{M}))]\!]$
- *and* $(\{\mathsf{d}\} \times \mathcal{S}) \subseteq [\![\varphi \wedge \mathbf{0} \succcurlyeq \mathcal{M} \rightarrow \phi]\!]$*,*
- *then* $(\mathsf{ind}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}}, \omega) \in [\![\langle \alpha^* \rangle \phi]\!]$

The claim for Demonic loops says (for invariant formula ψ *and postcondition* ϕ*);*

- *If* $(\mathsf{b}, \omega) \in [\![\psi]\!]$
- *and* $(\{\mathsf{c}\} \times \mathcal{S}) \subseteq [\![\psi \rightarrow [\alpha]\psi]\!]$
- *and* $(\{\mathsf{d}\} \times \mathcal{S}) \subseteq [\![\psi \rightarrow \phi]\!]$*,*
- *then* $(\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d}), \omega) \in [\![[\alpha^*]\phi]\!]$

Proof. **Angel cases:**

Call the assumptions (1), (2), and (3). By the loop semantics, suffices to assume poss such that (Loop) $\mathsf{poss} \in \{(\mathsf{ind}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}}, \omega)\}\langle\!\langle \alpha^* \rangle\!\rangle$ and prove $\mathsf{poss} \in [\![\phi]\!] \cup \{\top\}$.

The proof is by induction, allowing (b, ω) to vary to any other $(\hat{\mathsf{b}}, \nu)$ satisfying assumption (1) $(\hat{\mathsf{b}}, \nu) \in [\![\varphi]\!]$. The induction is on the value $[\![\mathcal{M}]\!]\omega$ of metric $\mathcal{M}$ in state ω, where $\mathcal{M}$ is a well- founded metric by assumption. While the induction is on the metric $\mathcal{M}$, we will also make use of the iterative semantics (Lemma 4.8):

$$\{(\mathsf{ind}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}}, \omega)\}\langle\!\langle \alpha^* \rangle\!\rangle = \bigcup_{k \in \mathbb{N}} (\{(\mathsf{ind}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}}, \omega)\}\langle\!\langle \alpha \rangle\!\rangle^k)_{\langle 0 \rangle}$$

in each case of the induction.

The base case is the case where (4) $(\epsilon, \omega) \in [\![\mathbf{0} \succcurlyeq \mathcal{M}]\!]$, i.e., the case where the metric says to stop. Then from (4), $([\![\pi_0(\mathsf{ind}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}})]\!]\omega) = 0$ by construction so that $\{(\mathsf{ind}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}}, \omega)\}_{\langle 1 \rangle} = \emptyset$, so that poss belongs to $\cdot_{\langle 0 \rangle}$ in (Poss), i.e.,

$$\begin{aligned} \mathsf{poss} &\in \{(\mathsf{ind}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}}, \omega)\}_{\langle 0 \rangle} \\ \text{thus } \mathsf{poss} &= (\pi_1(\mathsf{ind}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}}), \omega) \end{aligned}$$

which implies (5) $\mathsf{poss} = (\mathsf{d}\ (\mathsf{b}, \epsilon), \omega)$ by construction of $\mathsf{ind}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}}$. Thus, it suffices to show $(\mathsf{d}\ (\mathsf{b}, \epsilon), \omega) \in [\![\phi]\!]$. By (3) it suffices to show $((\mathsf{b}, \epsilon), \omega) \in [\![\varphi \wedge \mathbf{0} \succcurlyeq \mathcal{M}]\!]$ which respectively follows from (1) and (4), completing the base case.

In the inductive case, (NotDone) $(\epsilon, \omega) \in [\![\mathcal{M} \succ \mathbf{0}]\!]$. Let $\hat{\mathcal{M}}_0$ be the literal numeric term such that $\hat{\mathcal{M}}_0 = [\![\mathcal{M}]\!]\omega$.

Inductively assume (IH) that for all $\hat{\mathsf{b}}, \nu$ such that $[\![\mathcal{M}]\!]\omega \succ [\![\mathcal{M}]\!]\nu$ (i.e., ν is smaller than ω according to metric $\mathcal{M}$) and $(\hat{\mathsf{b}}, \nu) \in \ulcorner\varphi\urcorner$ (i.e., ν satisfies assumption (1)) then $(\hat{\mathsf{ind}}(\hat{\mathsf{b}},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}}, \nu) \in [\![\langle\alpha^*\rangle\phi]\!]$, i.e., the conclusion holds for ν.

Next, we expand the iterative semantics (Remark 4.1) and note by (NotDone) that the loop executes for at least one iteration. That is, from $\mathsf{poss} \in \{(\mathsf{ind}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}}, \omega)\}\langle\!\langle\alpha^*\rangle\!\rangle$ and (NotDone) have that there exists $i \in \mathbb{N}$ such that

$$\begin{aligned} \mathsf{poss} &\in \{(\mathsf{ind}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}}, \omega)\}\langle\!\langle\alpha\rangle\!\rangle^{i+1}_{\langle 0\rangle} \\ \text{thus } \mathsf{poss} &\in \{(\mathsf{ind}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}}, \omega)\}\langle\!\langle\alpha\rangle\!\rangle^{i}_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle_{\langle 0\rangle} \\ \text{thus } \mathsf{poss} &\in \{(\mathsf{ind}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}}, \omega)\}_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle\langle\!\langle\alpha\rangle\!\rangle^{i}_{\langle 0\rangle} \end{aligned}$$

To minimize the number of cases, note by (2) that $\bot \notin \{(\mathsf{ind}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}}, \omega)\}_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle$ since $\bot \notin [\![(\varphi \wedge \mathcal{M}_0 \succ \mathcal{M})]\!] \cup \{\top\}$. Combined with a likewise argument from the IH and for the $\cdot_{\langle 0\rangle}$ operation, we have $\bot \notin \{(\mathsf{ind}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}}, \omega)\}\langle\!\langle\alpha\rangle\!\rangle^{i+1}_{\langle 0\rangle}$ so that $\mathsf{poss} \neq \bot$. Secondly, note the case $\mathsf{poss} = \top$ trivially satisfies $\mathsf{poss} \in [\![\phi]\!] \cup \{\top\}$. Thus, it only remains to show the case where poss has form $(\hat{\mathsf{rz}}, \mu)$ for some realizer $\hat{\mathsf{rz}}$ and state μ.

From (NotDone) and construction of $\mathsf{ind}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}}$ have (5)

$$\begin{aligned} &\pi_1(\mathsf{ind}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}}) \\ &= \mathsf{mon}_{\langle\alpha\rangle}\ (\Lambda\mathsf{rz} : (\varphi \wedge (\hat{\mathcal{M}}_0 = \mathcal{M} \wedge \mathcal{M} \succ \mathbf{0}))\, \mathcal{R}\mathbf{z}.\ \mathsf{ind}(\pi_0\mathsf{rz},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}})\ ((\mathsf{c}\ \mathcal{M})\ (\mathsf{b}, (\epsilon, \epsilon))) \end{aligned}$$

Thus, (In5)

$$\begin{aligned} &(\hat{\mathsf{rz}}, \mu) \in \\ &(\{(\mathsf{mon}_{\langle\alpha\rangle}\ (\Lambda\mathsf{rz} : (\varphi \wedge (\hat{\mathcal{M}}_0 = \mathcal{M} \wedge \mathcal{M} \succ \mathbf{0}))\, \mathcal{R}\mathbf{z}.\ \mathsf{ind}(\pi_0\mathsf{rz},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}})\ ((\mathsf{c}\ \mathcal{M})\ (\mathsf{b}, (\epsilon, \epsilon))), \omega)\} \\ &\langle\!\langle\alpha\rangle\!\rangle\langle\!\langle\alpha\rangle\!\rangle^{i}_{\langle 0\rangle}) \end{aligned}$$

We proceed to analyze (In5) by applying Lemma A.2, whose preconditions we must first prove. In the application of Lemma A.2, let $R = \{(((\mathsf{c}\ \mathcal{M})(\mathsf{b}, (\epsilon, \epsilon))), \omega)\}\langle\!\langle\alpha\rangle\!\rangle$ be the final region resulting from executing the first iteration of α in (5). Let $\hat{R} = \{\omega \mid (\mathsf{b}, \omega) \in R \text{ for some } \mathsf{b}\}$ be all the states of R. The preconditions we need to show are (PC1)

$$\{(\Lambda\mathsf{rz} : (\varphi \wedge (\hat{\mathcal{M}}_0 = \mathcal{M} \wedge \mathcal{M} \succ \mathbf{0}))\, \mathcal{R}\mathbf{z}.\ \mathsf{ind}(\pi_0\mathsf{rz},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}})\} \times \hat{R} \subseteq [\![((\varphi \wedge \hat{\mathcal{M}}_0 \succ \mathcal{M}) \to \langle\alpha^*\rangle\phi)]\!]$$

and (PC2) $(((\mathsf{c}\ \mathcal{M})\ (\mathsf{b}, (\epsilon, \epsilon))), \omega) \in [\![\langle\alpha\rangle(\varphi \wedge \hat{\mathcal{M}}_0 \succ \mathcal{M})]\!]$ which will result in (Mon):

$$(\mathsf{mon}_{\langle\alpha\rangle}\ \mathsf{b}\ \mathsf{c}, \omega) \in [\![\langle\alpha\rangle\langle\alpha^*\rangle\phi]\!]$$

Precondition (PC2) holds from (1) and (2) and by noting $[\![\mathcal{M}]\!]\omega = \hat{\mathcal{M}}_0$ by construction of literal numeric term $\hat{\mathcal{M}}_0$. To show (PC1), consider arbitrary $\nu \in \hat{R}$ and show (7):

$$((\Lambda\mathsf{rz} : (\varphi \wedge (\hat{\mathcal{M}}_0 = \mathcal{M} \wedge \mathcal{M} \succ \mathbf{0}))\, \mathcal{R}\mathbf{z}.\ \mathsf{ind}(\pi_0\mathsf{rz},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}}), \nu) \in [\![((\varphi \wedge \hat{\mathcal{M}}_0 \succ \mathcal{M}) \to \langle\alpha^*\rangle\phi)]\!]$$

To do so, we fix some rz such that (2ν) $(\mathsf{rz}, \nu) \in [\![(\varphi \wedge \hat{\mathcal{M}}_0 \succ \mathcal{M})]\!]$ and show (PC2Simp)

$$((\mathsf{ind}(\pi_0\mathsf{rz},\ \mathsf{c},\ \mathsf{d})_{\mathcal{M}}), \nu) \in [\![\langle\alpha^*\rangle\phi]\!]$$

Note (PC2Simp) would be the result of applying the IH on ν, so we apply the IH on ν. To apply the IH we must show the well-foundedness condition $[\![\mathcal{M}]\!]\omega \succ [\![\mathcal{M}]\!]\nu$ and that assumption (1) holds for $(\pi_0\mathsf{rz}, \nu)$. The well-foundedness condition holds by (2ν) since $[\![\mathcal{M}]\!]\omega = \hat{\mathcal{M}}_0 = [\![\hat{\mathcal{M}}_0]\!]\nu \succ [\![\mathcal{M}]\!]\nu$ where the numeric literal term $\hat{\mathcal{M}}_0$ equals $[\![\mathcal{M}]\!]\omega$ by its construction. The assumption (1) holds for $(\pi_0\mathsf{rz}, \nu)$ because $(\pi_0\mathsf{rz}, \nu) \in [\![\varphi]\!]$ by (2ν). Then the IH immediately gives (PC2Simp) which gives (PC2) and thus completes the application of Lemma A.2, yielding (Mon):

$$(\mathsf{mon}_{\langle\alpha\rangle}\,(\Lambda \mathsf{rz} : (\varphi \wedge (\hat{\mathcal{M}}_0 = \mathcal{M} \wedge \mathcal{M} \succ \mathbf{0}))\,\mathcal{R}\mathbf{z}.\,\mathsf{ind}(\pi_0\mathsf{rz},\,\mathsf{c},\,\mathsf{d})_{\mathcal{M}})\;((\mathsf{c}\,\mathcal{M})(\mathsf{b},(\epsilon,\epsilon))), \omega) \in [\![\langle\alpha\rangle\langle\alpha^*\rangle\phi]\!]$$

which simplifies to

$$(\pi_1(\mathsf{ind}(\mathsf{b},\,\mathsf{c},\,\mathsf{d})_{\mathcal{M}}), \omega) \in [\![\langle\alpha\rangle\langle\alpha^*\rangle\phi]\!]$$

and by (NotDone) further simplifies to

$$(\{(\mathsf{ind}(\mathsf{b},\,\mathsf{c},\,\mathsf{d})_{\mathcal{M}})\}_{\langle 1\rangle}, \omega) \in [\![\langle\alpha\rangle\langle\alpha^*\rangle\phi]\!]$$

so that by the semantics of Angelic loops have (Looped)

$$(\{(\mathsf{ind}(\mathsf{b},\,\mathsf{c},\,\mathsf{d})_{\mathcal{M}})\}, \omega) \in [\![\langle\alpha^*\rangle\phi]\!]$$

To complete the inductive case, recall (Loop) which by (Looped) immediately gives $(\hat{\mathsf{rz}}, \mu) \in \phi$ as desired. This completes the induction and the Angel claim. The structure of the inductive proof may be surprising because the inductive case proves $\langle\alpha^*\rangle\phi$ (Looped) and does so by assuming an (IH) about $\langle\alpha^*\rangle\phi$. Note however that we do not assume the conclusion in the process: the induction is on the ordering of the *initial* state under metric $\mathcal{M}$. We assume $\langle\alpha^*\rangle\phi$ holds starting from states with a lower metric value and show it also holds for states with a higher metric value, so that by induction, it holds for all states satisfying the preconditions of the lemma.

Demon cases:

Call the assumptions (1), (2), and (3).

By the semantics of loops it suffices to assume arbitrary poss such that (Loop) $\mathsf{poss} \in \{(\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d}), \omega)\}[\![\alpha^*]\!]$ and prove $\mathsf{poss} \in [\![\phi]\!] \cup \{\top\}$.

The proof is by coinduction, allowing (b, ω) to vary to other $(\hat{\mathsf{b}}, \nu)$ satisfying assumption (1) $(\hat{\mathsf{b}}, \nu) \in [\![\psi]\!]$ so long as the application of the coinductive application is guarded by an iteration of α, i.e., the guard requirement is that $(\hat{\mathsf{c}}, \nu) \in \{((\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d}))_{[1]}, \omega)\}[\![\alpha]\!]$ for some[2] $\hat{\mathsf{c}}$. While the proof is coinductive, we will also make use of the iterative semantics (ItSem) (by Lemma 4.8):

$$\{(\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d}), \omega)\}[\![\alpha^*]\!] = \bigcup_{k\in\mathbb{N}} (\{(\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d}), \omega)\}[\![\alpha]\!]^k)_{[0]}$$

[2]While the coinduction principle would be clearer if $\hat{\mathsf{b}}$ were used in place of the existentially-quantified $\hat{\mathsf{c}}$ the monotonicity realizer lemma is stated too weakly to do so. In contrast to other proofs about Demon loops (e.g., Lemma 4.7) we use coinduction rather than induction because we vary initial states rather than final states. *Initial* regions satisfying a goal region are coinductive in nature: show that goal holds initially and show that prefixing an additional loop iteration preserves satisfaction of the postcondition. Also, $\cdot_{[1]}$ expresses Demon's decision to continue loop execution.

in the coinductive proof.

Within the single case of the coinductive proof, we proceed by cases on k from (ItSem).

In the first case, poss belongs to the $k = 0$ term. Then

$$\begin{aligned} \mathsf{poss} &\in \{(\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d}), \omega)\}[\![\alpha]\!]^0_{[0]} \\ \text{thus } \mathsf{poss} &\in \{(\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d}), \omega)\}_{[0]} \\ \text{thus } \mathsf{poss} &\in \{(\pi_1(\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})), \omega)\} \\ \text{thus } \mathsf{poss} &= (\mathsf{d}\ \mathsf{b}, \omega) \end{aligned}$$

by construction. Then (BC) $(\mathsf{b}, \omega) \in [\![\psi]\!]$ by (1) and $(\mathsf{d}\ \mathsf{b}, \omega) \in [\![\phi]\!]$ by (3) as desired since $\mathsf{poss} = (\mathsf{d}\ \mathsf{b}, \omega)$.

In the second case, poss belongs to the $k = i + 1$ term of the semantic union (ItSem) for some $i \in \mathbb{N}$. Then

$$\begin{aligned} \mathsf{poss} &\in \{(\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d}), \omega)\}[\![\alpha]\!]^{i+1}_{[0]} \\ \text{thus } \mathsf{poss} &\in \{(\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d}), \omega)\}[\![\alpha]\!]^{i}_{[1]}[\![\alpha]\!]_{[0]} \\ \text{thus } \mathsf{poss} &\in \{(\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d}), \omega)\}_{[1]}[\![\alpha]\!][\![\alpha]\!]^{i}_{[0]} \end{aligned}$$

by Remark 4.1, call this fact (sem).

To minimize the number of cases, note by (2) that $\bot \notin \{(\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d}), \omega)\}_{[1]}[\![\alpha]\!]$ since $\bot \notin [\![\psi]\!] \cup \{\top\}$. Combined with a likewise argument for the co-IH and for the $\cdot_{[0]}$ operation, we have $\bot \notin \{(\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d}), \omega)\}[\![\alpha]\!]^{i+1}_{[0]}$ so that $\mathsf{poss} \neq \bot$. Secondly, note the case $\mathsf{poss} = \top$ trivially satisfies $\mathsf{poss} \in [\![\phi]\!] \cup \{\top\}$. Thus, it only remains to show the case where poss has form $(\hat{\mathsf{rz}}, \mu)$ for some realizer $\hat{\mathsf{rz}}$ and state μ.

We have (4) $\{(\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d}), \omega)\}_{[1]} = \{(\mathsf{mon}_{[\alpha]}\ (\Lambda \mathsf{rz} : \psi\, \mathcal{R}\mathbf{z}.\ \mathsf{gen}(\mathsf{rz},\ \mathsf{c},\ \mathsf{d}))\ (\mathsf{c}\ \mathsf{b}), \omega)\}$ by construction. We proceed to analyze (4) by applying Lemma A.2, whose assumptions we must first prove. In the application, let $R = \{((\mathsf{c}\ \mathsf{b}), \omega)\}[\![\alpha]\!]$ be the final region resulting from executing the first iteration of α in (4). Let $\hat{R}$ be the set of states in R. The preconditions of Lemma A.2 we need to show are (PC1)

$$\{(\Lambda \mathsf{rz} : \psi\, \mathcal{R}\mathbf{z}.\ \mathsf{gen}(\mathsf{rz},\ \mathsf{c},\ \mathsf{d}))\} \times \hat{R} \subseteq [\![(\psi \to [\alpha^*]\phi)]\!]$$

and (PC2) $((\mathsf{c}\ \mathsf{b}), \omega) \in [\![[\alpha]\psi]\!]$ which will result in (Mon):

$$(\mathsf{mon}_{[\alpha]}\ (\Lambda \mathsf{rz} : \psi\, \mathcal{R}\mathbf{z}.\ \mathsf{gen}(\mathsf{rz},\ \mathsf{c},\ \mathsf{d}))\ (\mathsf{c}\ \mathsf{b}), \omega) \in [\![[\alpha][\alpha^*]\phi]\!]$$

Precondition (PC2) holds immediately by assumptions (1) and (2). To show (PC1), fix arbitrary (rz, ν) such that $\nu \in \hat{R}$ and $(\mathsf{rz}, \nu) \in [\![\psi]\!]$ and show (PC1Simp) $(\mathsf{gen}(\mathsf{rz},\ \mathsf{c},\ \mathsf{d}), \nu) \in [\![[\alpha^*]\phi]\!]$. By varying b to rz and ω to ν, precondition (PC1Simp) is immediate from the coinductive hypothesis. The coinductive hypothesis is applicable because the guardedness condition $((\hat{\mathsf{c}}, \nu) \in \{((\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d}))_{[1]}, \omega)\}[\![\alpha]\!]$ for some $\hat{\mathsf{c}}$) is satisfied. It is satisfied by construction of $\hat{R}$ and because $\{((\mathsf{c}\ \mathsf{b}), \omega)\}[\![\alpha]\!]$ contains the same states as $\{((\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d}))_{[1]}, \omega)\}[\![\alpha]\!]$ by definition of $\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d})$ and by definition of $\mathsf{mon}_{[\alpha]}\ (\Lambda \mathsf{rz} : \psi\, \mathcal{R}\mathbf{z}.\ \mathsf{gen}(\mathsf{rz},\ \mathsf{c},\ \mathsf{d}))\ (\mathsf{c}\ \mathsf{b})$.

The application of Lemma A.2 is now complete, resulting in (Mon):

$$(\mathsf{mon}_{[\alpha]}\ (\Lambda \mathsf{rz} : \psi\, \mathcal{R}\mathbf{z}.\ \mathsf{gen}(\mathsf{rz},\ \mathsf{c},\ \mathsf{d}))\ (\mathsf{c}\ \mathsf{b}), \omega) \in [\![[\alpha][\alpha^*]\phi]\!]$$

To finish the proof, combine (Mon) with (sem). Recall that fact (sem) says

$$\mathsf{poss} \in \{(\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d}), \omega)\}_{[1]}[\![[\alpha]]\!][\![[\alpha]]\!]^i_{[0]}$$

which implies $\mathsf{poss} \in \{\{(\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d}), \omega)\}_{[1]}[\![[\alpha]]\!]\}[\![[\alpha^*]]\!]$ as a result of Lemma 4.8. Then, since $(\mathsf{gen}(\mathsf{b},\ \mathsf{c},\ \mathsf{d}))_{[1]} = (\mathsf{mon}_{[\alpha]}\ (\Lambda \mathsf{rz} : \psi\, \mathcal{R}\mathbf{z}.\ \mathsf{gen}(\mathsf{rz},\ \mathsf{c},\ \mathsf{d}))\ (\mathsf{c}\ \mathsf{b})$, we have $\mathsf{poss} \in [\![\phi]\!]$ as a result of (Mon). This completes the coinduction and completes the proof. □

A.4.6 Soundness

Let $e\frac{y}{x}$ denote the (transposition) renaming of x for y (and vice versa) in e, (where e is term f formula ϕ, game α, or realizer b).

Lemma A.4 (Renaming is Self-dual)**.** *Renaming the same variables twice cancels, because renaming is by transposition.*

$$f\tfrac{y}{x}\tfrac{y}{x} = f \qquad \mathsf{b}\tfrac{y}{x}\tfrac{y}{x} = \mathsf{b} \qquad \phi\tfrac{y}{x}\tfrac{y}{x} = \phi \qquad \alpha\tfrac{y}{x}\tfrac{y}{x} = \alpha \qquad \omega\tfrac{y}{x}\tfrac{y}{x} = \omega$$

Proof. The first claim holds by induction on the term syntax. The second claim is by coinduction because realizers are defined coinductively. The next two claims are by simultaneous induction. The last claim is direct. □

Lemma A.5 (Renaming)**.** *Renaming commutes with the interpretation and projection functions. The claim for realizers expresses the case of closed realizers, i.e., those where no realizer variables are free. The proof for closed realizers will proceed by strengthening the claim to open realizers.*

- $[\![f\frac{y}{x}]\!]\omega = [\![f]\!]\omega\frac{y}{x}$
- $[\![\phi\frac{y}{x}]\!] = \{(\mathsf{b}\frac{y}{x}, \omega\frac{y}{x}) \mid (\mathsf{b}, \omega) \in [\![\phi]\!]\}$
- $X\frac{y}{x}\langle\!\langle\alpha\frac{y}{x}\rangle\!\rangle = (X\langle\!\langle\alpha\rangle\!\rangle)\frac{y}{x}$ *and* $X\frac{y}{x}[\![\alpha\frac{y}{x}]\!] = (X[\![\alpha]\!])\frac{y}{x}$
- $[\![\mathsf{b}\frac{y}{x}]\!]\omega = [\![\mathsf{b}]\!]\omega\frac{y}{x}$
- $(Z\frac{y}{x})_{\langle 0\rangle} = (Z_{\langle 0\rangle})\frac{y}{x}$ *and* $(Z\frac{y}{x})_{\langle 1\rangle} = (Z_{\langle 1\rangle})\frac{y}{x}$
- $(Z\frac{y}{x})_{[0]} = (Z_{[0]})\frac{y}{x}$ *and* $(Z\frac{y}{x})_{[1]} = (Z_{[1]})\frac{y}{x}$

Proof. The term cases hold by induction on the term syntax. The formula and game cases are proved by simultaneous induction on formulas and programs. The cases for realizers are proved by a separate coinduction.

Term cases: The cases for the binary term connectives all map through homomorphically and all symmetric to one another, but we list each case for the sake of thoroughness.

Case q: Have $[\![q\frac{y}{x}]\!]\omega = [\![q]\!]\omega = q = [\![q]\!]\omega\frac{y}{x}$.

Case z: Have $[\![z\frac{y}{x}]\!]\omega = \omega(z\frac{y}{x}) = \omega\frac{y}{x}(z) = [\![z]\!]\omega\frac{y}{x}$.

Case $f + g$: Have $[\![(f+g)\frac{y}{x}]\!]\omega = [\![f\frac{y}{x}]\!]\omega + [\![g\frac{y}{x}]\!]\omega = [\![f]\!]\omega\frac{y}{x} + [\![g]\!]\omega\frac{y}{x} = [\![f+g]\!]\omega\frac{y}{x}$.

Case $f \cdot g$: Have $[\![(f\cdot g)\frac{y}{x}]\!]\omega = [\![f\frac{y}{x}]\!]\omega \cdot [\![g\frac{y}{x}]\!]\omega = [\![f]\!]\omega\frac{y}{x} \cdot [\![g]\!]\omega\frac{y}{x} = [\![f\cdot g]\!]\omega\frac{y}{x}$.

Case f mod g: We have that $[\![(f \text{ mod } g)\frac{y}{x}]\!]\omega = [\![f\frac{y}{x}]\!]\omega \text{ mod } [\![g\frac{y}{x}]\!]\omega = [\![f]\!]\omega\frac{y}{x} \text{ mod } [\![g]\!]\omega\frac{y}{x} = [\![f \text{ mod } g]\!]\omega\frac{y}{x}$.

Case f div g: Have $[\![(f \text{ div } g)\frac{y}{x}]\!]\omega = [\![f\frac{y}{x}]\!]\omega \text{ div } [\![g\frac{y}{x}]\!]\omega = [\![f]\!]\omega\frac{y}{x} \text{ div } [\![g]\!]\omega\frac{y}{x} = [\![f \text{ div } g]\!]\omega\frac{y}{x}$.

Formula cases:

Case $f > g$: Have $[\![(f > g)\frac{y}{x}]\!] = \{(\epsilon, \omega) \mid [\![f\frac{y}{x}]\!]\omega > [\![g\frac{y}{x}]\!]\omega\} = \{(\epsilon, \omega) \mid [\![f]\!]\omega\frac{y}{x} > [\![g]\!]\omega\frac{y}{x}\} = \{(\epsilon, \omega\frac{y}{x}) \mid [\![f]\!]\omega > [\![g]\!]\omega\} = [\![f > g]\!]\frac{y}{x}$. The cases for $<, \leq, =, \neq \geq$ are symmetric.

Case $\langle\alpha\rangle\phi$: Have

$$\begin{aligned}
&[\![(\langle\alpha\rangle\phi)\tfrac{y}{x}]\!] \\
&= \{(\mathsf{b}, \omega) \mid \{(\mathsf{b}, \omega)\}\langle\!\langle \alpha\tfrac{y}{x}\rangle\!\rangle \subseteq [\![\phi\tfrac{y}{x}]\!] \cup \{\top\}\} \\
&= \{(\mathsf{b}, \omega) \mid \{(\mathsf{b}\tfrac{y}{x}, \omega\tfrac{y}{x})\}\langle\!\langle \alpha\rangle\!\rangle\tfrac{y}{x} \subseteq [\![\phi]\!]\tfrac{y}{x} \cup \{\top\}\} \\
&= \{(\mathsf{b}\tfrac{y}{x}, \omega\tfrac{y}{x}) \mid \{(\mathsf{b}, \omega)\}\langle\!\langle \alpha\rangle\!\rangle\tfrac{y}{x} \subseteq [\![\phi]\!]\tfrac{y}{x} \cup \{\top\}\} \\
&= \{(\mathsf{b}\tfrac{y}{x}, \omega\tfrac{y}{x}) \mid \{(\mathsf{b}, \omega)\}\langle\!\langle \alpha\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\}\} \\
&= [\![\langle\alpha\rangle\phi]\!]\tfrac{y}{x}
\end{aligned}$$

Case $[\alpha]\phi$: Have

$$\begin{aligned}
&[\![([\alpha]\phi)\tfrac{y}{x}]\!] \\
&= \{(\mathsf{b}, \omega) \mid \{(\mathsf{b}, \omega)\}[\![\alpha\tfrac{y}{x}]\!] \subseteq [\![\phi\tfrac{y}{x}]\!] \cup \{\top\}\} \\
&= \{(\mathsf{b}, \omega) \mid \{(\mathsf{b}\tfrac{y}{x}, \omega\tfrac{y}{x})\}[\![\alpha]\!]\tfrac{y}{x} \subseteq [\![\phi]\!]\tfrac{y}{x} \cup \{\top\}\} \\
&= \{(\mathsf{b}\tfrac{y}{x}, \omega\tfrac{y}{x}) \mid \{(\mathsf{b}, \omega)\}[\![\alpha]\!]\tfrac{y}{x} \subseteq [\![\phi]\!]\tfrac{y}{x} \cup \{\top\}\} \\
&= \{(\mathsf{b}\tfrac{y}{x}, \omega\tfrac{y}{x}) \mid \{(\mathsf{b}, \omega)\}[\![\alpha]\!] \subseteq [\![\phi]\!] \cup \{\top\}\} \\
&= [\![[\alpha]\phi]\!]\tfrac{y}{x}
\end{aligned}$$

Angel cases:

Case $x := f$: Have $X\frac{y}{x}\langle\!\langle\{x := f\}\frac{y}{x}\rangle\!\rangle = X\frac{y}{x}\langle\!\langle y := (f\frac{y}{x})\rangle\!\rangle = \{(\mathsf{b}, \omega[y \mapsto [\![f\frac{y}{x}]\!]\omega]) \mid (\mathsf{b}, \omega) \in X\frac{y}{x}\} = \{(\mathsf{b}, \omega[y \mapsto [\![f]\!]\omega\frac{y}{x}]) \mid (\mathsf{b}, \omega) \in X\frac{y}{x}\} = \{(\mathsf{b}\frac{y}{x}, \omega[x \mapsto [\![f]\!]\omega]) \mid (\mathsf{b}, \omega) \in X\} = (X\langle\!\langle x := f\rangle\!\rangle)\frac{y}{x}$.

Case $x := *$: Have

$$\begin{aligned}
&X\tfrac{y}{x}\langle\!\langle\{x := *\}\tfrac{y}{x}\rangle\!\rangle \\
&= X\tfrac{y}{x}\langle\!\langle y := *\rangle\!\rangle \\
&= \{(\pi_1\mathsf{b}, \omega[y \mapsto [\![(\pi_0\mathsf{b})\frac{y}{x}]\!]\omega]) \mid (\mathsf{b}, \omega) \in X\tfrac{y}{x}\} \\
&= \{(\pi_1\mathsf{b}, \omega[y \mapsto [\![\pi_0\mathsf{b}]\!]\omega\tfrac{y}{x}]) \mid (\mathsf{b}, \omega) \in X\tfrac{y}{x}\} \\
&= \{(\pi_1\mathsf{b}\tfrac{y}{x}, \omega[x \mapsto [\![\pi_0\mathsf{b}]\!]\omega[x \mapsto y]]\tfrac{y}{x}) \mid (\mathsf{b}, \omega) \in X\} \\
&= (X\langle\!\langle x := *\rangle\!\rangle)\tfrac{y}{x}.
\end{aligned}$$

Case $?\phi$: Have

$$\begin{aligned}
&X\tfrac{y}{x}\langle\!\langle\{?\phi\}\tfrac{y}{x}\rangle\!\rangle\\
=\ &X\tfrac{y}{x}\langle\!\langle?(\phi\tfrac{y}{x})\rangle\!\rangle\\
=\ &\{(\pi_1\mathsf{b},\omega)\mid(\mathsf{b},\omega)\in X\tfrac{y}{x},(\pi_0\mathsf{b},\omega)\in[\![\phi\tfrac{y}{x}]\!]\}\\
=\ &\{(\pi_1\mathsf{b},\omega)\mid(\mathsf{b},\omega)\in X\tfrac{y}{x},(\pi_0\mathsf{b}\tfrac{y}{x},\omega\tfrac{y}{x})\in[\![\phi]\!]\}\\
=\ &\{(\pi_1\mathsf{b},\omega)\mid(\mathsf{b}\tfrac{y}{x},\omega\tfrac{y}{x})\in X,(\pi_0\mathsf{b}\tfrac{y}{x},\omega\tfrac{y}{x})\in[\![\phi]\!]\}\\
=\ &\{(\pi_1\mathsf{b}\tfrac{y}{x},\omega\tfrac{y}{x})\mid(\mathsf{b},\omega)\in X,(\pi_0\mathsf{b},\omega)\in[\![\phi]\!]\}\\
=\ &(X\langle\!\langle?\phi\rangle\!\rangle)\tfrac{y}{x}
\end{aligned}$$

Case $\alpha\cup\beta$: Have $X\frac{y}{x}\langle\!\langle\{\alpha\cup\beta\}\frac{y}{x}\rangle\!\rangle = X_{\langle0\rangle}\frac{y}{x}\langle\!\langle\alpha\frac{y}{x}\rangle\!\rangle\cup X_{\langle1\rangle}\frac{y}{x}\langle\!\langle\beta\frac{y}{x}\rangle\!\rangle = (X_{\langle0\rangle}\langle\!\langle\alpha\rangle\!\rangle)\frac{y}{x}\cup(X_{\langle1\rangle}\langle\!\langle\beta\rangle\!\rangle)\frac{y}{x} = (X\langle\!\langle\alpha\cup\beta\rangle\!\rangle)\frac{y}{x}$.

Case $\alpha;\beta$: By transitivity, $X\frac{y}{x}\langle\!\langle\{\alpha;\beta\}\frac{y}{x}\rangle\!\rangle = X\frac{y}{x}\langle\!\langle\{\alpha\frac{y}{x}\};\{\beta\frac{y}{x}\}\rangle\!\rangle = (X\frac{y}{x}\langle\!\langle\alpha\frac{y}{x}\rangle\!\rangle)\langle\!\langle\beta\frac{y}{x}\rangle\!\rangle = (X\langle\!\langle\alpha\rangle\!\rangle)\frac{y}{x}\langle\!\langle\beta\frac{y}{x}\rangle\!\rangle = ((X\langle\!\langle\alpha\rangle\!\rangle)\langle\!\langle\beta\rangle\!\rangle)\frac{y}{x} = (X\langle\!\langle\alpha;\beta\rangle\!\rangle)\frac{y}{x}$.

Case α^*: Have

$$\begin{aligned}
&X\tfrac{y}{x}\langle\!\langle\{\alpha\tfrac{y}{x}\}^*\rangle\!\rangle\\
=\ &\bigcap\{Z_{\langle0\rangle}\mid X\tfrac{y}{x}\cup Z_{\langle1\rangle}\langle\!\langle\alpha\tfrac{y}{x}\rangle\!\rangle\subseteq Z\}\\
=\ &\bigcap\{Z_{\langle0\rangle}\mid X\tfrac{y}{x}\cup Z_{\langle1\rangle}\tfrac{y}{x}\langle\!\langle\alpha\rangle\!\rangle\tfrac{y}{x}\subseteq Z\}\\
=\ &\bigcap\{Z_{\langle0\rangle}\mid X\cup Z_{\langle1\rangle}\tfrac{y}{x}\langle\!\langle\alpha\rangle\!\rangle\subseteq Z\tfrac{y}{x}\}\\
=\ &\bigcap\{Z_{\langle0\rangle}\tfrac{y}{x}\mid X\cup Z_{\langle1\rangle}\langle\!\langle\alpha\rangle\!\rangle\subseteq Z\}\\
=\ &\bigcap\{(Z\tfrac{y}{x})_{\langle0\rangle}\mid X\cup Z_{\langle1\rangle}\langle\!\langle\alpha\rangle\!\rangle\subseteq Z\}\\
=\ &(X\langle\!\langle\alpha^*\rangle\!\rangle)\tfrac{y}{x}.
\end{aligned}$$

Case α^d: Have $X\frac{y}{x}\langle\!\langle\{\alpha\frac{y}{x}\}^d\rangle\!\rangle = X\frac{y}{x}[\![\alpha\frac{y}{x}]\!] = (X[\![\alpha]\!])\frac{y}{x} = (X\langle\!\langle\alpha^d\rangle\!\rangle)\frac{y}{x}$.

Demon cases:

Case $x:=f$: Have $X\frac{y}{x}[\![\{x:=f\}\frac{y}{x}]\!] = X\frac{y}{x}[\![y:=(f\frac{y}{x})]\!] = \{(\mathsf{b},\omega[y\mapsto[\![f\frac{y}{x}]\!]\omega])\mid(\mathsf{b},\omega)\in X\frac{y}{x}\} = \{(\mathsf{b},\omega[y\mapsto[\![f]\!]\omega\frac{y}{x}])\mid(\mathsf{b},\omega)\in X\frac{y}{x}\} = \{(\mathsf{b}\frac{y}{x},\omega\frac{y}{x}[y\mapsto[\![f]\!]\omega])\mid(\mathsf{b},\omega)\in X\} = \{(\mathsf{b}\frac{y}{x},\omega[x\mapsto[\![f]\!]\omega]\frac{y}{x})\mid(\mathsf{b},\omega)\in X\} = (X[\![x:=f]\!])\frac{y}{x}$.

Case $x:=*$: Have $X\frac{y}{x}[\![\{x:=*\}\frac{y}{x}]\!] = X\frac{y}{x}[\![y:=*]\!] = \{\omega[y\mapsto r]\mid\omega\in X\frac{y}{x},\text{ exists } r\in\mathbb{Q}\} = \{\omega[x\mapsto r]\frac{y}{x}\mid\omega\in X,\text{ exists } r\in\mathbb{Q}\} = (X[\![x:=*]\!])\frac{y}{x}$.

Case $?\phi$: Have $X\frac{y}{x}[\![\{?\phi\}\frac{y}{x}]\!] = X\frac{y}{x}[\![?(\phi\frac{y}{x})]\!] = \{(\mathsf{b}\,\mathsf{c},\omega)\mid(\mathsf{b},\omega)\in X\frac{y}{x},(\mathsf{c},\omega)\in[\![\phi\frac{y}{x}]\!]\} = \{(\mathsf{b}\,\mathsf{c},\omega)\mid(\mathsf{b}\frac{y}{x},\omega\frac{y}{x})\in X,(\mathsf{c}\frac{y}{x},\omega\frac{y}{x})\in[\![\phi]\!]\} = \{(\mathsf{b}\,\mathsf{c}\frac{y}{x},\omega\frac{y}{x})\mid(\mathsf{b},\omega)\in X,(\mathsf{c},\omega)\in[\![\phi]\!]\} = (X[\![?\phi]\!])\frac{y}{x}$.

Case $\alpha\cup\beta$: By transitivity, have $X\frac{y}{x}[\![\{\alpha\cup\beta\}\frac{y}{x}]\!] = (X\frac{y}{x})_{[0]}[\![\alpha\frac{y}{x}]\!]\cup(X\frac{y}{x})_{[1]}[\![\beta\frac{y}{x}]\!] = X_{[0]}\frac{y}{x}[\![\alpha\frac{y}{x}]\!]\cup X_{[1]}\frac{y}{x}[\![\beta\frac{y}{x}]\!] = (X_{[0]}[\![\alpha]\!])\frac{y}{x}\cup(X_{[1]}[\![\beta]\!])\frac{y}{x} = (X_{[0]}[\![\alpha]\!]\cup X_{[1]}[\![\beta]\!])\frac{y}{x} = (X[\![\alpha\cup\beta]\!])\frac{y}{x}$.

Case $\alpha;\beta$: By transitivity, have $X\frac{y}{x}[\![\{\alpha;\beta\}\frac{y}{x}]\!] = X\frac{y}{x}[\![\{\alpha\frac{y}{x}\};\{\beta\frac{y}{x}\}]\!] = (X\frac{y}{x}[\![\alpha\frac{y}{x}]\!])[\![\beta\frac{y}{x}]\!] = ((X[\![\alpha]\!])\frac{y}{x})[\![\beta\frac{y}{x}]\!] = (X[\![\alpha]\!][\![\beta]\!])\frac{y}{x} = (X[\![\alpha;\beta]\!])\frac{y}{x}$.

Case α^*: Have

$$\begin{aligned}
&X\tfrac{y}{x}[\![\{\alpha\tfrac{y}{x}\}^*]\!]\\
=&\bigcap\{Z_{[0]} \mid X\tfrac{y}{x}\cup Z_{[1]}[\![\alpha\tfrac{y}{x}]\!]\subseteq Z\}\\
=&\bigcap\{Z_{[0]} \mid X\tfrac{y}{x}\cup Z_{[1]}\tfrac{y}{x}[\![\alpha]\!]\tfrac{y}{x}\subseteq Z\}\\
=&\bigcap\{Z_{[0]} \mid X\cup Z_{[1]}\tfrac{y}{x}[\![\alpha]\!]\subseteq Z\tfrac{y}{x}\}\\
=&\bigcap\{Z_{[0]}\tfrac{y}{x} \mid X\cup Z_{[1]}[\![\alpha]\!]\subseteq Z\}\\
=&\bigcap\{(Z\tfrac{y}{x})_{[0]} \mid X\cup Z_{[1]}[\![\alpha]\!]\subseteq Z\}\\
=&(X[\![\alpha^*]\!])\tfrac{y}{x}
\end{aligned}$$

Case α^d: $X\frac{y}{x}[\![\{\alpha\frac{y}{x}\}^d]\!] = X\frac{y}{x}\langle\!\langle\alpha\frac{y}{x}\rangle\!\rangle = X\langle\!\langle\alpha\rangle\!\rangle\frac{y}{x} = X[\![\alpha^d]\!]\frac{y}{x}$.

We now prove the claim for realizers, and in order to do that, we strengthen the claim: in order to reason about realizers which contain free realizer variables, we introduce a *realizer substitution* ξ which binds each free realizer variable to a closed realizer. We write $Dom(\xi)$ for the set of realizer variables z substituted by ξ and write $\xi\, z$ for each replacement. The application $\xi(\mathsf{b})$ is equivalent to b except that every occurrence of each realizer variable $z \in Dom(\xi)$ is replaced by $\xi\, z$. When we need to distinguish ξ from σ, we call σ a *program variable substitution* to distinguish it from a realizer substitution. We prove the strengthened claim which says:

Claim 1. Let ξ be a realizer substitution such that $\xi(z)$ is closed for all free realizer variables z of b. Assume $[\![\xi(z)\frac{y}{x}]\!]\omega = [\![\xi(z)]\!]\omega\frac{y}{x}$ for all $z \in Dom(\xi)$. Then $[\![\xi(\mathsf{b})\frac{y}{x}]\!]\omega = [\![\xi(\mathsf{b})]\!]\omega\frac{y}{x}$.

The renaming lemma for closed realizers follows directly from the strengthened claim by letting ξ be the empty substitution (which we choose to write as $\{\}$) so that $\xi(\mathsf{b}) = \mathsf{b}$ and the assumptions on ξ are trivially satisfied. The strengthening is necessary because closed realizers will contain realizer variables within them; we reason about those variable by strengthening ξ in cases that bind realizer variables.

We prove the strengthened claim by coinduction on realizers and by induction on the realized game, i.e., we can assume the claim holds for all realizers of smaller games, even if those realizers are not smaller.

Case z: Then $[\![\xi(z)\frac{y}{x}]\!]\omega = [\![\xi(z)]\!]\xi(z)\frac{y}{x}$ by assumption.

Case f: Because terms do not mention realizer variables, have $[\![\xi(f)\frac{y}{x}]\!]\omega = [\![f\frac{y}{x}]\!]\omega = [\![f]\!]\omega\frac{y}{x} = [\![\xi(f)]\!]\omega\frac{y}{x}$ by the claim for terms.

Case $\pi_0\mathsf{b}$: Have $[\![(\xi(\pi_0\mathsf{b}))\frac{y}{x}]\!]\omega = [\![\pi_0(\xi(\mathsf{b})\frac{y}{x})]\!]\omega = \pi_0([\![\xi(\mathsf{b})\frac{y}{x}]\!]\omega) = \pi_0([\![\xi(\mathsf{b})]\!]\omega\frac{y}{x}) = [\![\pi_0\xi(\mathsf{b})]\!]\omega\frac{y}{x}$.

Case $\pi_1\mathsf{b}$: Have $[\![(\xi(\pi_1\mathsf{b}))\frac{y}{x}]\!]\omega = [\![\pi_1(\xi(\mathsf{b})\frac{y}{x})]\!]\omega = \pi_1([\![\xi(\mathsf{b})\frac{y}{x}]\!]\omega) = \pi_1([\![\xi(\mathsf{b})]\!]\omega\frac{y}{x}) = [\![\pi_1\xi(\mathsf{b})]\!]\omega\frac{y}{x}$.

Case ϵ: Have $[\![\xi(\epsilon)\frac{y}{x}]\!]\omega = [\![\epsilon\frac{y}{x}]\!]\omega = [\![\epsilon]\!]\omega = () = [\![\epsilon]\!]\omega\frac{y}{x} = [\![\xi(\epsilon)]\!]\omega\frac{y}{x}$.

Case (b,c): We transitivity: $[\![\xi((\mathsf{b},\mathsf{c}))\frac{y}{x}]\!]\omega = [\![(\xi(\mathsf{b}),\xi(\mathsf{c}))\frac{y}{x}]\!]\omega = [\![(\xi(\mathsf{b})\frac{y}{x},\xi(\mathsf{c})\frac{y}{x})]\!]\omega = ([\![\xi(\mathsf{b})\frac{y}{x}]\!]\omega, [\![\xi(\mathsf{c})\frac{y}{x}]\!]\omega) = ([\![\xi(\mathsf{b})]\!]\omega\frac{y}{x}, [\![\xi(\mathsf{c})]\!]\omega\frac{y}{x}) = [\![(\xi(\mathsf{b}),\xi(\mathsf{c}))]\!]\omega\frac{y}{x} = [\![\xi((\mathsf{b},\mathsf{c}))]\!]\omega\frac{y}{x}$ where b is an arbitrary realizer, not necessarily a term f.

Case $(\Lambda z : \mathbb{Q}.\ \mathsf{b})$: By transitivity, have $[\![\xi(\Lambda z : \mathbb{Q}.\ \mathsf{b})\frac{y}{x}]\!]\omega = [\![(\Lambda z : \mathbb{Q}.\ \xi(\mathsf{b}))\frac{y}{x}]\!]\omega = [\![\Lambda z\frac{y}{x} : \mathbb{Q}.\ \xi(\mathsf{b})\frac{y}{x}]\!]\omega = (\Lambda w : \mathbb{Q}.\ [\![\xi(\mathsf{b})\frac{y}{x}]\!]\omega[z\frac{y}{x} \mapsto w]) = (\Lambda w : \mathbb{Q}.\ [\![\xi(\mathsf{b})]\!](\omega[z\frac{y}{x} \mapsto w])\frac{y}{x} = (\Lambda w : \mathbb{Q}.\ [\![\xi(\mathsf{b})]\!]\omega\frac{y}{x}[z \mapsto w]) = [\![\Lambda z : \mathbb{Q}.\ \xi(\mathsf{b})]\!]\omega\frac{y}{x}$ where w is a fresh mathematical variable standing for the function argument.

Case $(\Lambda z : \phi\,\mathcal{R}\mathbf{z}.\ \mathsf{b})$: The case is proved by extensionality. We give the case as a chain of equalities first, then justify each step. The argument for this case is $[\![\xi(\Lambda z : \phi\,\mathcal{R}\mathbf{z}.\ \mathsf{b})\frac{y}{x}]\!]\omega = [\![\Lambda z : \phi\,\mathcal{R}\mathbf{z}.\ \xi(\mathsf{b})\frac{y}{x}]\!]\omega = [\![\Lambda z : \phi\frac{y}{x}\,\mathcal{R}\mathbf{z}.\ \xi(\mathsf{b})\frac{y}{x}]\!]\omega = (\Lambda w : \phi\frac{y}{x}\,\mathcal{R}\mathbf{z}.\ [\![\xi(\mathsf{b})\frac{y}{x}[z \mapsto w]]\!]\omega) =_{(*)} (\Lambda w : \phi\,\mathcal{R}\mathbf{z}.\ [\![\xi'(\mathsf{b})\frac{y}{x}]\!]\omega) \underset{\text{IH}}{=} (\Lambda w : \phi\,\mathcal{R}\mathbf{z}.\ [\![\xi'(\mathsf{b})]\!]\omega\frac{y}{x}) = [\![\xi(\Lambda z : \phi\,\mathcal{R}\mathbf{z}.\ \mathsf{b})]\!]\omega\frac{y}{x}$, where substitution $\xi' = \xi[z \mapsto w]$ extends ξ by assigning argument value w to realizer parameter variable z. The proof begins by simplifying ξ and $\cdot\frac{y}{x}$ homomorphically. The denotation of a function realizer is a function returning denotations, so the equality of two denotations is shown extensionally by showing the bodies agree for every argument w. The step marked (*) says the substitution $[z \mapsto w]$ can be moved inside the renaming $\cdot\frac{y}{x}$ by renaming x and y in the type of w. The crucial step is showing that the (co)-IH is applicable. The co-IH is well-founded because it is guarded by a constructor (a λ-abstraction). To satisfy the assumption of the co-IH we must show that the renaming lemma applies to the realizer w. Because any appearance of w would be guarded by the same λ-abstraction, one could attempt a well-foundedness argument, but there is a simpler argument: The type of w is structurally smaller than the type of $(\Lambda z : \phi\,\mathcal{R}\mathbf{z}.\ \mathsf{b})$, so the renaming lemma can be assumed for w by an outer structural induction on types.

Case $(\mathsf{b}\ f)$: By transitivity, have $[\![\xi(\mathsf{b}\ f)\frac{y}{x}]\!]\omega = [\![\xi(\mathsf{b})\frac{y}{x}\ \xi(f)\frac{y}{x}]\!]\omega = [\![\xi(\mathsf{b})\frac{y}{x}\ f\frac{y}{x}]\!]\omega = [\![\xi(\mathsf{b})\frac{y}{x}]\!]\omega\ [\![f\frac{y}{x}]\!]\omega = [\![\xi(\mathsf{b})]\!]\omega\frac{y}{x}\ [\![f]\!]\omega\frac{y}{x} = [\![\xi(\mathsf{b})\ f]\!]\omega\frac{y}{x} = [\![\xi(\mathsf{b}\ f)]\!]\omega\frac{y}{x}$.

Case $(\mathsf{b}\ \mathsf{c})$: By transitivity, have $[\![\xi(\mathsf{b}\ \mathsf{c})\frac{y}{x}]\!]\omega = [\![\xi(\mathsf{b})\frac{y}{x}\ \xi(\mathsf{c})\frac{y}{x}]\!]\omega = [\![\xi(\mathsf{b})\frac{y}{x}]\!]\omega\ [\![\xi(\mathsf{c})\frac{y}{x}]\!]\omega = [\![\xi(\mathsf{b})]\!]\omega\frac{y}{x}\ [\![\xi(\mathsf{c})]\!]\omega\frac{y}{x} = [\![\xi(\mathsf{b})\ \xi(\mathsf{c})]\!]\omega\frac{y}{x} = [\![\xi(\mathsf{b}\ \mathsf{c})]\!]\omega\frac{y}{x}$.

Case $(\mathtt{if}\ (f)\ \mathsf{b}\ \mathtt{else}\ \mathsf{c})$:

$$
\begin{aligned}
&[\![\xi(\mathtt{if}\ (f)\ \mathsf{b}\ \mathtt{else}\ \mathsf{c})\tfrac{y}{x}]\!]\omega\\
&= [\![(\mathtt{if}\ (\xi(f)\tfrac{y}{x})\ \xi(\mathsf{b})\tfrac{y}{x}\ \mathtt{else}\ \xi(\mathsf{c})\tfrac{y}{x})]\!]\omega\\
&= (\mathtt{if}\ ([\![\xi(f)\tfrac{y}{x}]\!]\omega)\ [\![\xi(\mathsf{b})\tfrac{y}{x}]\!]\omega\ \mathtt{else}\ [\![\xi(\mathsf{c})\tfrac{y}{x}]\!]\omega)\\
&= (\mathtt{if}\ ([\![\xi(f)]\!]\omega\tfrac{y}{x})\ [\![\xi(\mathsf{b})]\!]\omega\tfrac{y}{x}\ \mathtt{else}\ [\![\xi(\mathsf{c})]\!]\omega\tfrac{y}{x})\\
&= [\![(\mathtt{if}\ (\xi(f))\ \xi(\mathsf{b})\ \mathtt{else}\ \xi(\mathsf{c})]\!]\omega\tfrac{y}{x}\\
&= [\![\xi(\mathtt{if}\ (f)\ \mathsf{b}\ \mathtt{else}\ \mathsf{c})]\!]\omega\tfrac{y}{x}
\end{aligned}
$$

□

Recall that $\mathrm{FV}(e)$ are the free variables of expression e and $\mathrm{MBV}(\alpha)$ are the must-bound variables of game α as defined in Fig. A.11.

Lemma 4.11 (Coincidence)**.** *Only the free variables of an expression and of its realizer influence its semantics. That is, assume $\omega = \tilde{\omega}$ on $V \supseteq \mathrm{FV}(e)$ (where e is f or ϕ or α or b). In the claims for projections, games, and formulas, additionally assume $V \supseteq \mathrm{FV}(\mathsf{b})$. Recall the free variables $\mathrm{FV}(\mathsf{b})$ of realizer b are those which appear syntactically free at any point in b, not just components which get used in the current state. Then:*

- $[\![f]\!]\omega = [\![f]\!]\tilde{\omega}$

- $[\![\mathsf{b}]\!]\omega = [\![\mathsf{b}]\!]\tilde{\omega}$
- $\{(\mathsf{b},\omega)\}_{[0]} \overset{V}{=} \{(\mathsf{b},\tilde{\omega})\}_{[0]}$ *and* $\{(\mathsf{b},\omega)\}_{[1]} \overset{V}{=} \{(\mathsf{b},\tilde{\omega})\}_{[1]}$
- $\{(\mathsf{b},\omega)\}_{\langle 0\rangle} \overset{V}{=} \{(\mathsf{b},\tilde{\omega})\}_{\langle 0\rangle}$ *and* $\{(\mathsf{b},\omega)\}_{\langle 1\rangle} \overset{V}{=} \{(\mathsf{b},\tilde{\omega})\}_{\langle 1\rangle}$
- $\{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle = \{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha\rangle\!\rangle$ *on* $\mathrm{MBV}(\alpha)\cup V$
- $\{(\mathsf{b},\omega)\}[\![\alpha]\!] = \{(\mathsf{b},\tilde{\omega})\}[\![\alpha]\!]$ *on* $\mathrm{MBV}(\alpha)\cup V$
- $(\mathsf{b},\omega)\in[\![\phi]\!]$ *iff* $(\mathsf{b},\tilde{\omega})\in[\![\phi]\!]$

Proof. In each case the assumption (Agree) is the assumption $\omega = \tilde{\omega}$ on $V \supseteq \mathrm{FV}(e)$. In the projection, game, and formula cases also assume (AgreeRz) $V \supseteq \mathrm{FV}(\mathsf{b})$ where $\mathrm{FV}(\mathsf{b})$ contains all variables which appear in free syntactic position at any point throughout b. The term claim is by induction on terms f. The formula and game claims are by induction simultaneously on α, ϕ. The realizer claim is shown by strengthening the claim to maintain a realizer substitution ξ which closes the realizer and where the coincidence lemma is assumed for every element of ξ. The realizer claim then proceeds by an outer induction on the type of a realizer and inner coinduction on realizer syntax.

Recall that the notation $\omega \overset{V}{=} \nu$ says ω and ν assign the same values for all $x \in V$ where $V \subseteq \mathcal{V}$. Recall that for proper regions X and Y, we write $X = Y$ on V when for all $(\mathsf{b},\omega)\in X$ there exists $\tilde{\omega}$ where $\omega = \tilde{\omega}$ on V such that $(\mathsf{b},\tilde{\omega})\in Y$ and vice versa. For (not necessarily proper) regions X and Y, we write $X = Y$ on V iff $X \setminus \{\top,\bot\} = Y \setminus \{\top,\bot\}$ on V and $X \cap \{\top,\bot\} = Y \cap \{\top,\bot\}$.

Realizer cases: The realizer cases prove the stronger claim $[\![\xi(\mathsf{b})]\!]\omega = [\![\xi(\mathsf{b})]\!]\tilde{\omega}$ assuming coincidence for all realizers $\xi(z)$.

Case ϵ: Have $[\![\xi(\epsilon)]\!]\omega = [\![\epsilon]\!]\omega = () = [\![\epsilon]\!]\tilde{\omega} = [\![\xi(\epsilon)]\!]\tilde{\omega}$.

Case z, where z is a variable over realizers: Then $[\![\xi(z)]\!]\omega = [\![z]\!]\tilde{\omega}$ for all ω and $\tilde{\omega}$ satisfying assumption (Agree), by the invariant on ξ.

Case f: Then $[\![\xi(f)]\!]\omega = [\![f]\!]\omega = [\![f]\!]\tilde{\omega} = [\![\xi(f)]\!]\tilde{\omega}$ by the claim on terms and because realizer substitutions have no effect on terms and because (Agree) means $\omega \overset{\mathrm{FV}(f)}{=} \tilde{\omega}$.

Case $\pi_0\mathsf{b}$: Have $[\![\xi(\pi_0\mathsf{b})]\!]\omega = [\![\pi_0\xi(\mathsf{b}))]\!]\omega = \pi_0[\![\xi(\mathsf{b})]\!]\omega \underset{\text{IH}}{=} \pi_0[\![\xi(\mathsf{b})]\!]\tilde{\omega} = [\![\xi(\pi_0\mathsf{b})]\!]\tilde{\omega}$.

Case $\pi_1\mathsf{b}$: Have $[\![\xi(\pi_1\mathsf{b})]\!]\omega = [\![\pi_1\xi(\mathsf{b}))]\!]\omega = \pi_1[\![\xi(\mathsf{b})]\!]\omega \underset{\text{IH}}{=} \pi_1[\![\xi(\mathsf{b})]\!]\tilde{\omega} = [\![\xi(\pi_1\mathsf{b})]\!]\tilde{\omega}$.

Case (b,c): Have

$$\begin{aligned}&[\![\xi((\mathsf{b},\mathsf{c}))]\!]\omega = [\![(\xi(\mathsf{b}),\xi(\mathsf{c})))]\!]\omega = ([\![\xi(\mathsf{b})]\!]\omega, [\![\xi(\mathsf{c})]\!]\omega)\\ \underset{\text{IH}}{=}\ &([\![\xi(\mathsf{b})]\!]\tilde{\omega}, [\![\xi(\mathsf{c})]\!]\tilde{\omega}) = [\![(\xi(\mathsf{b}),\xi(\mathsf{c})))]\!]\tilde{\omega} = [\![\xi((\mathsf{b},\mathsf{c}))]\!]\tilde{\omega}\end{aligned}$$

Case $(\Lambda x : \mathbb{Q}.\ \mathsf{b})$: By transitivity, have $[\![\xi(\Lambda x:\mathbb{Q}.\ \mathsf{b})]\!]\omega = [\![\Lambda x:\mathbb{Q}.\ \xi(\mathsf{b})]\!]\omega = (\Lambda q : \mathbb{Q}.\ [\![\xi(\mathsf{b})]\!]\omega[x\mapsto q]) \underset{\text{IH}}{=} (\Lambda q:\mathbb{Q}.\ [\![\xi(\mathsf{b})]\!]\tilde{\omega}[x\mapsto q] = [\![\Lambda x:\mathbb{Q}.\ \xi(\mathsf{b})]\!]\tilde{\omega} = [\![\xi(\Lambda x:\mathbb{Q}.\ \mathsf{b})]\!]\tilde{\omega}$ where the IH step uses the fact that $\omega[x\mapsto q]$ and $\tilde{\omega}[x\mapsto q]$ agree on the free variables of $\xi(\mathsf{b})$ because ω and $\tilde{\omega}$ do by (Agree) and because they agree on the value q of x as well.

Case $(\Lambda z : \phi\,\mathcal{R}\mathbf{z}.\ \mathsf{b})$:

We have $[\![\xi(\Lambda z:\phi\,\mathcal{R}\mathbf{z}.\ \mathsf{b})]\!]\omega = [\![(\Lambda z:\phi\,\mathcal{R}\mathbf{z}.\ \xi(\mathsf{b}))]\!]\omega = (\Lambda w:\phi\,\mathcal{R}\mathbf{z}.\ [\![\xi(\mathsf{b})[z\mapsto w]]\!]\omega) =_{(*)} (\Lambda w:\phi\,\mathcal{R}\mathbf{z}.\ [\![\xi'(\mathsf{b})]\!]\omega) \underset{\text{IH}}{=} (\Lambda w:\phi\,\mathcal{R}\mathbf{z}.\ [\![\xi'(\mathsf{b})]\!]\tilde{\omega}) = [\![\Lambda z:\phi\,\mathcal{R}\mathbf{z}.\ \xi(\mathsf{b})]\!]\tilde{\omega} = [\![\xi(\Lambda z:\phi\,\mathcal{R}\mathbf{z}.\ \mathsf{b})]\!]\tilde{\omega}$ where ξ' extends ξ with the substitution $[z\mapsto w]$. The proof begins by simplifying ξ

and the realizer interpretation function homomorphically. The denotation of a function realizer is a function returning denotations, so the equality of two denotations is shown extensionally by showing the bodies agree for every argument w. The step marked (*) says substitution by $\xi[z \mapsto w]$ is the same as substitution by ξ', by definition of ξ'. The crucial step is showing that the (co)-IH is applicable. The co-IH is well-founded because it is guarded by a constructor (a λ-abstraction). To satisfy the assumption of the co-IH we must show that the coincidence lemma applies to the realizer w. Because any appearance of w would be guarded by the same λ-abstraction, one could attempt a well-foundedness argument, but there is a simpler argument: The type of w is structurally smaller than the type of $(\Lambda z : \phi\, \mathcal{R}\mathbf{z}.\ \mathsf{b})$, so the coincidence lemma can be assumed for w by an outer structural induction on types.

Case $(\mathsf{b}\ f)$: Have

$$\begin{aligned}&[\![\xi(\mathsf{b}\ f)]\!]\omega = [\![\xi(\mathsf{b})\ \xi(f))]\!]\omega = [\![\xi(\mathsf{b})\ f)]\!]\omega = [\![\xi(\mathsf{b})]\!]\omega\ [\![f]\!]\omega\\ \underset{\text{IH}}{=}\ &[\![\xi(\mathsf{b})]\!]\tilde{\omega}\ [\![f]\!]\tilde{\omega} = [\![\xi(\mathsf{b})]\!]\tilde{\omega}\ [\![\xi(f)]\!]\tilde{\omega} = [\![\xi(\mathsf{b})\ \xi(f))]\!]\tilde{\omega} = [\![\xi(\mathsf{b}\ f)]\!]\tilde{\omega}\end{aligned}$$

where the IH step also uses the claim on terms.

Case $(\mathsf{b}\ \mathsf{c})$: We have $[\![\xi(\mathsf{b}\ \mathsf{c})]\!]\omega = [\![\xi(\mathsf{b})\ \xi(\mathsf{c}))]\!]\omega = [\![\xi(\mathsf{b})]\!]\omega\ [\![\xi(\mathsf{c})]\!]\omega \underset{\text{IH}}{=} [\![\xi(\mathsf{b})]\!]\tilde{\omega}\ [\![\xi(\mathsf{c})]\!]\tilde{\omega} = [\![\xi(\mathsf{b})\ \xi(\mathsf{c}))]\!]\tilde{\omega} = [\![\xi(\mathsf{b}\ \mathsf{c})]\!]\tilde{\omega}$.

Case $(\mathtt{if}\ (f)\ \mathsf{b}\ \mathtt{else}\ \mathsf{c})$: Have

$$\begin{aligned}&[\![\xi(\mathtt{if}\ (f)\ \mathsf{b}\ \mathtt{else}\ \mathsf{c})]\!]\omega = [\![\mathtt{if}\ (\xi(f))\ \xi(\mathsf{b})\ \mathtt{else}\ \xi(\mathsf{c})]\!]\omega\\ =\ &\mathtt{if}\ ([\![\xi(f)]\!]\omega)\ [\![\xi(\mathsf{b})]\!]\omega\ \mathtt{else}\ [\![\xi(\mathsf{c})]\!]\omega = \mathtt{if}\ ([\![\xi(f)]\!]\tilde{\omega})\ [\![\xi(\mathsf{b})]\!]\tilde{\omega}\ \mathtt{else}\ [\![\xi(\mathsf{c})]\!]\tilde{\omega}\\ =\ &[\![\mathtt{if}\ (\xi(f))\ \xi(\mathsf{b})\ \mathtt{else}\ \xi(\mathsf{c})]\!]\tilde{\omega} = [\![\xi(\mathtt{if}\ (f)\ \mathsf{b}\ \mathtt{else}\ \mathsf{c})]\!]\tilde{\omega}\end{aligned}$$

Term cases:

Case q: Have $[\![q]\!]\omega = q = [\![q]\!]\tilde{\omega}$.

Case x: Have $[\![x]\!]\omega = \omega(x) = \tilde{\omega}(x) = [\![x]\!]\tilde{\omega}$ by (Agree).

Case $f + g$: Have $[\![f + g]\!]\omega = [\![f]\!]\omega + [\![g]\!]\omega = [\![f]\!]\tilde{\omega} + [\![g]\!]\tilde{\omega} = [\![f + g]\!]\tilde{\omega}$ by the IHs.

Case $f \cdot g$: Have $[\![f \cdot g]\!]\omega = [\![f]\!]\omega \cdot [\![g]\!]\omega = [\![f]\!]\tilde{\omega} \cdot [\![g]\!]\tilde{\omega} = [\![f \cdot g]\!]\tilde{\omega}$ by the IHs.

Case $f\ \mathsf{div}\ g$: Have $[\![f\ \mathsf{div}\ g]\!]\omega = [\![f]\!]\omega\ \mathsf{div}\ [\![g]\!]\omega = [\![f]\!]\tilde{\omega}\ \mathsf{div}\ [\![g]\!]\tilde{\omega} = [\![f\ \mathsf{div}\ g]\!]\tilde{\omega}$ by the inductive hypotheses.

Case $f\ \mathsf{mod}\ g$: Have $[\![f\ \mathsf{mod}\ g]\!]\omega = [\![f]\!]\omega\ \mathsf{mod}\ [\![g]\!]\omega = [\![f]\!]\tilde{\omega}\ \mathsf{mod}\ [\![g]\!]\tilde{\omega} = [\![f\ \mathsf{mod}\ g]\!]\tilde{\omega}$ by the inductive hypotheses.

Projection cases:

We consider Demonic projection first.

Demonic projection cases:

Case $\cdot_{[0]}$: Have $\{(\mathsf{b}, \omega)\}_{[0]} = \{(\pi_0\mathsf{b}, \omega)\} \overset{V}{=} \{(\pi_0\mathsf{b}, \tilde{\omega})\} = \{(\mathsf{b}, \tilde{\omega})\}_{[0]}$ where the crucial $\overset{V}{=}$ step holds by (Agree).

Case $\cdot_{[1]}$: Have $\{(\mathsf{b}, \omega)\}_{[1]} = \{(\pi_1\mathsf{b}, \omega)\} \overset{V}{=} \{(\pi_1\mathsf{b}, \tilde{\omega})\} = \{(\mathsf{b}, \tilde{\omega})\}_{[1]}$ where the crucial $\overset{V}{=}$ step holds by (Agree).

Angelic projection cases:

In the proof, call the argument region of the projection function X, which has form $\{(\mathsf{b},\omega)\}$. In this proof, the set comprehension (for example) $\{(\pi_1\mathsf{b},\omega)\in X \mid [\![\pi_0\mathsf{b}]\!]\omega = 0\}$ indicates the singleton set $\{(\pi_1\mathsf{b},\omega)\}$ when $[\![\pi_0\mathsf{b}]\!]\omega = 0$, else the empty set.

Case $\cdot_{\langle 0\rangle}$: Have $\{(\mathsf{b},\omega)\}_{\langle 0\rangle} = \{(\pi_1\mathsf{b},\omega)\in X \mid [\![\pi_0\mathsf{b}]\!]\omega = 0\} \overset{V}{=} \{(\pi_1\mathsf{b},\tilde{\omega})\in X \mid [\![\pi_0\mathsf{b}]\!]\tilde{\omega} = 0\} = \{(\mathsf{b},\tilde{\omega})\}_{\langle 0\rangle}$. The crucial step is the $\overset{V}{=}$ step. To show the step, first note $\mathrm{FV}(\pi_0\mathsf{b}) \subseteq \mathrm{FV}(\mathsf{b})$ by definition, so that by (Agree) we can apply term coincidence to get $[\![\pi_0\mathsf{b}]\!]\omega = [\![\pi_0\mathsf{b}]\!]\tilde{\omega} = 0$. To complete the justification of the step, note $\omega \overset{V}{=} \tilde{\omega}$ already holds by (Agree).

Case $\cdot_{\langle 1\rangle}$: Have $\{(\mathsf{b},\omega)\}_{\langle 1\rangle} = \{(\pi_1\mathsf{b},\omega)\in X \mid [\![\pi_0\mathsf{b}]\!]\omega = 1\} \overset{V}{=} \{(\pi_1\mathsf{b},\tilde{\omega})\in X \mid [\![\pi_0\mathsf{b}]\!]\tilde{\omega} = 1\} = \{(\mathsf{b},\tilde{\omega})\}_{\langle 1\rangle}$. The crucial step is the $\overset{V}{=}$ step. To show the step, first note $\mathrm{FV}(\pi_0\mathsf{b}) \subseteq \mathrm{FV}(\mathsf{b})$ by definition, so that by (Agree) we can apply term coincidence to get $[\![\pi_0\mathsf{b}]\!]\omega = [\![\pi_0\mathsf{b}]\!]\tilde{\omega} = 1$. To complete the justification of the step, note $\omega \overset{V}{=} \tilde{\omega}$ already holds by (Agree).

Angel cases:

Case $x := f$: Have $\{(\mathsf{b},\omega)\}\langle\!\langle x := f\rangle\!\rangle = \{(\mathsf{b},\omega[x\mapsto [\![f]\!]\omega]\} = \{(\mathsf{b},\omega[x\mapsto [\![f]\!]\tilde{\omega}]\}$ which equals $\{(\mathsf{b},\tilde{\omega}[x\mapsto [\![f]\!]\tilde{\omega}]\}$ on $V\cup \mathrm{MBV}(\alpha) = V\cup\{x\}$.

Case $x := *$: Have $\{(\mathsf{b},\omega)\}\langle\!\langle x := *\rangle\!\rangle = \{(\pi_1\mathsf{b},\omega[x\mapsto [\![\pi_0\mathsf{b}]\!]\omega]\}$ and want to show $\{(\pi_1\mathsf{b},\omega[x\mapsto [\![\pi_0\mathsf{b}]\!]\omega]\} \overset{V}{=} \{(\pi_1\mathsf{b},\tilde{\omega}[x\mapsto [\![\pi_0\mathsf{b}]\!]\tilde{\omega}]\}$ so that the case will follow in one more step from $\{(\pi_1\mathsf{b},\tilde{\omega}[x\mapsto [\![\pi_0\mathsf{b}]\!]\tilde{\omega}]\} = \{(\mathsf{b},\tilde{\omega})\}\langle\!\langle x := *\rangle\!\rangle$.

It suffices to show $\omega[x\mapsto [\![\pi_0\mathsf{b}]\!]\omega] \overset{V\cup\mathrm{MBV}(x:=*)}{=} \tilde{\omega}[x\mapsto [\![\pi_0\mathsf{b}]\!]\tilde{\omega}]$ for $V\cup\mathrm{MBV}(x := *) = V\cup\{x\}$. Because $\omega = \tilde{\omega}$ on V it suffices to show that the values of x agree, i.e., that $[\![\pi_0\mathsf{b}]\!]\omega = [\![\pi_0\mathsf{b}]\!]\tilde{\omega}$, which holds by claim for realizers which is applicable since since $\omega \overset{V}{=} \tilde{\omega}$.

Case $?\phi$: Have

$$
\begin{aligned}
&\{(\mathsf{b},\omega)\}\langle\!\langle ?\phi\rangle\!\rangle\\
&\overset{V}{=} \{(\pi_1\mathsf{b},\omega)\} \text{ if } (\pi_0\mathsf{b},\omega)\in[\![\phi]\!] \text{ else } \{\bot\}\\
&\overset{V}{=}_{\mathrm{IH}} \{(\pi_1\mathsf{b},\omega)\} \text{ if } (\pi_0\mathsf{b},\tilde{\omega})\in[\![\phi]\!] \text{ else } \{\bot\}\\
&\overset{V}{=}_{(\mathrm{Agree})} \{(\pi_1\mathsf{b},\tilde{\omega})\} \text{ if } (\pi_0\mathsf{b},\tilde{\omega})\in[\![\phi]\!] \text{ else } \{\bot\}\\
&\overset{V}{=} \{(\mathsf{b},\tilde{\omega})\}\langle\!\langle ?\phi\rangle\!\rangle
\end{aligned}
$$

which suffices since $V = V\cup\mathrm{MBV}(?\phi) = V\cup\emptyset$.

Case $\alpha;\beta$: Have $\{(\mathsf{b},\omega)\}\langle\!\langle\alpha;\beta\rangle\!\rangle = \{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle\langle\!\langle\beta\rangle\!\rangle$. Then by the IH on α have $\{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle = \{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha\rangle\!\rangle$ on $\mathrm{MBV}(\alpha)\cup V \supseteq \mathrm{FV}(\beta)$. Then by Lemma A.1 and the inductive hypothesis on β have $(\{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle)\langle\!\langle\beta\rangle\!\rangle = (\{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha\rangle\!\rangle)\langle\!\langle\beta\rangle\!\rangle$ on $\mathrm{MBV}(\alpha;\beta)\cup V = \mathrm{MBV}(\alpha)\cup\mathrm{MBV}(\beta)\cup V$.

Case $\alpha\cup\beta$: Have $\{(\mathsf{b},\omega)\}\langle\!\langle\alpha\cup\beta\rangle\!\rangle = (\{(\mathsf{b},\omega)\}_{\langle 0\rangle}\langle\!\langle\alpha\rangle\!\rangle)\cup(\{(\mathsf{b},\omega)\}_{\langle 1\rangle}\langle\!\langle\beta\rangle\!\rangle)$. It suffices to show $(\{(\mathsf{b},\omega)\}_{\langle 0\rangle}\langle\!\langle\alpha\rangle\!\rangle)\cup(\{(\mathsf{b},\omega)\}_{\langle 1\rangle}\langle\!\langle\beta\rangle\!\rangle)$ agrees with $(\{(\mathsf{b},\tilde{\omega})\}_{\langle 0\rangle}\langle\!\langle\alpha\rangle\!\rangle)\cup(\{(\mathsf{b},\tilde{\omega})\}_{\langle 1\rangle}\langle\!\langle\beta\rangle\!\rangle)$ on $\mathrm{MBV}(\alpha\cup\beta)\cup V$, the claim holds by $(\{(\mathsf{b},\tilde{\omega})\}_{\langle 0\rangle}\langle\!\langle\alpha\rangle\!\rangle)\cup(\{(\mathsf{b},\tilde{\omega})\}_{\langle 1\rangle}\langle\!\langle\beta\rangle\!\rangle) = \{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha\cup\beta\rangle\!\rangle$.

Note by the $\cdot_{\langle 0\rangle}$ and $\cdot_{\langle 1\rangle}$ claims and (RzAgree) that $\{(\mathsf{b},\omega)\}_{\langle 0\rangle} \overset{V}{=} \{(\mathsf{b},\tilde{\omega})\}_{\langle 0\rangle}$ and $\{(\mathsf{b},\omega)\}_{\langle 1\rangle} \overset{V}{=} \{(\mathsf{b},\tilde{\omega})\}_{\langle 1\rangle}$, respectively.

Then by the IHs[3] on α and β have (IHA) $\{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle = \{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha\rangle\!\rangle$ on $\mathrm{MBV}(\alpha)\cup V$

[3]The agreement requirement on realizers is met since projection never increases the free variable set.

and (IHB) $\{(\mathsf{b},\omega)\}\langle\!\langle\beta\rangle\!\rangle = \{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\beta\rangle\!\rangle$ on $\mathrm{MBV}(\beta)\cup V$ so that in each case agreement holds (AgreeV) on $\mathrm{MBV}(\alpha\cup\beta)\cup V$ by the definition $\mathrm{MBV}(\alpha\cup\beta) = \mathrm{MBV}(\alpha)\cap\mathrm{MBV}(\beta)$.

The case holds immediately by (IHA), (IHB), and (AgreeV): the semantics for each branch agree on $\mathrm{MBV}(\alpha\cup\beta)\cup V$, thus the semantics of the choice do as well.

Case α^*: Note $\mathrm{MBV}(\alpha^*) = \emptyset$ by definition so $\mathrm{MBV}(\alpha^*)\cup V = V$ and it thus suffices to show $\{(\mathsf{b},\omega)\}\langle\!\langle\alpha^*\rangle\!\rangle = \{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha^*\rangle\!\rangle$ on V. By Lemma 4.8 have $\{(\mathsf{b},\omega)\}\langle\!\langle\alpha^*\rangle\!\rangle = \bigcup_{k\in\mathbb{N}}(\{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle^k)_{\langle 0\rangle}$ and $\{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha^*\rangle\!\rangle = \bigcup_{k\in\mathbb{N}}(\{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha\rangle\!\rangle^k)_{\langle 0\rangle}$, so transitivity it suffices to show

$$\bigcup_{k\in\mathbb{N}}(\{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle^k)_{\langle 0\rangle} \overset{V}{=} \bigcup_{k\in\mathbb{N}}(\{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha\rangle\!\rangle^k)_{\langle 0\rangle}$$

By the assumption (RzAgree) and the coincidence claim for Angelic projection, it suffices to show $(\{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle^k \overset{V}{=} (\{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha\rangle\!\rangle^k$ for all $k\in\mathbb{N}$. Proceed by inner induction on k, letting b,ω, and $\tilde{\omega}$ vary.

In the base case, have $\{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle^0 = \{(\mathsf{b},\omega)\}$ by definition and $\{(\mathsf{b},\omega)\} = \{(\mathsf{b},\tilde{\omega})\}$ on V by assumption and $\{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha\rangle\!\rangle^0 = \{(\mathsf{b},\tilde{\omega})\}$ by definition so that $\{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle^0 = \{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha\rangle\!\rangle^0$ on V by transitivity.

In the inductive step, have $i = j+1$ for some $j\in\mathbb{N}$. Assume the IH that for all $\mathsf{c},\nu,\tilde{\nu}$ we have $\{(\mathsf{c},\nu)\}\langle\!\langle\alpha\rangle\!\rangle^j = \{(\mathsf{c},\tilde{\nu})\}\langle\!\langle\alpha\rangle\!\rangle^j$ on V if $\nu=\tilde{\nu}$ on V. Show $\{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle^i = \{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha\rangle\!\rangle^i$ by showing

$$\begin{aligned}
&\{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle^i && \\
&= \{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle^{j+1} && \text{By case}\\
&= (\{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle^j)_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle && \text{By definition}\\
&\overset{V}{=}_{(*)} (\{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha\rangle\!\rangle^j)_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle && \text{Proved below}\\
&= \{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha\rangle\!\rangle^{j+i} && \text{By definition}\\
&= \{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha\rangle\!\rangle^i && \text{By case}
\end{aligned}$$

We show the starred step (*). By the inner IH have $\{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle^j = \{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha\rangle\!\rangle^j$ on V, then $(\{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle^j)_{\langle 1\rangle} = (\{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha\rangle\!\rangle^j)_{\langle 1\rangle}$ on V by (RzAgree) and the claim for Angelic projections. Then $(\{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle^j)_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle = (\{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha\rangle\!\rangle^j)_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle$ on $\mathrm{MBV}(\alpha)\cup V$ by the outer IH on α, which proves the inner inductive case since $\mathrm{MBV}(\alpha)\cup V\supseteq V$. This completes the inner induction, which completes the (outer) case.

Case α^d: Have $\{(\mathsf{b},\omega)\}\langle\!\langle\alpha^d\rangle\!\rangle = \{(\mathsf{b},\omega)\}[\![\alpha]\!] \underset{\text{IH}}{=} \{(\mathsf{b},\tilde{\omega})\}[\![\alpha]\!]$ on $V\cup\mathrm{MBV}(\alpha) = V\cup\mathrm{MBV}(\alpha^d)$ which equals $\{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha^d\rangle\!\rangle$, yielding the case.

Demon cases:

Case $x := f$: Have $\{(\mathsf{b},\omega)\}[\![x:=f]\!] = \{(\mathsf{b},\omega[x\mapsto[\![f]\!]\omega]\} = \{(\mathsf{b},\omega[x\mapsto[\![f]\!]\tilde{\omega}]\}$ which equals $\{(\mathsf{b},\tilde{\omega}[x\mapsto[\![f]\!]\tilde{\omega}]\}$ on $V\cup\mathrm{MBV}(\alpha) = V\cup\{x\}$.

Case $x := *$: Have $\{(\mathsf{b},\omega)\}[\![x:=*]\!] = \{(\mathsf{b}\,v,\omega[x\mapsto v]) \mid v\in\mathbb{Q}\}$ and want to show $\{(\mathsf{b}\,v,\omega[x\mapsto v]) \mid v\in\mathbb{Q}\} = \{(\mathsf{b}\,v,\tilde{\omega}[x\mapsto v]) \mid v\in\mathbb{Q}\}$ on V so that the case will follow in one more step from $\{(\mathsf{b}\,v,\omega[x\mapsto v]) \mid v\in\mathbb{Q}\} = \{(\mathsf{b},\tilde{\omega})\}[\![x:=*]\!]$.

It suffices to show $\omega[x\mapsto v] = \tilde{\omega}[x\mapsto v]$ on $V\cup\mathrm{MBV}(\alpha) = V\cup\{x\}$ for all $v\in\mathbb{Q}$. Because $\omega=\tilde{\omega}$ on V it suffices to show the values of x agree. By reflexivity, they do.

Case $?\phi$: Where S and R are sets we write "S else R" to mean a set which equals R when $S = \emptyset$, else it equals S. In the set comprehensions that follow, realizer c is existentially quantified, i.e., the value of the comprehension is a union across all realizers c. By expanding the semantics of tests, have $\{(\mathsf{b},\omega)\}[\![?\phi]\!] = \{(\mathsf{b}\,\mathsf{c},\omega) \mid (\mathsf{c},\omega) \in [\![\phi]\!]\}$ else $\{\top\}$, likewise, we have

$$\{(\mathsf{b},\tilde{\omega})\}[\![?\phi]\!] = \{(\mathsf{b}\,\mathsf{c},\tilde{\omega}) \mid (\mathsf{c},\tilde{\omega}) \in [\![\phi]\!]\} \text{ else } \{\top\}$$

then by the formula IH and (Agree) have for all realizers c that (AgreeA) $(\mathsf{c},\tilde{\omega}) \in [\![\phi]\!]$ iff $(\mathsf{c},\omega) \in [\![\phi]\!]$. Thus, (AgreeE) there exists c such that $(\mathsf{c},\tilde{\omega}) \in [\![\phi]\!]$ iff there exists c such that $(\mathsf{c},\omega) \in [\![\phi]\!]$. From (Agree) we have $\omega = \tilde{\omega}$ on $V \cup \text{MBV}(?\phi) = V$, which, combined with (AgreeA), gives that $\{(\mathsf{b}\,\mathsf{c},\omega) \mid (\mathsf{c},\omega) \in [\![\phi]\!]\}$ and $\{(\mathsf{b}\,\mathsf{c},\tilde{\omega}) \mid (\mathsf{c},\tilde{\omega}) \in [\![\phi]\!]\}$ agree on $V \cup \text{MBV}(?\phi)$. Then $(\{(\mathsf{b}\,\mathsf{c},\omega) \mid (\mathsf{c},\omega) \in [\![\phi]\!]\}$ else $\{\top\})$ and $(\{(\mathsf{b}\,\mathsf{c},\tilde{\omega}) \mid (\mathsf{c},\tilde{\omega}) \in [\![\phi]\!]\}$ else $\{\top\})$ also agree on $V \cup \text{MBV}(?\phi)$ because, by (AgreeE), either both tests succeed (return proper possibilities) or both fail (return $\{\top\}$).

Case $\alpha;\beta$: Have $\{(\mathsf{b},\omega)\}[\![\alpha;\beta]\!] = \{(\mathsf{b},\omega)\}[\![\alpha]\!][\![\beta]\!]$. By IH on α, have $\{(\mathsf{b},\omega)\}[\![\alpha]\!] = \{(\mathsf{b},\tilde{\omega})\}[\![\alpha]\!]$ on $\text{MBV}(\alpha) \cup V$. Then by Lemma A.1 and the IH on β have $(\{(\mathsf{b},\omega)\}[\![\alpha]\!])[\![\beta]\!] = (\{(\mathsf{b},\tilde{\omega})\}[\![\alpha]\!])[\![\beta]\!]$ on $\text{MBV}(\alpha;\beta) \cup V = \text{MBV}(\alpha) \cup \text{MBV}(\beta) \cup V$.

Case $\alpha \cup \beta$: Have $\{(\mathsf{b},\omega)\}[\![\alpha \cup \beta]\!] = (\{(\mathsf{b},\omega)\}_{[0]}[\![\alpha]\!]) \cup (\{(\mathsf{b},\omega)\}_{[1]}[\![\beta]\!])$. It suffices to show that $(\{(\mathsf{b},\omega)\}_{[0]}[\![\alpha]\!]) \cup (\{(\mathsf{b},\omega)\}_{[1]}[\![\beta]\!])$ agrees with $(\{(\mathsf{b},\tilde{\omega})\}_{[0]}[\![\alpha]\!]) \cup (\{(\mathsf{b},\tilde{\omega})\}_{[1]}[\![\beta]\!])$ on $\text{MBV}(\alpha \cup \beta) \cup V$, from which the case follows by $(\{(\mathsf{b},\tilde{\omega})\}_{[0]}[\![\alpha]\!]) \cup (\{(\mathsf{b},\tilde{\omega})\}_{[1]}[\![\beta]\!]) = \{(\mathsf{b},\tilde{\omega})\}[\![\alpha \cup \beta]\!]$.

Note by the claim for projections and (RzAgree) that $\{(\mathsf{b},\omega)\}_{[0]} \overset{V}{=} \{(\mathsf{b},\tilde{\omega})\}_{[0]}$ and $\{(\mathsf{b},\omega)\}_{[1]} \overset{V}{=} \{(\mathsf{b},\tilde{\omega})\}_{[1]}$, respectively.

Then by the IHs[4] on α and β have (IHA) $\{(\mathsf{b},\omega)\}[\![\alpha]\!] = \{(\mathsf{b},\tilde{\omega})\}[\![\alpha]\!]$ on $\text{MBV}(\alpha) \cup V$ and (IHB) $\{(\mathsf{b},\omega)\}[\![\beta]\!] = \{(\mathsf{b},\tilde{\omega})\}[\![\beta]\!]$ on $\text{MBV}(\beta) \cup V$ so that in each case agreement holds (AgreeV) on $\text{MBV}(\alpha \cup \beta) \cup V$ by the definition $\text{MBV}(\alpha \cup \beta) = \text{MBV}(\alpha) \cap \text{MBV}(\beta)$.

The case holds immediately by (IHA), (IHB), and (AgreeV): the semantics for each branch agree on $\text{MBV}(\alpha \cup \beta) \cup V$, thus the semantics of the choice do as well.

Case α^*: Note $\text{MBV}(\alpha^*) = \emptyset$ by definition so $\text{MBV}(\alpha^*) \cup V = V$ and it thus suffices to show $\{(\mathsf{b},\omega)\}[\![\alpha^*]\!] = \{(\mathsf{b},\tilde{\omega})\}[\![\alpha^*]\!]$ on V. By Lemma 4.8 have $\{(\mathsf{b},\omega)\}[\![\alpha^*]\!] = \bigcup_{k\in\mathbb{N}}(\{(\mathsf{b},\omega)\}[\![\alpha]\!]^k)_{[0]}$ and $\{(\mathsf{b},\tilde{\omega})\}[\![\alpha^*]\!] = \bigcup_{k\in\mathbb{N}}(\{(\mathsf{b},\tilde{\omega})\}[\![\alpha]\!]^k)_{[0]}$, so by transitivity suffices to show $\bigcup_{k\in\mathbb{N}}(\{(\mathsf{b},\omega)\}[\![\alpha]\!]^k)_{[0]} = \bigcup_{k\in\mathbb{N}}(\{(\mathsf{b},\tilde{\omega})\}[\![\alpha]\!]^k)_{[0]}$ on V. By the Demonic projection claim and assumption (RzAgree), it suffices to show $(\{(\mathsf{b},\omega)\}[\![\alpha]\!]^k = (\{(\mathsf{b},\tilde{\omega})\}[\![\alpha]\!]^k$ for all $k \in \mathbb{N}$. Proceed by inner induction on k, letting b,ω, and $\tilde{\omega}$ vary.

In the base case, have $\{(\mathsf{b},\omega)\}[\![\alpha]\!]^0 = \{(\mathsf{b},\omega)\}$ by definition and $\{(\mathsf{b},\omega)\} = \{(\mathsf{b},\tilde{\omega})\}$ on V by assumption and $\{(\mathsf{b},\tilde{\omega})\}[\![\alpha]\!]^0 = \{(\mathsf{b},\tilde{\omega})\}$ by definition so that $\{(\mathsf{b},\omega)\}[\![\alpha]\!]^0 = \{(\mathsf{b},\tilde{\omega})\}[\![\alpha]\!]^0$ on V by transitivity.

In the inductive step, have $i = j + 1$ for some $j \in \mathbb{N}$. Assume the IH that for all $\mathsf{c},\nu,\tilde{\nu}$ we have $\{(\mathsf{c},\nu)\}[\![\alpha]\!]^j = \{(\mathsf{c},\tilde{\nu})\}[\![\alpha]\!]^j$ on V if $\nu = \tilde{\nu}$ on V. Show $\{(\mathsf{b},\omega)\}[\![\alpha]\!]^i = \{(\mathsf{b},\tilde{\omega})\}[\![\alpha]\!]^i$

[4]The agreement requirement on realizers is met since projection never increases the free variable set.

by showing

$$
\begin{aligned}
&\{(\mathsf{b},\omega)\}[\![\alpha]\!]^i \\
&= \{(\mathsf{b},\omega)\}[\![\alpha]\!]^{j+1} && \text{By case}\\
&= (\{(\mathsf{b},\omega)\}[\![\alpha]\!]^j)_{[1]}[\![\alpha]\!] && \text{By definition}\\
&\overset{V}{=}_{(*)} (\{(\mathsf{b},\tilde{\omega})\}[\![\alpha]\!]^j)_{[1]}[\![\alpha]\!] && \text{Proved below}\\
&= \{(\mathsf{b},\tilde{\omega})\}[\![\alpha]\!]^{j+i} && \text{By definition}\\
&= \{(\mathsf{b},\tilde{\omega})\}[\![\alpha]\!]^i && \text{By case}
\end{aligned}
$$

We show the starred step (*). By the inner IH have $\{(\mathsf{b},\omega)\}[\![\alpha]\!]^j = \{(\mathsf{b},\tilde{\omega})\}[\![\alpha]\!]^j$ on V, then $(\{(\mathsf{b},\omega)\}[\![\alpha]\!]^j)_{[1]} = (\{(\mathsf{b},\tilde{\omega})\}[\![\alpha]\!]^j)_{[1]}$ on V by (RzAgree) and the projection claim. Then by the outer IH on α have $(\{(\mathsf{b},\omega)\}[\![\alpha]\!]^j)_{[1]}[\![\alpha]\!] = (\{(\mathsf{b},\tilde{\omega})\}[\![\alpha]\!]^j)_{[1]}[\![\alpha]\!]$ on $\mathrm{MBV}(\alpha)\cup V$ which proves the inner inductive case since $\mathrm{MBV}(\alpha)\cup V \supseteq V$. This completes the inner induction, which completes the (outer) case.

Case α^d: Have $\{(\mathsf{b},\omega)\}[\![\alpha^d]\!] = \{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle \underset{\text{IH}}{=} \{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha\rangle\!\rangle$ on $V\cup\mathrm{MBV}(\alpha) = V\cup\mathrm{MBV}(\alpha^d)$. which equals $\{(\mathsf{b},\tilde{\omega})\}[\![\alpha^d]\!]$ yielding the case.

Formula cases:

Case $f>g$: Have $(\epsilon,\omega)\in[\![f>g]\!]$ iff $[\![f]\!]\omega > [\![g]\!]\omega$ iff $[\![f]\!]\tilde{\omega} > [\![g]\!]\tilde{\omega}$ iff $(\epsilon,\tilde{\omega})\in[\![f>g]\!]$. The cases for $\leq,<,=,\neq,\geq$ are symmetric.

Case $\langle\alpha\rangle\phi$: Have $(\mathsf{b},\omega)\in[\![\langle\alpha\rangle\phi]\!]$ iff $\{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle \subseteq [\![\phi]\!]\cup\{\top\}$ iff$_{(*)}$ $\{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha\rangle\!\rangle \subseteq [\![\phi]\!]\cup\{\top\}$ iff $(\mathsf{b},\tilde{\omega})\in[\![\langle\alpha\rangle\phi]\!]$ where the starred step holds by the game IH because $\omega=\tilde{\omega}$ on $\mathrm{FV}(\langle\alpha\rangle\phi) = \mathrm{FV}(\alpha)\cup(\mathrm{FV}(\phi)\setminus\mathrm{MBV}(\alpha))$, thus $\{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle = \{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha\rangle\!\rangle$ on $\mathrm{MBV}(\alpha)\cup V\supseteq \mathrm{FV}(\phi)\cup\mathrm{FV}(\mathsf{b})$, so that by the IH on ϕ we have $\{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle \subseteq [\![\phi]\!]\cup\{\top\}$ iff $\{(\mathsf{b},\tilde{\omega})\}\langle\!\langle\alpha\rangle\!\rangle \subseteq [\![\phi]\!]\cup\{\top\}$. The realizer free variable assumption holds in the IH on ϕ because the notion of free variables of a realizer captures all variables referenced throughout the future, meaning the set of free variables can only decrease or stay constant throughout execution.

Case $[\alpha]\phi$: Have $(\mathsf{b},\omega)\in[\![[\alpha]\phi]\!]$ iff $\{(\mathsf{b},\omega)\}[\![\alpha]\!]\subseteq[\![\phi]\!]\cup\{\top\}$ iff$_{(*)}$ $\{(\mathsf{b},\tilde{\omega})\}[\![\alpha]\!]\subseteq[\![\phi]\!]\cup\{\top\}$ iff $(\mathsf{b},\tilde{\omega})\in[\![[\alpha]\phi]\!]$ where the starred step holds by the game IH because $\omega=\tilde{\omega}$ on $\mathrm{FV}([\alpha]\phi) = \mathrm{FV}(\alpha)\cup(\mathrm{FV}(\phi)\setminus\mathrm{MBV}(\alpha))$, thus $\{(\mathsf{b},\omega)\}[\![\alpha]\!] = \{(\mathsf{b},\tilde{\omega})\}[\![\alpha]\!]$ on $\mathrm{MBV}(\alpha)\cup V\supseteq \mathrm{FV}(\phi)\cup\mathrm{FV}(\mathsf{b})$ so that by the IH on ϕ we have $\{(\mathsf{b},\omega)\}[\![\alpha]\!]\subseteq[\![\phi]\!]\cup\{\top\}$ iff $\{(\mathsf{b},\tilde{\omega})\}[\![\alpha]\!]\subseteq[\![\phi]\!]\cup\{\top\}$. The realizer free variable assumption holds in the IH on ϕ because the notion of free variables of a realizer captures all variables referenced throughout the future, meaning the set of free variables can only decrease or stay constant throughout execution. □

Lemma 4.12 (Bound effect)**.** *Only the bound variables of a game are modified by execution. Let* $X = \{(\mathsf{b},\omega)\}$. *Let* $(\mathsf{c},\nu)\in X[\![\alpha]\!]$ *or* $(\mathsf{c},\nu)\in X\langle\!\langle\alpha\rangle\!\rangle$. *Then* $\omega=\nu$ *on* $\mathrm{BV}(\alpha)^{\complement}$, *the complement of set* $\mathrm{BV}(\alpha)$.

Proof. **Angel cases:**

Case $x:=f$: Have $(\mathsf{c},\nu)\in X\langle\!\langle x:=f\rangle\!\rangle$ iff $\mathsf{c}=\mathsf{b}$ and $\nu=\omega[x\mapsto[\![f]\!]\omega]$ and $\mathrm{BV}(x:=f)=\{x\}$ and $\nu=\omega$ on $\{x\}^{\complement}$.

Case $x:=*$: Have $(\mathsf{c},\nu)\in X\langle\!\langle x:=*\rangle\!\rangle$ iff $\mathsf{c}=\pi_1\mathsf{b}$ and $\nu=\omega[x\mapsto[\![\pi_0\mathsf{b}]\!]\omega]$ and $\mathrm{BV}(x:=*)=\{x\}$ and $\nu=\omega$ on $\{x\}^{\complement}$.

Case $?\phi$: Have $(\mathsf{c},\nu) \in X\langle\!\langle ?\phi \rangle\!\rangle$ iff $\nu = \omega$ and $\mathsf{c} = \pi_1\mathsf{b}$ and $(\pi_0\mathsf{b},\omega) \in [\![\phi]\!]$ and $(\mathsf{b},\omega) \in X$ then $\omega = \omega$ on $\emptyset^{\complement}$ and $\mathrm{BV}(?\phi) = \emptyset$.

Case $\alpha;\beta$: Have $(\mathsf{c},\nu) \in X\langle\!\langle \alpha;\beta \rangle\!\rangle$ iff $(\mathsf{c},\nu) \in X\langle\!\langle \alpha \rangle\!\rangle\langle\!\langle \beta \rangle\!\rangle$ iff exists $(\mathsf{d},\mu) \in X\langle\!\langle \alpha \rangle\!\rangle$ s.t. $(\mathsf{c},\nu) \in \{(\mathsf{c},\nu)\}\langle\!\langle \beta \rangle\!\rangle$, then result holds by Lemma A.1 and the IH twice since $\mathrm{BV}(\alpha;\beta) = \mathrm{BV}(\alpha) \cup \mathrm{BV}(\beta)$.

Case $\alpha \cup \beta$: Have $(\mathsf{c},\nu) \in X\langle\!\langle \alpha \cup \beta \rangle\!\rangle$ iff $(\mathsf{c},\nu) \in X_{\langle 0\rangle}\langle\!\langle \alpha \rangle\!\rangle \cup X_{\langle 1\rangle}\langle\!\langle \beta \rangle\!\rangle$. In each case one branch is empty and the other singleton, so by the IH on the singleton side, $\omega = \nu$ on $\mathrm{BV}(\alpha)^{\complement}$ or $\mathrm{BV}(\beta)^{\complement}$ and thus in both cases on $\mathrm{BV}(\alpha \cup \beta)^{\complement} = \mathrm{BV}(\alpha)^{\complement} \cap \mathrm{BV}(\beta)^{\complement}$.

Case α^*: Recall that $X = \{(\mathsf{b},\omega)\}$ is given (defX). Let $C = \bigcap\{Z \mid X \cup Z_{\langle 1\rangle}\langle\!\langle \alpha \rangle\!\rangle \subseteq Z\}$ so that we have $X\langle\!\langle \alpha^* \rangle\!\rangle = C_{\langle 0\rangle}$ by the construction of C.

Show by inner induction on the fixed-point construction of C that (Agree) for all $(\mathsf{d},\mu) \in C$ we have $\mu = \omega$ on $\mathrm{BV}(\alpha^*)^{\complement}$. The base case is $(\mathsf{d},\mu) \in X$ which means $\mu = \omega$ by (defX) so that $\mu = \omega$ on $\mathrm{BV}(\alpha^*)^{\complement}$ by reflexivity. In the inductive case, fix Z and assume an inductive hypothesis that for all $(\hat{\mathsf{c}},\hat{\nu}) \in Z$ have $\hat{\nu} = \omega$ on $\mathrm{BV}(\alpha^*)^{\complement}$. The inner case assumption is $(\mathsf{d},\mu) \in Z_{\langle 1\rangle}\langle\!\langle \alpha \rangle\!\rangle$. By the inner IH and definition of $\cdot_{\langle 1\rangle}$, have $\nu = \omega$ on $\mathrm{BV}(\alpha^*)^{\complement}$ for all $(\mathsf{c},\nu) \in Z_{\langle 1\rangle}$. By the outer IH on α have $\mu = \hat{\nu}$ on $\mathrm{BV}(\alpha)^{\complement} = \mathrm{BV}(\alpha^*)^{\complement}$ for some $(\mathsf{c},\hat{\nu}) \in Z_{\langle 1\rangle}$, so by transitivity have $\mu = \omega$ on $\mathrm{BV}(\alpha^*)^{\complement}$ as desired. This completes the inner induction.

To show the case, first observe by construction of C and the (outer) case assumption that $(\mathsf{c},\nu) \in C_{\langle 0\rangle}$, then by unfolding the definition of $C_{\langle 0\rangle}$ have $(\mathsf{d},\nu) \in C$ for some realizer d, then by (Agree) have $\nu = \omega$ on $\mathrm{BV}(\alpha^*)^{\complement}$ as desired.

Case α^d: $(\mathsf{c},\nu) \in X\langle\!\langle \alpha^d \rangle\!\rangle$ iff $(\mathsf{c},\nu) \in X[\![\alpha]\!]$ so $\nu = \omega$ on $\mathrm{BV}(\alpha^d)^{\complement} = \mathrm{BV}(\alpha)^{\complement}$, by the inductive hypothesis.

Demon cases:

Case $x := f$: Have $(\mathsf{c},\nu) \in X[\![x := f]\!]$ iff $\mathsf{c} = \mathsf{b}$ and $\nu = \omega[x \mapsto [\![f]\!]\omega]$ and $\mathrm{BV}(x := f) = \{x\}$ and $\nu = \omega$ on $\{x\}^{\complement}$.

Case $x := *$: Have $(\mathsf{c},\nu) \in X[\![x := *]\!]$ iff $\mathsf{c} = \mathsf{b}\,v$ and $\nu = \omega[x \mapsto v]$, some $v \in \mathbb{Q}$. So $\mathrm{BV}(x := *) = \{x\}$ and $\nu = \omega$ on $\{x\}^{\complement}$.

Case $\alpha;\beta$: Have $(\mathsf{c},\nu) \in X[\![\alpha;\beta]\!]$ iff $(\mathsf{c},\nu) \in X[\![\alpha]\!][\![\beta]\!]$ iff exists $(\mathsf{d},\mu) \in X[\![\alpha]\!]$ s.t. $(\mathsf{c},\nu) \in \{(\mathsf{c},\nu)\}[\![\beta]\!]$, then result holds by Lemma A.1 and the IH twice since $\mathrm{BV}(\alpha;\beta) = \mathrm{BV}(\alpha) \cup \mathrm{BV}(\beta)$.

Case $\alpha \cup \beta$: Have $(\mathsf{c},\nu) \in X[\![\alpha \cup \beta]\!]$ iff $(\mathsf{c},\nu) \in X_{[0]}[\![\alpha]\!] \cup X_{[1]}[\![\beta]\!]$. In each case one branch is empty and the other singleton, so by the IH on the singleton side, $\omega = \nu$ on $\mathrm{BV}(\alpha)^{\complement}$ or $\mathrm{BV}(\beta)^{\complement}$ and thus in both cases on $\mathrm{BV}(\alpha \cup \beta)^{\complement} = \mathrm{BV}(\alpha)^{\complement} \cap \mathrm{BV}(\beta)^{\complement}$.

Case α^*: Let $C = \bigcap\{Z_{[0]} \mid X \cup Z_{[1]}[\![\alpha]\!] \subseteq Z\}$ so that we have $X[\![\alpha^*]\!] = C_{[0]}$ by the construction of C.

Show by inner induction on the fixed-point construction of C that (Agree) for all $(\mathsf{d},\mu) \in C$ we have $\mu = \omega$ on $\mathrm{BV}(\alpha^*)^{\complement}$. The base case is $(\mathsf{d},\mu) \in X$ which means $\mu = \omega$ by (defX) so that $\mu = \omega$ on $\mathrm{BV}(\alpha^*)^{\complement}$ by reflexivity. In the inductive case, fix Z and assume an inductive hypothesis that for all $(\hat{\mathsf{c}},\hat{\nu}) \in Z$ have $\hat{\nu} = \omega$ on $\mathrm{BV}(\alpha^*)^{\complement}$. The inner case assumption is $(\mathsf{d},\mu) \in Z_{[1]}[\![\alpha]\!]$. By the inner IH and definition of $\cdot_{[1]}$, have $\hat{\nu} = \omega$ on $\mathrm{BV}(\alpha^*)^{\complement}$ for all $(\mathsf{c},\hat{\nu}) \in Z_{[1]}$. By the outer IH on α have $\mu = \hat{\nu}$ on $\mathrm{BV}(\alpha)^{\complement} = \mathrm{BV}(\alpha^*)^{\complement}$ for some $(\mathsf{c},\hat{\nu}) \in Z_{[1]}$, so by transitivity have $\mu = \omega$ on $\mathrm{BV}(\alpha^*)^{\complement}$ as desired. This completes the

inner induction.

To show the case, first observe by construction of C and the (outer) case assumption that $(\mathsf{c},\nu) \in C_{[0]}$, then by unfolding the definition of $C_{[0]}$ have $(\mathsf{d},\nu) \in C$ for some realizer d, then by (Agree) have $\nu = \omega$ on $\mathrm{BV}(\alpha^*)^{\complement}$ as desired.

Case α^d: Have $(\mathsf{c},\nu) \in X[\![\alpha^d]\!]$ iff $(\mathsf{c},\nu) \in X\langle\!\langle\alpha\rangle\!\rangle$ so $\nu = \omega$ on $\mathrm{BV}(\alpha^d)^{\complement} = \mathrm{BV}(\alpha)^{\complement}$, by the inductive hypothesis. □

We now turn our attention from static semantics to substitution. The following lemma (Lemma A.6) is a preliminary for the later Lemma A.9. Here we first prove a weak but important statement. Recall that $[\alpha]\phi\,\mathcal{R}\mathbf{z}$ (for example) is the set of Angel realizers which can be *played in* game α with postcondition ϕ where Demon moves first, not the set of realizers which cause Angel to *win* the game. In Lemma A.9, we will prove a stronger statement showing that substitution preserves winnability, but here we must first show that substitution preserves *playability*. That is, if a realizer plays a certain game, then the substitution of the realizer plays the substitution of the game. While the winnability proved in Lemma A.9 is the more interesting property, its theorem statement would not typecheck without Lemma A.6 because the invocation of the semantics in the statement of Lemma A.9 assumes that the realizer types in the substituted initial region agree with the substituted game.

Lemma A.6 (Substitution and realizer typing)**.** *Let b be a realizer and σ be a program variable substitution. Assume σ is admissible for $[\alpha]\phi$ (likewise $\langle\alpha\rangle\phi$). Then $\mathsf{b} \in [\alpha]\phi\,\mathcal{R}\mathbf{z}$ iff $\sigma(\mathsf{b}) \in \sigma([\alpha]\phi)\,\mathcal{R}\mathbf{z}$ and likewise $\mathsf{b} \in \langle\alpha\rangle\phi\,\mathcal{R}\mathbf{z}$ iff $\sigma(\mathsf{b}) \in \sigma(\langle\alpha\rangle\phi)\,\mathcal{R}\mathbf{z}$*

Proof. By simultaneous inductions on α for the Demon and Angel claims. In each case, note that substitution does not change the shape of a realizer, but only substitutes into annotations and term realizers. In cases such as Demonic tests, the realizer typing constraint for $\sigma([?\psi]\phi)$ requires a type annotation $\sigma(\psi)$ for the realizer argument, which is precisely what is provided by $\sigma(\mathsf{b})$. □

Definition A.4 (Substitutions and adjoints)**.** We write σ for any (program variable) substitution, where $\cdot^f_x$ is a notation specifically for the singleton case that replaces x with f. We write $\sigma(e)$ for the result of substituting σ in expression e, which could be a term f, formula ϕ, game α, or realizer b. We write $\sigma^*_\nu(\omega)$ for the adjoint state of ω where the substituted variables of program variable substitution σ are substituted by their values at state ν. We abbreviate $\sigma^*(\omega)$ for the common case $\sigma^*_\omega(\omega)$.

Recall that a region X is a set of possibilities, which are either pairs (b,ω) of realizers b with states ω or one of the pseudopossibilities $\top$ or $\bot$ representing early victory. Substitution is a no-op when applied to $\top$ or $\bot$. For any region X, we write $\sigma^*(X)$ for the adjoint region which applies program variable substitution σ to every possibility in region X. We write $\sigma^L(X)$ for the left adjoint $\sigma^L(X) = \{(\sigma(\mathsf{b}),\omega) \mid (\mathsf{b},\omega) \in X\}$ and $\sigma^R(X)$ for the right adjoint $\sigma^R(X) = \{(\mathsf{b},\sigma^*(\omega)) \mid (\mathsf{b},\omega) \in X\}$. The left adjoint $\sigma^L(X)$ represents the action of program variable substitution σ on the realizers of every possibility in X (which are the left element of the possibility pair) while the right adjoint $\sigma^R(X)$ represents the action of σ on every state (right pair element) in X.

The name "adjoint" is an analogy to the notion of substitution adjoints in the semantics of dL (Platzer, 2017a) and our left and right adjoint operations are not meant to be left or right adjoints in the category-theoretic sense. As in dL (Platzer, 2017a), our substitution lemmas will show that the syntactic effect of substituting in an expression is equivalent to the semantic effect of substituting in the (right) adjoint state. A difference from the dL substitution lemmas (Platzer, 2017a) is that our proof features realizers which must also be substituted (left adjoint). We will specifically show (Lemma A.9) that the semantics of a substituted expression in a left-adjoint region agrees with the meaning of a non-substituted expression in a right-adjoint region.

Lemma A.7 (Substitution distributes over unions)**.** *For all regions A and B and program variable substitutions σ have $\sigma^*(A \cup B) = \sigma^*(A) \cup \sigma^*(B)$*

Proof. The result follows from the realizer claim ($[\![\sigma(\mathsf{b})]\!]\omega = [\![\mathsf{b}]\!]\sigma^*(\omega)$) of Lemma A.9. □

Definition A.5 (Free variables of a substitution)**.** Let σ be a program variable substitution. Let $Dom(\sigma)$ be the set of variables substituted by σ. Then

$$\mathrm{FV}(\sigma) \equiv \bigcup_{x \in Dom(\sigma)} \mathrm{FV}(\sigma\, x)$$

That is, the free variables of a program variable substitution are the union of the free variables of the replacements specified in the substitution.

Lemma A.8 (Adjoints)**.** *Let σ be a program variable substitution. Recall that $Dom(\sigma)$ stands for the set of program variables which are substituted by the program variable substitution σ. We then write $\sigma^*_\nu(\omega)$ for the adjoint state of ω (to program variable substitution σ) where the value of each $x \in Dom(\sigma)$ has been updated to the value of the corresponding $\sigma\, f$ at state ν. We abbreviate $\sigma^*(\omega)$ for the common case $\sigma^*_\omega(\omega)$. Function $\mathrm{FV}(\sigma)$ denotes program term substitution free variables as defined in Def. A.5. If ω equals ν on $\mathrm{FV}(\sigma)$, then $\sigma^*_\omega(\mu) = \sigma^*_\nu(\mu)$ for all μ. We say that σ is U-admissible for expression e iff $\big(\bigcup_{x \in \mathrm{FV}(e) \cap Dom(\sigma)} \mathrm{FV}(\sigma\, x)\big) \cap U = \emptyset$, i.e., if no replacement performed in $\sigma(e)$ mentions any element of U as a free variable.*

- *If ω equals ν on $\mathrm{FV}(\sigma)$, then $\sigma^*_\omega(\mu) = \sigma^*_\nu(\mu)$ for all μ.*
- *If ω equals ν on $U^{\complement}$ and σ is U-admissible for an expression f, α, ϕ, or b, then for all $\mu, \tilde{\mu}$:*
  - $[\![f]\!]\sigma^*_\omega(\mu) = [\![f]\!]\sigma^*_\nu(\mu)$
  - $[\![\mathsf{b}]\!]\sigma^*_\omega(\mu) = [\![\mathsf{b}]\!]\sigma^*_\nu(\mu)$
  - $(\mathsf{b}, \sigma^*_\omega(\mu)) \in [\![\phi]\!]$ *iff* $(\mathsf{b}, \sigma^*_\nu(\mu))[\![\phi]\!]$
  - $(\mathsf{b}, \sigma^*_\omega(\mu)) \in \{(\mathsf{c}, \sigma^*_\omega(\tilde{\mu}))\}\langle\!\langle \alpha \rangle\!\rangle$ *iff* $(\mathsf{b}, \sigma^*_\nu(\mu)) \in \{(\mathsf{c}, \sigma^*_\nu(\tilde{\mu}))\}\langle\!\langle \alpha \rangle\!\rangle$
  - $(\mathsf{b}, \sigma^*_\omega(\mu)) \in \{(\mathsf{c}, \sigma^*_\omega(\tilde{\mu}))\}[\![\alpha]\!]$ *iff* $(\mathsf{b}, \sigma^*_\nu(\mu)) \in \{(\mathsf{c}, \sigma^*_\nu(\tilde{\mu}))\}[\![\alpha]\!]$

Proof. The first claim holds by definition of adjoints and applying Lemma 4.11 on each replacement in σ. The remaining claims hold by induction or coinduction on the expressions, and reduce to the case for program variables. If x is not substituted in σ, the adjoints agree by definition. If it is, then $\mathrm{FV}(\sigma(x)) \subseteq U^{\complement}$ by definition of admissibility, then ω and ν agree on $\mathrm{FV}(\sigma(x))$ by assumption, and the semantics agree by Lemma 4.11. □

Lemma A.9 (Expression substitution)**.** *Assume program variable substitution σ is admissible for each expression in which substitution is applied in the following claims. Then:*

- $[\![\sigma(\mathsf{b})]\!]\omega = [\![\mathsf{b}]\!]\sigma^*(\omega)$
- $[\![\sigma(f)]\!]\omega = [\![f]\!]\sigma^*(\omega)$
- $(\sigma(\mathsf{b}), \omega) \in \{(\sigma(\mathsf{c}), \nu)\}[\![\sigma(\alpha)]\!]$ *iff* $(\mathsf{b}, \sigma^*_\nu(\omega)) \in \{(\mathsf{c}, \sigma^*_\nu(\nu))\}[\![\alpha]\!]$
- $(\sigma(\mathsf{b}), \omega) \in \{(\sigma(\mathsf{c}), \nu)\}\langle\!\langle\sigma(\alpha)\rangle\!\rangle$ *iff* $(\mathsf{b}, \sigma^*_\nu(\omega)) \in \{(\mathsf{c}, \sigma^*_\nu(\nu))\}\langle\!\langle\alpha\rangle\!\rangle$
- $(\sigma(\mathsf{b}), \omega) \in [\![\sigma(\phi)]\!]$ *iff* $(\mathsf{b}, \sigma^*_\omega(\omega)) \in [\![\phi]\!]$

The claim for realizers holds in the case of closed realizers. The generalized claim for open realizers is given in the proof.

Proof. **Realizer cases:**

The claim for realizers uses the same generalized induction principle used in Lemma A.5: we reason about open realizers by assuming a realizer substitution ξ which closes the free realizer variables of the realizer (as opposed to program variable substitution σ which substitutes program variables). We maintain the invariant that the substitution lemma holds for all elements of the realizer substitution ξ, and we allow coinductive hypotheses on realizers as well as inductive hypotheses on smaller types. Under that assumption, we show $[\![\sigma(\xi(\mathsf{b}))]\!]\omega = [\![\xi(\mathsf{b})]\!]\sigma^*(\omega)$ which trivially implies the main realizer claim in the closed case where ξ is an empty substitution $\{\}$.

Case ϵ: Have $[\![\sigma(\xi(\epsilon))]\!]\omega = [\![\epsilon]\!]\omega = () = [\![\epsilon]\!]\sigma^*(\omega) = [\![\xi(\epsilon)]\!]\sigma^*(\omega)$

Case z: Have $[\![\sigma(\xi(z))]\!]\omega = [\![\xi(z)]\!]\sigma^*(\omega)$ by the invariant that substitution holds for all $\xi(z)$ in substitution ξ.

Case f: Have $[\![\sigma(\xi(f))]\!]\omega = [\![\sigma(f)]\!]\omega = [\![f]\!]\sigma^*(\omega) = [\![\xi(f)]\!]\sigma^*(\omega)$ by the substitution claim for terms and because realizer substitutions ξ are no-ops when applied to terms.

Case $\pi_0\mathsf{b}$: Have $[\![\sigma(\xi(\pi_0\mathsf{b}))]\!]\omega = [\![\pi_0(\sigma(\xi(\mathsf{b})))]\!]\omega = \pi_0[\![\sigma(\xi(\mathsf{b}))]\!]\omega = [\![\pi_0(\xi(\mathsf{b}))]\!]\sigma^*(\omega) = [\![\xi(\pi_0\mathsf{b})]\!]\sigma^*(\omega)$.

Case $\pi_1\mathsf{b}$: Have $[\![\sigma(\xi(\pi_1\mathsf{b}))]\!]\omega = [\![\pi_1(\sigma(\xi(\mathsf{b})))]\!]\omega = \pi_1([\![\sigma(\xi(\mathsf{b}))]\!]\omega = [\![\pi_1(\xi(\mathsf{b}))]\!]\sigma^*(\omega) = [\![\xi(\pi_1\mathsf{b})]\!]\sigma^*(\omega)$

Case (b, c): Have $[\![\sigma(\xi((\mathsf{b}, \mathsf{c})))]\!]\omega = [\![(\sigma(\xi(\mathsf{b})), \sigma(\xi(\mathsf{c})))]\!]\omega = ([\![\sigma(\xi(\mathsf{b}))]\!]\omega, [\![\sigma(\xi(\mathsf{c}))]\!]\omega) = ([\![\xi(\mathsf{b})]\!]\sigma^*(\omega), [\![\xi(\mathsf{c})]\!]\sigma^*(\omega)) = [\![(\xi(\mathsf{b}), \xi(\mathsf{c}))]\!]\sigma^*(\omega) = [\![\xi((\mathsf{b}, \mathsf{c}))]\!]\sigma^*(\omega)$.

Case $(\Lambda x : \mathbb{Q}.\ \mathsf{b})$: We reason by transitivity:

$$
\begin{aligned}
& [\![\sigma(\xi(\Lambda x : \mathbb{Q}.\ \mathsf{b}))]\!]\omega \\
&= [\![(\Lambda x : \mathbb{Q}.\ \sigma(\xi(\mathsf{b})))]\!]\omega \\
&= (\Lambda q : \mathbb{Q}.\ [\![\sigma(\xi(\mathsf{b}))]\!]\omega[x \mapsto q]) \\
&\underset{\text{IH}}{=} (\Lambda q : \mathbb{Q}.\ [\![\xi(\mathsf{b})]\!]\sigma^*(\omega[x \mapsto q])) \\
&=_{(*)} (\Lambda q : \mathbb{Q}.\ [\![\xi(\mathsf{b})]\!](\sigma^*(\omega)[x \mapsto q])) \\
&= [\![(\Lambda x : \mathbb{Q}.\ \xi(\mathsf{b}))]\!]\sigma^*(\omega) \\
&= [\![\xi(\Lambda x : \mathbb{Q}.\ \mathsf{b})]\!]\sigma^*(\omega)
\end{aligned}
$$

The step marked (*) relies on the fact that σ was admissible, so that either σ does not bind x or x does not appear free in b.

Case $(\Lambda z : \phi\, \mathcal{R}\mathbf{z}.\ \mathsf{b})$: In this case, have:

$$\begin{aligned}
&[\![\sigma(\xi(\Lambda z : \phi\, \mathcal{R}\mathbf{z}.\ \mathsf{b}))]\!]\omega \\
&= [\![\sigma((\Lambda z : \phi\, \mathcal{R}\mathbf{z}.\ \xi(\mathsf{b})))]\!]\omega \\
&= [\![\Lambda z : \sigma(\phi)\, \mathcal{R}\mathbf{z}.\ \sigma(\xi(\mathsf{b}))]\!]\omega \\
&= (\Lambda w : \sigma(\phi)\, \mathcal{R}\mathbf{z}.\ [\![\sigma(\xi(\mathsf{b}))[z \mapsto w]]\!]\omega) \\
&=_{(*)} (\Lambda w : \phi\, \mathcal{R}\mathbf{z}.\ [\![\sigma(\xi'(\mathsf{b}))]\!]\omega) \\
&\underset{\text{IH}}{=} (\Lambda w : \phi\, \mathcal{R}\mathbf{z}.\ [\![\xi'(\mathsf{b})]\!]\sigma^*(\omega)) \\
&= [\![\Lambda z : \phi\, \mathcal{R}\mathbf{z}.\ \xi(\mathsf{b})]\!]\sigma^*(\omega) \\
&= [\![\xi(\Lambda z : \phi\, \mathcal{R}\mathbf{z}.\ \mathsf{b})]\!]\sigma^*(\omega)
\end{aligned}$$

where ξ' extends ξ with the substitution $[z \mapsto w]$. The proof begins by simplifying ξ and $\sigma(\cdot)$ homomorphically. The denotation of a function realizer is a function returning denotations, so the equality of two denotations is shown extensionally by showing the bodies agree for every argument w. The step marked (*) says the realizer substitution $[z \mapsto w]$ can be moved inside the term substitution $\sigma(\cdot)$ by substituting σ in the type of w. The crucial step is showing that the (co)-IH is applicable. The co-IH is well-founded because it is guarded by a constructor (a λ-abstraction). To satisfy the assumption of the co-IH we must show that the substitution theorem applies to the realizer w. Because any appearance of w would be guarded by the same λ-abstraction, one could attempt a well-foundedness argument, but there is a simpler argument: The type of w is structurally smaller than the type of $(\Lambda z : \phi\, \mathcal{R}\mathbf{z}.\ \mathsf{b})$, so the substitution lemma can be assumed for w by an outer structural induction on types.

Case $(\mathsf{b}\ f)$: We have: $[\![\sigma(\xi(\mathsf{b}\ f))]\!]\omega = [\![\sigma(\xi(\mathsf{b}))\ \sigma(\xi(f))]\!]\omega = [\![\sigma(\xi(\mathsf{b}))\ \sigma(f)]\!]\omega = [\![\sigma(\xi(\mathsf{b}))]\!]\omega\ [\![\sigma(f)]\!]\omega \underset{\text{IH}}{=} [\![\xi(\mathsf{b})]\!]\sigma^*(\omega)\ [\![f]\!]\sigma^*(\omega) = [\![\xi(\mathsf{b})\ f]\!]\sigma^*(\omega) = [\![\xi(\mathsf{b}\ f)]\!]\sigma^*(\omega)$ where the IH step relies on the substitution claim for terms as well as the hypothesis for b. Realizer substitutions ξ have no effect on terms f.

Case $(\mathsf{b}\ \mathsf{c})$: We have: $[\![\sigma(\xi(\mathsf{b}\ \mathsf{c}))]\!]\omega = [\![\sigma(\xi(\mathsf{b}))\ \sigma(\xi(\mathsf{c}))]\!]\omega = [\![\sigma(\xi(\mathsf{b}))]\!]\omega\ [\![\sigma(\xi(\mathsf{c}))]\!]\omega \underset{\text{IH}}{=} [\![\xi(\mathsf{b})]\!]\sigma^*(\omega)\ [\![\sigma(\xi(\mathsf{c}))]\!]\sigma^*(\omega) = [\![\xi(\mathsf{b})\ \xi(\mathsf{c})]\!]\sigma^*(\omega) = [\![\xi(\mathsf{b}\ \mathsf{c})]\!]\sigma^*(\omega)$. The coinductive hypotheses are well-founded since b and c are guarded by an application constructor $\mathsf{b}\ \mathsf{c}$.

Case $(\mathtt{if}\ (f)\ \mathsf{b}\ \mathtt{else}\ \mathsf{c})$: We have

$$\begin{aligned}
&[\![\sigma(\xi(\mathtt{if}\ (f)\ \mathsf{b}\ \mathtt{else}\ \mathsf{c}))]\!]\omega \\
&= [\![\mathtt{if}\ (\sigma(f))\ \sigma(\xi(\mathsf{b}))\ \mathtt{else}\ \sigma(\xi(\mathsf{c}))]\!]\omega \\
&= \mathtt{if}\ ([\![\sigma(f)]\!]\omega)\ [\![\sigma(\xi(\mathsf{b}))]\!]\omega\ \mathtt{else}\ [\![\sigma(\xi(\mathsf{c}))]\!]\omega \\
&\underset{\text{IH}}{=} \mathtt{if}\ ([\![f]\!]\sigma^*(\omega))\ [\![\xi(\mathsf{b})]\!]\sigma^*(\omega)\ \mathtt{else}\ [\![\xi(\mathsf{c})]\!]\sigma^*(\omega) \\
&= [\![\mathtt{if}\ (f)\ \xi(\mathsf{b})\ \mathtt{else}\ \xi(\mathsf{c})]\!]\sigma^*(\omega) \\
&= [\![\xi(\mathtt{if}\ (f)\ \mathsf{b}\ \mathtt{else}\ \mathsf{c})]\!]\sigma^*(\omega)
\end{aligned}$$

Term cases:

The term cases hold by induction on the term syntax. The cases for the binary term connectives all map through homomorphically and all symmetric to one another, but we list each case for the sake of thoroughness.

Case q: Have $[\![\sigma(q)]\!]\omega = q = [\![q]\!]\sigma^*(\omega)$.

Case x: Have $[\![\sigma(x)]\!]\omega = \omega(\sigma(x)) = \omega(x) = \sigma^*(\omega)(x) = [\![x]\!]\sigma^*(\omega)$ in the case $x \notin Dom(\sigma)$ or $[\![\sigma(x)]\!]\omega = [\![\sigma\, x]\!]\omega = \sigma^*(\omega)(x) = [\![x]\!]\sigma^*(\omega)$ in the case $x \in Dom(\sigma)$.

Case $f + g$: Have $[\![\sigma(f+g)]\!]\omega = [\![\sigma(f) + \sigma(g)]\!]\omega = [\![\sigma(f)]\!]\omega + [\![\sigma(g)]\!]\omega = [\![f]\!]\sigma^*(\omega) + [\![g]\!]\sigma^*(\omega) = [\![f+g]\!]\sigma^*(\omega)$.

Case $f \cdot g$: Have $[\![\sigma(f \cdot g)]\!]\omega = [\![\sigma(f) \cdot \sigma(g)]\!]\omega = [\![\sigma(f)]\!]\omega \cdot [\![\sigma(g)]\!]\omega = [\![f]\!]\sigma^*(\omega) \cdot [\![g]\!]\sigma^*(\omega) = [\![f \cdot g]\!]\sigma^*(\omega)$.

Case f div g: We have that: $[\![\sigma(f \text{ div } g)]\!]\omega = [\![\sigma(f) \text{ div } \sigma(g)]\!]\omega = [\![\sigma(f)]\!]\omega \text{ div } [\![\sigma(g)]\!]\omega = [\![f]\!]\sigma^*(\omega) \text{ div } [\![g]\!]\sigma^*(\omega) = [\![f \text{ div } g]\!]\sigma^*(\omega)$.

Case f mod g: Have $[\![\sigma(f \bmod g)]\!]\omega = [\![\sigma(f) \bmod \sigma(g)]\!]\omega = [\![\sigma(f)]\!]\omega \bmod [\![\sigma(g)]\!]\omega = [\![f]\!]\sigma^*(\omega) \bmod [\![g]\!]\sigma^*(\omega) = [\![f \bmod g]\!]\sigma^*(\omega)$.

Angel cases:

Case $x := f$: Have

$$
\begin{aligned}
& (\sigma(\mathsf{b}), \omega) \in \{(\sigma(\mathsf{c}), \nu)\}\langle\!\langle \sigma(x := f)\rangle\!\rangle \\
\text{iff } & (\sigma(\mathsf{b}), \omega) \in \{(\sigma(\mathsf{c}), \nu)\}\langle\!\langle x := \sigma(f)\rangle\!\rangle \\
\text{iff }_{\text{(IH)}} \; & \omega = \nu[x \mapsto [\![\sigma(f)]\!]\nu] \text{ for some } (\sigma(\mathsf{c}), \nu) \in \{(\sigma(\mathsf{c}), \nu)\} \\
\text{iff } & \omega = \nu[x \mapsto [\![f]\!]\sigma^*_\nu(\nu)] \text{ for some } (\mathsf{b}, \nu) \in \{(\mathsf{c}, \nu)\} \\
\text{iff } & \omega = \nu[x \mapsto [\![f]\!]\sigma^*_\nu(\nu)] \text{ for some } (\mathsf{b}, \sigma^*_\nu(\nu)) \in \{(\mathsf{b}, \sigma^*_\nu(\nu))\} \\
\text{iff } & \sigma^*_\nu(\omega) = \sigma^*_\nu(\nu[x \mapsto [\![f]\!]\sigma^*_\nu(\nu)]), \text{ for some } (\mathsf{b}, \sigma^*_\nu(\nu)) \in \{(\mathsf{c}, \sigma^*_\nu(\nu))\} \\
\text{iff }_{(*)} \; & \sigma^*_\nu(\omega) = \sigma^*_\nu(\nu)[x \mapsto [\![f]\!]\sigma^*_\nu(\nu)], \text{ for some } (\mathsf{b}, \sigma^*_\nu(\nu)) \in \{(\mathsf{c}, \sigma^*_\nu(\nu))\} \\
\text{iff } & (\mathsf{b}, \sigma^*_\nu(\omega)) \in \{(\mathsf{c}, \sigma^*_\nu(\nu))\}\langle\!\langle x := f\rangle\!\rangle
\end{aligned}
$$

where the step marked (IH) holds by the IH on f and the starred step by admissibility of σ, i.e., by $x \notin Dom(\sigma)$.

Case $x := *$: Have

$$
\begin{aligned}
& (\sigma(\mathsf{c}), \omega) \in \{(\sigma((\mathsf{b}, \mathsf{c})), \nu)\}\langle\!\langle \sigma(x := *)\rangle\!\rangle \\
\text{iff } & (\sigma(\mathsf{c}), \omega) \in \{(\sigma((\mathsf{b}, \mathsf{c})), \nu)\}\langle\!\langle x := *\rangle\!\rangle \\
\text{iff } & \omega = \nu[x \mapsto [\![\sigma(\mathsf{b})]\!]\nu], \text{ for some } (\sigma((\mathsf{b}, \mathsf{c})), \nu) \in \{(\sigma((\mathsf{b}, \mathsf{c})), \nu)\} \\
\text{iff }_{\text{(IH)}} \; & \omega = \nu[x \mapsto [\![\mathsf{b}]\!]\sigma^*_\nu(\nu)], \text{ for some } (\sigma((\mathsf{b}, \mathsf{c})), \nu) \in \{(\sigma((\mathsf{b}, \mathsf{c})), \nu)\} \\
\text{iff } & \omega = \nu[x \mapsto [\![\mathsf{b}]\!]\sigma^*_\nu(\nu)], \text{ for some } ((\mathsf{b}, \mathsf{c}), \sigma^*_\nu(\nu)) \in \{((\mathsf{b}, \mathsf{c}), \sigma^*_\nu(\nu))\} \\
\text{iff } & \sigma^*_\nu(\omega) = \sigma^*_\nu(\nu[x \mapsto [\![\mathsf{b}]\!]\sigma^*_\nu(\nu)]), \text{ for some } ((\mathsf{b}, \mathsf{c}), \sigma^*(\nu)) \in \{((\mathsf{b}, \mathsf{c}), \sigma^*_\nu(\nu))\} \\
\text{iff }_{(*)} \; & \sigma^*_\nu(\omega) = \sigma^*_\nu(\nu)[x \mapsto [\![\mathsf{b}]\!]\sigma^*_\nu(\nu)], \text{ for some } ((\mathsf{b}, \mathsf{c}), \sigma^*_\nu(\nu)) \in \{((\mathsf{b}, \mathsf{c}), \sigma^*_\nu(\nu))\} \\
\text{iff } & (\mathsf{c}, \sigma^*_\nu(\omega)) \in \{((\mathsf{b}, \mathsf{c}), \sigma^*_\nu(\nu))\}\langle\!\langle x := f\rangle\!\rangle
\end{aligned}
$$

where the step marked (IH) holds by the IH on f and the starred step by admissibility of σ, i.e., by $x \notin Dom(\sigma)$.

Case $?\phi$: Have

$$\begin{aligned}
&(\sigma(\mathsf{c}),\omega) \in \{(\sigma((\mathsf{b},\mathsf{c})),\omega)\}\langle\!\langle \sigma(?\phi)\rangle\!\rangle \\
\text{iff } &(\sigma(\mathsf{b}),\omega) \in [\![\sigma(\phi)]\!] \text{ and } ((\mathsf{b},\mathsf{c}),\omega) \in \{((\mathsf{b},\mathsf{c}),\omega)\} \\
\text{iff } &(\mathsf{b},\sigma^*_\omega(\omega)) \in [\![\phi]\!] \text{ and } ((\mathsf{b},\mathsf{c}),\sigma^*_\omega(\omega)) \in \{((\mathsf{b},\mathsf{c}),\sigma^*_\omega(\omega))\} \\
\text{iff } &(\mathsf{b},\sigma^*_\omega(\omega)) \in \{((\mathsf{b},\mathsf{c}),\sigma^*_\omega(\omega))\}\langle\!\langle ?\phi\rangle\!\rangle
\end{aligned}$$

Case $\alpha;\beta$: Recall the assumption that σ is admissible for $\alpha;\beta$, which includes the assumption (NB) that no substituted variable of σ is bound in $\alpha;\beta$. Have

$$\begin{aligned}
&(\sigma(\mathsf{b}),\omega) \in \{(\sigma(\mathsf{c}),\nu)\}\langle\!\langle \sigma(\alpha;\beta)\rangle\!\rangle \\
\text{iff } &(\sigma(\mathsf{b}),\omega) \in (\{(\sigma(\mathsf{c}),\nu)\}\langle\!\langle \sigma(\alpha)\rangle\!\rangle)\langle\!\langle \sigma(\beta)\rangle\!\rangle \\
\text{iff } &(\sigma(\mathsf{b}),\omega) \in \{(\sigma(\mathsf{d}),\mu) \mid \{(\sigma(\mathsf{d}),\mu)\} \in (\{(\sigma(\mathsf{c}),\nu)\}\langle\!\langle \sigma(\alpha)\rangle\!\rangle)\}\langle\!\langle \sigma(\beta)\rangle\!\rangle \\
\text{iff } &(\mathsf{b},\sigma^*_\mu(\omega)) \in \{(\mathsf{d},\sigma^*_\mu(\mu)) \mid \{(\sigma(\mathsf{d}),\mu)\} \in (\{(\sigma(\mathsf{c}),\nu)\}\langle\!\langle \sigma(\alpha)\rangle\!\rangle)\}\langle\!\langle \beta\rangle\!\rangle\} && \text{by IH on } \beta \\
\text{iff } &(\mathsf{b},\sigma^*_\mu(\omega)) \in \{(\mathsf{d},\sigma^*_\mu(\mu)) \mid \{(\mathsf{d},\sigma^*_\nu(\mu))\} \in (\{(\mathsf{c},\sigma^*_\nu(\nu))\}\langle\!\langle \alpha\rangle\!\rangle)\}\langle\!\langle \beta\rangle\!\rangle && \text{by IH on } \alpha \\
\text{iff } &(\mathsf{b},\sigma^*_\nu(\omega)) \in \{(\mathsf{d},\sigma^*_\nu(\mu)) \mid \{(\mathsf{d},\sigma^*_\nu(\mu))\} \in (\{(\mathsf{c},\sigma^*_\nu(\nu))\}\langle\!\langle \alpha\rangle\!\rangle)\}\langle\!\langle \beta\rangle\!\rangle && \text{by Lemma A.8} \\
\text{iff } &(\mathsf{b},\sigma^*_\nu(\omega)) \in (\{(\mathsf{c},\sigma^*_\nu(\nu))\}\langle\!\langle \alpha\rangle\!\rangle)\langle\!\langle \beta\rangle\!\rangle \\
\text{iff } &(\mathsf{b},\sigma^*_\nu(\omega)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}\langle\!\langle \alpha;\beta\rangle\!\rangle
\end{aligned}$$

In the IHs on α and β, the requirements that σ be admissible for α and β hold as a direct consequence of our assumption that σ is admissible for $\alpha;\beta$ because the definition of admissibility requires that all admissibility checks hold recursively in all subprograms, i.e., in α and β. In the application of Lemma A.8, its admissibility assumption requires that σ is $\mathrm{BV}(\sigma(\alpha))$-admissible for β, which follows from admissibility for β because $\mathrm{BV}(\sigma(\alpha)) = \mathrm{BV}(\alpha)$ by (NB). The other assumption of Lemma A.8 is that ν and μ agree on $\mathrm{BV}(\alpha)^{\complement}$, which holds by Lemma 4.12.

Case $\alpha \cup \beta$: Have

$$\begin{aligned}
&(\sigma(\mathsf{b}),\omega) \in \{(\sigma(\mathsf{c}),\nu)\}\langle\!\langle \sigma(\alpha\cup\beta)\rangle\!\rangle \\
\text{iff } &(\sigma(\mathsf{b}),\omega) \in (\{(\sigma(\mathsf{c}),\nu)\})_{\langle 0\rangle}\langle\!\langle \sigma(\alpha)\rangle\!\rangle \cup (\{(\sigma(\mathsf{c}),\nu)\})_{\langle 1\rangle}\langle\!\langle \sigma(\beta)\rangle\!\rangle \\
\text{iff } &{}_{(*)}(\sigma(\mathsf{b}),\omega) \in (\sigma^L_\nu((\{(\mathsf{c},\nu)\}_{\langle 0\rangle})))\langle\!\langle \sigma(\alpha)\rangle\!\rangle \cup (\sigma^L_\nu((\{(\mathsf{c},\nu)\}_{\langle 1\rangle})))\langle\!\langle \sigma(\beta)\rangle\!\rangle \\
\text{iff } &(\mathsf{b},\sigma^*_\nu(\omega)) \in (\sigma^R_\nu(\{(\mathsf{c},\nu)\}_{\langle 0\rangle}))\langle\!\langle \alpha\rangle\!\rangle \cup (\sigma^R_\nu(\{(\mathsf{c},\nu)\}_{\langle 1\rangle}))\langle\!\langle \beta\rangle\!\rangle && \text{IHs on } \alpha,\beta \\
\text{iff } &{}_{(*)}(\mathsf{b},\sigma^*_\nu(\omega)) \in ((\{(\mathsf{c},\sigma^*_\nu(\nu))\})_{\langle 0\rangle})\langle\!\langle \alpha\rangle\!\rangle \cup (\{(\mathsf{c},\sigma^*_\nu(\nu))\}_{\langle 1\rangle})\langle\!\langle \beta\rangle\!\rangle \\
\text{iff } &(\mathsf{b},\sigma^*_\nu(\omega)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}\langle\!\langle \alpha\cup\beta\rangle\!\rangle
\end{aligned}$$

where the steps marked (*) use the inductive hypothesis on $\pi_0\mathsf{b}$ and the step marked IH uses the inductive hypotheses on α and β.

Case α^*: Below, the notation $\sigma^R(Z)$ indicates $\{(\mathsf{b},\sigma^*_\omega(\omega)) \mid (\mathsf{b},\omega) \in Z\}$, i.e., the adjoint is applied individually at every state in Z, while $\sigma^R_\mu(Z)$ indicates $\{(\mathsf{b},\sigma^*_\mu(\omega)) \mid (\mathsf{b},\omega) \in Z\}$, meaning every adjoint is uniformly applied at state μ. Below, b ranges over realizers that meet the appropriate typing constraints and ω ranges over states.

By Lemma 4.8, we have for all b and all ω that $(\sigma(\mathsf{b}),\omega) \in \{(\sigma(\mathsf{c}),\nu)\}\langle\!\langle\sigma(\alpha^*)\rangle\!\rangle$ iff $(\sigma(\mathsf{b}),\omega) \in \bigcup_{k\in\mathbb{N}}(\{(\sigma(\mathsf{c}),\nu)\}\langle\!\langle\sigma(\alpha)\rangle\!\rangle^k)_{\langle 0\rangle}$. The crucial step will show for all b and ω that (Equivs) $(\sigma(\mathsf{b}),\omega) \in \bigcup_{k\in\mathbb{N}}(\{(\sigma(\mathsf{c}),\nu)\}\langle\!\langle\alpha\rangle\!\rangle^k)_{\langle 0\rangle}$ iff $(\mathsf{b},\sigma^*_\nu(\omega)) \in \bigcup_{k\in\mathbb{N}}(\{(\mathsf{c},\sigma^*_\nu(\nu))\}\langle\!\langle\alpha\rangle\!\rangle^k)_{\langle 0\rangle}$ which will imply the claim by a second application of Lemma 4.8 which shows $(\mathsf{b},\sigma^*_\nu(\omega)) \in \bigcup_{k\in\mathbb{N}}(\{(\mathsf{c},\sigma^*_\nu(\nu))\}\langle\!\langle\alpha\rangle\!\rangle^k)_{\langle 0\rangle}$ iff $(\mathsf{b},\sigma^*_\nu(\omega)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}\langle\!\langle\alpha^*\rangle\!\rangle$.

To show (Equivs) it suffices to show (Equiv) $(\sigma(\hat{\mathsf{b}}),\omega) \in \{(\sigma(\mathsf{c}),\nu)\}\langle\!\langle\alpha\rangle\!\rangle^k$ iff $(\hat{\mathsf{b}},\sigma^*_\nu(\omega)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}\langle\!\langle\alpha\rangle\!\rangle^k$ for all $k\in\mathbb{N}$ and all (well-typed realizers) b and (states) ω. Fact (Equiv) suffices to show (Equivs) by mapping through the application of $\cdot_{\langle 0\rangle}$ and the union. Note the use of variable name $\hat{\mathsf{b}}$ to distinguish from realizer b: realizer $\hat{\mathsf{b}}$ and realizer b from (Equivs) differ by an application of $\cdot_{\langle 0\rangle}$.

We now show the subclaim (Equiv) by induction on k, letting $\hat{\mathsf{b}}$ and ω vary throughout the induction: $(\sigma(\hat{\mathsf{b}}),\omega) \in \{(\sigma(\mathsf{c}),\nu)\}\langle\!\langle\alpha\rangle\!\rangle^k$ iff $(\hat{\mathsf{b}},\sigma^*_\nu(\omega)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}\langle\!\langle\alpha\rangle\!\rangle^k$.

In the base case, $k=0$. Have $(\sigma(\hat{\mathsf{b}}),\omega) \in \{(\sigma(\mathsf{c}),\nu)\}\langle\!\langle\alpha\rangle\!\rangle^k$ iff (by definition) $(\sigma(\hat{\mathsf{b}}),\omega) \in \{(\sigma(\mathsf{c}),\nu)\}$ iff $\hat{\mathsf{b}} = \mathsf{c}$ and $\omega = \nu$ iff $(\hat{\mathsf{b}},\sigma^*_\nu(\omega)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}$ iff (by definition) $(\hat{\mathsf{b}},\sigma^*_\nu(\omega)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}$.

In the inductive case, $k = j+1$ for some $j\in\mathbb{N}$. Assume the IH that (for all well-typed d and μ) $(\sigma(\mathsf{d}),\mu) \in \{(\sigma(\mathsf{c}),\nu)\}\langle\!\langle\alpha\rangle\!\rangle^j$ iff $(\mathsf{d},\sigma^*_\nu(\mu)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}\langle\!\langle\alpha\rangle\!\rangle^j$.

Now show the case.

$$
\begin{array}{ll}
(\sigma(\hat{\mathsf{b}}),\omega) \in \{(\sigma(\mathsf{c}),\nu)\}\langle\!\langle\sigma(\alpha)\rangle\!\rangle^{j+1} & \\
\text{iff } (\sigma(\hat{\mathsf{b}}),\omega) \in (\{(\sigma(\mathsf{c}),\nu)\}\langle\!\langle\sigma(\alpha)\rangle\!\rangle^j)_{\langle 1\rangle}\langle\!\langle\sigma(\alpha)\rangle\!\rangle & \text{By definition} \\
\text{iff } (\sigma(\hat{\mathsf{b}}),\omega) \in \{(\sigma(\mathsf{d}),\mu) \mid (\sigma(\mathsf{d}),\mu) \in (\{(\sigma(\mathsf{c}),\nu)\}\langle\!\langle\sigma(\alpha)\rangle\!\rangle^j)\}_{\langle 1\rangle}\langle\!\langle\sigma(\alpha)\rangle\!\rangle & \\
\text{iff } (\hat{\mathsf{b}},\sigma^*_\mu(\omega)) \in \{(\mathsf{d},\sigma^*_\mu(\mu)) \mid (\sigma(\mathsf{d}),\mu) \in (\{(\sigma(\mathsf{c}),\nu)\}\langle\!\langle\sigma(\alpha)\rangle\!\rangle^j)\}_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle & \text{By outer IH} \\
\text{iff } (\hat{\mathsf{b}},\sigma^*_\mu(\omega)) \in \{(\mathsf{d},\sigma^*_\mu(\mu)) \mid (\mathsf{d},\sigma^*_\nu(\mu)) \in (\{(\mathsf{c},\sigma^*_\nu(\nu))\}\langle\!\langle\alpha\rangle\!\rangle^j)\}_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle & \text{By inner IH} \\
\text{iff } (\hat{\mathsf{b}},\sigma^*_\nu(\omega)) \in \{(\mathsf{d},\sigma^*_\nu(\mu)) \mid (\mathsf{d},\sigma^*_\nu(\mu)) \in (\{(\mathsf{c},\sigma^*_\nu(\nu))\}\langle\!\langle\alpha\rangle\!\rangle^j)\}_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle & \text{by Lemma A.8} \\
\text{iff } (\hat{\mathsf{b}},\sigma^*_\nu(\omega)) \in (\{(\mathsf{c},\sigma^*_\nu(\nu))\}\langle\!\langle\alpha\rangle\!\rangle^j)_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle & \\
\text{iff } (\hat{\mathsf{b}},\sigma^*_\nu(\omega)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}\langle\!\langle\alpha\rangle\!\rangle^{j+1} & \text{By definition}
\end{array}
$$

The application of Lemma A.8 requires that $\nu = \mu$ on $\mathrm{BV}(\alpha^*)^\complement$, since the admissibility assumption for loops says σ is $\mathrm{BV}(\sigma(\alpha))$-admissible in α and because $\mathrm{BV}(\sigma(\alpha)) = \mathrm{BV}(\alpha)$ because admissible program variable substitutions never change the bound variables of a game. The fact $\nu = \mu$ on $\mathrm{BV}(\alpha^*)^\complement$ holds by inductively generalizing Lemma 4.12 to $\{(\mathsf{c},\sigma^*_\nu(\nu))\}\langle\!\langle\alpha\rangle\!\rangle^j$ by repeating it j times and observing $\mathrm{BV}(\alpha^*) = \mathrm{BV}(\alpha)$ by definition.

This completes the inductive case of the inner induction, thus completes the inner induction and the case.

Case α^d: Have $(\sigma(\mathsf{b}),\omega) \in \{(\sigma(\mathsf{c}),\nu)\}\langle\!\langle\sigma(\alpha^d)\rangle\!\rangle$ iff $(\sigma(\mathsf{b}),\omega) \in \{(\sigma(\mathsf{c}),\nu)\}[\![\sigma(\alpha)]\!]$ iff $(\mathsf{b},\sigma^*_\nu(\omega)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}[\![\alpha]\!]$ iff $(\mathsf{b},\sigma^*_\nu(\omega)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}\langle\!\langle\alpha^d\rangle\!\rangle$.

Demon cases:

Case $x := f$: Have

$$\begin{aligned}
&(\sigma(\mathsf{b}),\omega) \in \{(\sigma(\mathsf{c}),\nu)\}[\![\sigma(x := f)]\!]\\
\text{iff } &(\sigma(\mathsf{b}),\omega) \in \{(\sigma(\mathsf{c}),\nu)\}[\![x := \sigma(f)]\!]\\
\text{iff }_{\text{(IH)}} &\ \omega = \nu[x \mapsto [\![\sigma(f)]\!]\nu],\ \text{for some } (\sigma(\mathsf{c}),\nu) \in \{(\sigma(\mathsf{c}),\nu)\}\\
\text{iff } &\ \omega = \nu[x \mapsto [\![f]\!]\sigma^*_\nu(\nu)],\ \text{for some } (\mathsf{c},\nu) \in \{(\mathsf{c},\nu)\}\\
\text{iff } &\ \omega = \nu[x \mapsto [\![f]\!]\sigma^*_\nu(\nu)],\ \text{for some } (\mathsf{c},\sigma^*_\nu(\nu)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}\\
\text{iff } &\ \sigma^*_\nu(\omega) = \sigma^*_\nu(\nu[x \mapsto [\![f]\!]\sigma^*_\nu(\nu)]),\ \text{for some } (\mathsf{c},\sigma^*_\nu(\nu)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}\\
\text{iff }_{(*)} &\ \sigma^*(\omega) = \sigma^*_\nu(\nu)[x \mapsto [\![f]\!]\sigma^*_\nu(\nu)],\ \text{for some } (\mathsf{c},\sigma^*_\nu(\nu)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}\\
\text{iff } &\ (\mathsf{b},\sigma^*_\nu(\omega)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}[\![x := f]\!]
\end{aligned}$$

where the step marked (IH) holds by the IH on f and the starred step by admissibility of σ, i.e., by $x \notin \mathit{Dom}(\sigma)$.

Case $x := *$: Have $(\sigma(\mathsf{b}),\omega) \in \{(\sigma(\mathsf{c}),\nu)\}[\![\sigma(x := *)]\!]$ iff $(\sigma(\mathsf{b}),\omega) \in \{(\sigma(\mathsf{c}),\nu)\}[\![x := *]\!]$ iff $\omega = \nu[x \mapsto v]$ and $\sigma(\mathsf{b}) = \sigma(\mathsf{c})\ v$ for some $(\sigma(\mathsf{c}),\nu) \in \{(\sigma(\mathsf{c}),\nu)\}, v \in \mathbb{Q}$ iff $\omega = \nu[x \mapsto v]$ and $\mathsf{b} = \mathsf{c}\ v$ for some $(\mathsf{c},\sigma^*_\nu(\nu)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}, v \in \mathbb{Q}$ iff $\sigma^*(\omega) = \sigma^*(\nu[x \mapsto v])$ and $\mathsf{b} = \mathsf{c}\ v$ for some $(\mathsf{b},\sigma^*_\nu(\nu)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}, v \in \mathbb{Q}$ iff$_{(*)}$ $\sigma^*_\nu(\omega) = \sigma^*_\nu(\nu)[x \mapsto v]$ and $\mathsf{b} = \mathsf{c}\ v$ for some $(\mathsf{c},\sigma^*(\nu)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}$ iff $(\mathsf{b},\sigma^*_\nu(\omega)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}[\![x := *]\!]$ by the IH on f and (in the starred step) since by admissibility of σ have $x \notin \mathit{Dom}(\sigma)$.

Case $?\phi$: Have

$$\begin{aligned}
&(\sigma(\mathsf{b}\,\mathsf{c}),\omega) \in \{(\sigma(\mathsf{b}),\omega)\}[\![\sigma(?\phi)]\!]\\
\text{iff } &(\sigma(\mathsf{b}),\omega) \in \{(\sigma(\mathsf{b}),\omega)\} \text{ assuming } (\sigma(\mathsf{c}),\omega) \in [\![\sigma(\phi)]\!]\\
\text{iff } &(\sigma(\mathsf{b}),\omega) \in \{(\sigma(\mathsf{b}),\omega)\} \text{ assuming } (\mathsf{c},\sigma^*_\omega(\omega)) \in [\![\phi]\!]\\
\text{iff } &(\mathsf{b},\sigma^*_\omega(\omega)) \in \{(\mathsf{b},\sigma^*_\omega(\omega))\} \text{ assuming } (\mathsf{c},\sigma^*_\omega(\omega)) \in [\![\phi]\!]\\
\text{iff } &(\mathsf{b}\,\mathsf{c},\sigma^*_\omega(\omega)) \in \{(\mathsf{b},\sigma^*_\omega(\omega))\}[\![?\phi]\!]
\end{aligned}$$

Case $\alpha;\beta$: Have

$$\begin{aligned}
&(\sigma(\mathsf{b}),\omega) \in \{(\sigma(\mathsf{c}),\nu)\}[\![\sigma(\alpha;\beta)]\!]\\
\text{iff } &(\sigma(\mathsf{b}),\omega) \in (\{(\sigma(\mathsf{c}),\nu)\}[\![\sigma(\alpha)]\!])[\![\sigma(\beta)]\!]\\
\text{iff } &(\sigma(\mathsf{b}),\omega) \in \{(\sigma(\mathsf{d}),\mu) \mid \{(\sigma(\mathsf{d}),\mu)\} \in (\{(\sigma(\mathsf{c}),\nu)\}[\![\sigma(\alpha)]\!])\}[\![\sigma(\beta)]\!]\\
\text{iff } &(\mathsf{b},\sigma^*_\mu(\omega)) \in \{(\mathsf{d},\sigma^*_\mu(\mu)) \mid \{(\sigma(\mathsf{d}),\mu)\} \in (\{(\sigma(\mathsf{c}),\nu)\}[\![\sigma(\alpha)]\!])\}[\![\beta]\!]\}\\
\text{iff } &(\mathsf{b},\sigma^*_\mu(\omega)) \in \{(\mathsf{d},\sigma^*_\mu(\mu)) \mid \{(\mathsf{d},\sigma^*_\nu(\mu))\} \in (\{(\mathsf{c},\sigma^*_\nu(\nu))\}[\![\alpha]\!])\}[\![\beta]\!]\\
\text{iff}_{(*)} &(\mathsf{b},\sigma^*_\nu(\omega)) \in \{(\mathsf{d},\sigma^*_\nu(\mu)) \mid \{(\mathsf{d},\sigma^*_\nu(\mu))\} \in (\{(\mathsf{c},\sigma^*_\nu(\nu))\}[\![\alpha]\!])\}[\![\beta]\!]\\
\text{iff } &(\mathsf{b},\sigma^*_\nu(\omega)) \in (\{(\mathsf{c},\sigma^*_\nu(\nu))\}[\![\alpha]\!])[\![\beta]\!]\\
\text{iff } &(\mathsf{b},\sigma^*_\nu(\omega)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}[\![\alpha;\beta]\!]
\end{aligned}$$

using the IHs on α and β. The starred step holds by Lemma A.8 which is applicable because the admissibility assumption says σ is $\mathrm{BV}(\sigma(\alpha))$-admissible for β, because $\mathrm{BV}(\sigma(\alpha)) = \mathrm{BV}(\alpha)$, and because ν and μ agree on $\mathrm{BV}(\alpha)^{\complement}$ by Lemma 4.12.

Case $\alpha \cup \beta$: Have

$$
\begin{aligned}
&(\sigma(\mathsf{b}), \omega) \in \{(\sigma(\mathsf{c}), \nu)\}[\![\sigma(\alpha \cup \beta)]\!] \\
\text{iff } &(\sigma(\mathsf{b}), \omega) \in (\{(\sigma(\mathsf{c}), \nu)\})_{[0]}[\![\sigma(\alpha)]\!] \cup (\{(\sigma(\mathsf{c}), \nu)\})_{[1]}[\![\sigma(\beta)]\!] \\
\text{iff } &_{(*)}(\sigma(\mathsf{b}), \omega) \in (\sigma^L_\nu((\{(\mathsf{c}, \nu)\}_{[0]})))[\![\sigma(\alpha)]\!] \cup (\sigma^L_\nu((\{(\mathsf{c}, \nu)\}_{[1]})))[\![\sigma(\beta)]\!] \\
\text{iff } &_{(IH)}(\mathsf{b}, \sigma^*_\nu(\omega)) \in (\sigma^R_\nu(\{(\mathsf{c}, \nu)\}_{[0]}))[\![\alpha]\!] \cup (\sigma^R_\nu(\{(\mathsf{c}, \nu)\}_{[1]}))[\![\beta]\!] \\
\text{iff } &_{(*)}(\mathsf{b}, \sigma^*_\nu(\omega)) \in ((\{(\mathsf{c}, \sigma^*_\nu(\nu))\})_{[0]})[\![\alpha]\!] \cup (\{(\mathsf{c}, \sigma^*_\nu(\nu))\}_{[1]})[\![\beta]\!] \\
\text{iff } &(\mathsf{b}, \sigma^*_\nu(\omega)) \in \{(\mathsf{c}, \sigma^*_\nu(\nu))\}[\![\alpha \cup \beta]\!]
\end{aligned}
$$

where the steps marked (*) use the inductive hypothesis on $\pi_0\mathsf{b}$ and the step marked IH uses the inductive hypotheses on α and β.

Case α^*:

Below, the notation $\sigma^R(Z)$ indicates $\{(\mathsf{b}, \sigma^*_\omega(\omega)) \mid (\mathsf{b}, \omega) \in Z\}$, i.e., the adjoint is applied individually at every state in Z, while $\sigma^R_\mu(Z)$ indicates $\{(\mathsf{b}, \sigma^*_\mu(\omega)) \mid (\mathsf{b}, \omega) \in Z\}$, meaning every adjoint is uniformly applied at state μ. Below, b ranges over realizers that meet the appropriate typing constraints and ω ranges over states.

By Lemma 4.8, have for all b and ω that $(\sigma(\mathsf{b}), \omega) \in \{(\sigma(\mathsf{c}), \nu)\}[\![\sigma(\alpha^*)]\!]$ iff $(\sigma(\mathsf{b}), \omega) \in \bigcup_{k \in \mathbb{N}}(\{(\sigma(\mathsf{c}), \nu)\}[\![\sigma(\alpha)]\!]^k)_{[0]}$. The heart of the proof for this case will show for all b and ω that (Equivs) $(\sigma(\mathsf{b}), \omega) \in \bigcup_{k \in \mathbb{N}}(\{(\sigma(\mathsf{c}), \nu)\}[\![\alpha]\!]^k)_{[0]}$ iff $(\mathsf{b}, \sigma^*_\nu(\omega)) \in \bigcup_{k \in \mathbb{N}}(\{(\mathsf{c}, \sigma^*_\nu(\nu))\}[\![\alpha]\!]^k)_{[0]}$ which will imply the claim by a second application of Lemma 4.8 which shows for all b and ω that $(\mathsf{b}, \sigma^*_\nu(\omega)) \in \bigcup_{k \in \mathbb{N}}(\{(\mathsf{c}, \sigma^*_\nu(\nu))\}[\![\alpha]\!]^k)_{[0]}$ iff $(\mathsf{b}, \sigma^*_\nu(\omega)) \in \{(\mathsf{c}, \sigma^*_\nu(\nu))\}[\![\alpha^*]\!]$.

To show (Equivs) it suffices to show (Equiv) $(\sigma(\hat{\mathsf{b}}), \omega) \in \{(\sigma(\mathsf{c}), \nu)\}[\![\alpha]\!]^k$ iff $(\hat{\mathsf{b}}, \sigma^*_\nu(\omega)) \in \{(\mathsf{c}, \sigma^*_\nu(\nu))\}[\![\alpha]\!]^k$ for all $k \in \mathbb{N}$ and all (well-typed realizers) $\hat{\mathsf{b}}$ and (states) ω. Fact (Equiv) suffices to show (Equivs) by mapping through the application of $\cdot_{[0]}$ and the union. We use variable name $\hat{\mathsf{b}}$ to emphasize that it is differs from b from (Equiv), specifically they differ by an application of $\cdot_{[0]}$.

We now show the subclaim (Equiv) by induction on k, letting $\hat{\mathsf{b}}$ and ω throughout the induction: $(\sigma(\hat{\mathsf{b}}), \omega) \in \{(\sigma(\mathsf{c}), \nu)\}[\![\alpha]\!]^k$ iff $(\hat{\mathsf{b}}, \sigma^*_\nu(\omega)) \in \{(\mathsf{c}, \sigma^*_\nu(\nu))\}[\![\alpha]\!]^k$.

In the base case, $k = 0$. Have $(\sigma(\hat{\mathsf{b}}), \omega) \in \{(\sigma(\mathsf{c}), \nu)\}[\![\alpha]\!]^k$ iff (by definition) $(\sigma(\hat{\mathsf{b}}), \omega) \in \{(\sigma(\mathsf{c}), \nu)\}$ iff $\hat{\mathsf{b}} = \mathsf{c}$ and $\omega = \nu$ iff $(\hat{\mathsf{b}}, \sigma^*_\nu(\omega)) \in \{(\mathsf{c}, \sigma^*_\nu(\nu))\}$ iff (by definition) $(\hat{\mathsf{b}}, \sigma^*_\nu(\omega)) \in \{(\mathsf{c}, \sigma^*_\nu(\nu))\}$.

In the inductive case, $k = j + 1$ for some $k \in \mathbb{N}$. Assume the IH that (for all well-typed d and μ) $(\sigma(\mathsf{d}), \mu) \in \{(\sigma(\mathsf{c}), \nu)\}[\![\alpha]\!]^j$ iff $(\mathsf{d}, \sigma^*_\nu(\mu)) \in \{(\mathsf{c}, \sigma^*_\nu(\nu))\}[\![\alpha]\!]^j$.

Now show the case.

$$
\begin{array}{lll}
& (\sigma(\hat{\mathsf{b}}),\omega) \in \{(\sigma(\mathsf{c}),\nu)\}[\![\sigma(\alpha)]\!]^{j+1} & \\
\text{iff } & (\sigma(\hat{\mathsf{b}}),\omega) \in (\{(\sigma(\mathsf{c}),\nu)\}[\![\sigma(\alpha)]\!]^{j})_{[1]}[\![\sigma(\alpha)]\!] & \text{By definition} \\
\text{iff } & (\sigma(\hat{\mathsf{b}}),\omega) \in \{(\sigma(\mathsf{d}),\mu) \mid (\sigma(\mathsf{d}),\mu) \in (\{(\sigma(\mathsf{c}),\nu)\}[\![\sigma(\alpha)]\!]^{j})\}_{[1]}[\![\sigma(\alpha)]\!] & \\
\text{iff } & (\hat{\mathsf{b}},\sigma^*_\mu(\omega)) \in \{(\mathsf{d},\sigma^*_\mu(\mu)) \mid (\sigma(\mathsf{d}),\mu) \in (\{(\sigma(\mathsf{c}),\nu)\}[\![\sigma(\alpha)]\!]^{j})\}_{[1]}[\![\alpha]\!] & \text{By outer IH} \\
\text{iff } & (\hat{\mathsf{b}},\sigma^*_\mu(\omega)) \in \{(\mathsf{d},\sigma^*_\mu(\mu)) \mid (\mathsf{d},\sigma^*_\nu(\mu)) \in (\{(\mathsf{c},\sigma^*_\nu(\nu))\}[\![\alpha]\!]^{j})\}_{[1]}[\![\alpha]\!] & \text{By inner IH} \\
\text{iff } & (\hat{\mathsf{b}},\sigma^*_\nu(\omega)) \in \{(\mathsf{d},\sigma^*_\nu(\mu)) \mid (\mathsf{d},\sigma^*_\nu(\mu)) \in (\{(\mathsf{c},\sigma^*_\nu(\nu))\}[\![\alpha]\!]^{j})\}_{[1]}[\![\alpha]\!] & \text{by Lemma A.8} \\
\text{iff } & (\hat{\mathsf{b}},\sigma^*_\nu(\omega)) \in (\{(\mathsf{c},\sigma^*_\nu(\nu))\}[\![\alpha]\!]^{j})_{[1]}[\![\alpha]\!] & \\
\text{iff } & (\hat{\mathsf{b}},\sigma^*_\nu(\omega)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}[\![\alpha]\!]^{j+1} & \text{By definition}
\end{array}
$$

The application of Lemma A.8 requires that $\nu = \mu$ on $\mathrm{BV}(\alpha^*)^{\complement}$, since the admissibility assumption for loops says σ is $\mathrm{BV}(\sigma(\alpha))$-admissible in α and because $\mathrm{BV}(\sigma(\alpha)) = \mathrm{BV}(\alpha)$ because admissible program variable substitutions never change the bound variables of a game. The fact $\nu = \mu$ on $\mathrm{BV}(\alpha^*)^{\complement}$ holds by inductively generalizing Lemma 4.12 to $\{(\mathsf{c},\sigma^*_\nu(\nu))\}[\![\alpha]\!]^j$ by repeating it j times and observing $\mathrm{BV}(\alpha^*) = \mathrm{BV}(\alpha)$ by definition.

This completes the inductive case of the inner induction, thus completes the inner induction and the case.

Case α^d: Have $(\sigma(\mathsf{b}),\omega) \in \{(\sigma(\mathsf{c}),\nu)\}[\![\sigma(\alpha^d)]\!]$ iff $(\sigma(\mathsf{b}),\omega) \in \{(\sigma(\mathsf{c}),\nu)\}\langle\!\langle\sigma(\alpha)\rangle\!\rangle$ iff $(\mathsf{b},\sigma^*_\nu(\omega)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}\langle\!\langle\alpha\rangle\!\rangle$ iff $(\mathsf{b},\sigma^*_\nu(\omega)) \in \{(\mathsf{c},\sigma^*_\nu(\nu))\}[\![\alpha^d]\!]$.

Formula cases:

Case $f > g$: Have $(\sigma(\epsilon),\omega) \in [\![\sigma(f > g)]\!]$ iff $(\epsilon,\omega) \in [\![\sigma(f > g)]\!]$ iff $[\![\sigma(f)]\!]\omega > [\![\sigma(g)]\!]\omega$ iff $[\![f]\!]\sigma^*_\omega(\omega) > [\![g]\!]\sigma^*_\omega(\omega)$ iff $(\epsilon,\sigma^*_\omega(\omega)) \in [\![f > g]\!]$. The cases for $\leq, <, =, \neq, \geq$ are symmetric.

Case $\langle\alpha\rangle\phi$: Have

$$
\begin{array}{lll}
& (\sigma(\mathsf{b}),\omega) \in [\![\sigma(\langle\alpha\rangle\phi)]\!] & \\
\text{iff } & \{(\sigma(\mathsf{b}),\omega)\}\langle\!\langle\sigma(\alpha)\rangle\!\rangle \subseteq \{(\sigma(\mathsf{b}),\omega) \mid (\sigma(\mathsf{b}),\omega) \in [\![\sigma(\phi)]\!] \cup \{\top\}\} & \\
\text{iff } & \{(\sigma(\mathsf{b}),\omega)\}\langle\!\langle\sigma(\alpha)\rangle\!\rangle \subseteq \{(\sigma(\mathsf{b}),\omega) \mid (\mathsf{b},\sigma^*_\omega(\omega)) \in [\![\phi]\!] \cup \{\top\}\} & \text{IH on } \phi \\
\text{iff } & \{(\mathsf{b},\sigma^*_\omega(\omega))\}\langle\!\langle\alpha\rangle\!\rangle \subseteq \{(\mathsf{b},\sigma^*_\omega(\omega)) \mid (\mathsf{b},\sigma^*_\omega(\omega)) \in [\![\phi]\!] \cup \{\top\}\} & \text{IH on } \alpha \\
\text{iff } & \{(\mathsf{b},\sigma^*_\omega(\omega))\}\langle\!\langle\alpha\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\} & \\
\text{iff } & (\mathsf{b},\sigma^*_\omega(\omega)) \in [\![\langle\alpha\rangle\phi]\!] &
\end{array}
$$

Case $[\alpha]\phi$: Have

$$
\begin{array}{lll}
& (\sigma(\mathsf{b}),\omega) \in [\![\sigma([\alpha]\phi)]\!] & \\
\text{iff } & \{(\sigma(\mathsf{b}),\omega)\}[\![\sigma(\alpha)]\!] \subseteq \{(\sigma(\mathsf{b}),\omega) \mid (\sigma(\mathsf{b}),\omega) \in [\![\sigma(\phi)]\!] \cup \{\top\}\} & \\
\text{iff } & \{(\sigma(\mathsf{b}),\omega)\}[\![\sigma(\alpha)]\!] \subseteq \{(\sigma(\mathsf{b}),\omega) \mid (\mathsf{b},\sigma^*_\omega(\omega)) \in [\![\phi]\!] \cup \{\top\}\} & \text{IH on } \phi \\
\text{iff } & \{(\mathsf{b},\sigma^*_\omega(\omega))\}[\![\alpha]\!] \subseteq \{(\mathsf{b},\sigma^*_\omega(\omega)) \mid (\mathsf{b},\sigma^*_\omega(\omega)) \in [\![\phi]\!] \cup \{\top\}\} & \text{IH on } \alpha \\
\text{iff } & \{(\mathsf{b},\sigma^*_\omega(\omega))\}[\![\alpha]\!] \subseteq [\![\phi]\!] \cup \{\top\} & \\
\text{iff } & (\mathsf{b},\sigma^*_\omega(\omega)) \in [\![[\alpha]\phi]\!] &
\end{array}
$$

□

Theorem 4.15 (Soundness of proof calculus)**.** *If the proof judgement* $\Gamma \vdash M : \phi$ *holds then the* CGL *natural deduction sequent* $(\Gamma \vdash \phi)$ *is valid. As a special case for empty context* $\cdot$*, if* $\cdot \vdash M : \phi$*, then* ϕ *is valid.*

Proof. By structural induction on the proof term M. In cases with one premise, we write $\mathcal{D}$ for the sole subderivation; in cases with multiple premises, we write $\mathcal{D}_1, \mathcal{D}_2, \ldots$ for the subderivations from left to right.

Case $\dfrac{\Gamma \vdash M : \langle ?\phi \rangle \psi}{\Gamma \vdash \langle \pi_L M \rangle : \phi}$

By $\mathcal{D}$, we assume that $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$ and that $(\mathsf{d}\,\mathsf{c}, \omega) \in [\![\langle ?\phi \rangle \psi]\!]$. Then $(\mathsf{d}\,\mathsf{c}, \omega) \in [\![\langle ?\phi \rangle \psi]\!]$ iff $\{(\mathsf{d}\,\mathsf{c}, \omega)\}\langle\!\langle ?\phi \rangle\!\rangle \subseteq [\![\psi]\!] \cup \{\top\}$ iff $(\pi_0 \mathsf{d}\,\mathsf{c}, \omega) \in [\![\phi]\!]$ and $(\pi_1 \mathsf{d}\,\mathsf{c}, \omega) \in [\![\psi]\!]$. Now define $\mathsf{b}(\mathsf{c}) = \pi_0 \mathsf{d}\,\mathsf{c}$ and observe $(\mathsf{b}, \omega) \in [\![\phi]\!]$ by construction, then since this held for all $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$ and d, we have that $\Gamma \vdash \phi$ is valid.

Case $\dfrac{\Gamma \vdash M : \langle ?\phi \rangle \psi}{\Gamma \vdash \langle \pi_R M \rangle : \psi}$

By $\mathcal{D}$, we assume that $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$ and that $(\mathsf{d}\,\mathsf{c}, \omega) \in [\![\langle ?\phi \rangle \psi]\!]$. Then $(\mathsf{d}\,\mathsf{c}, \omega) \in [\![\langle ?\phi \rangle \psi]\!]$ iff $\{(\mathsf{d}\,\mathsf{c}, \omega)\}\langle\!\langle ?\phi \rangle\!\rangle \subseteq [\![\psi]\!] \cup \{\top\}$ iff $(\pi_0 \mathsf{d}\,\mathsf{c}, \omega) \in [\![\phi]\!]$ and $(\pi_1 \mathsf{d}\,\mathsf{c}, \omega) \in [\![\psi]\!]$. Now define $\mathsf{b}(\mathsf{c}) = \pi_1 \mathsf{d}\,\mathsf{c}$ and observe $(\mathsf{b}, \omega) \in [\![\psi]\!]$ by construction, then since this held for all $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$ and d, we have that $\Gamma \vdash \psi$ is valid.

Case $\dfrac{\Gamma \vdash M : [\alpha \cup \beta]\phi}{\Gamma \vdash [\pi_L M] : [\alpha]\phi}$

By $\mathcal{D}$, assume $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$ and $(\mathsf{d}\,\mathsf{c}, \omega) \in [\![[\alpha \cup \beta]\phi]\!]$. Observe $(\mathsf{d}\,\mathsf{c}, \omega) \in [\![[\alpha \cup \beta]\phi]\!]$ iff $\{(\mathsf{d}\,\mathsf{c}, \omega)\}[\![\alpha \cup \beta]\!] \subseteq [\![\phi]\!] \cup \{\top\}$ iff $\{(\mathsf{d}\,\mathsf{c}, \omega)\}_{[0]}[\![\alpha]\!] \cup \{(\mathsf{d}\,\mathsf{c}, \omega)\}_{[1]}[\![\beta]\!] \subseteq [\![\phi]\!] \cup \{\top\}$, which implies $\{(\mathsf{d}\,\mathsf{c}, \omega)\}_{[0]}[\![\alpha]\!] \subseteq [\![\phi]\!] \cup \{\top\}$ so $(\pi_0(\mathsf{d}\,\mathsf{c}), \omega) \in [\![[\alpha]\phi]\!]$. Now define $\mathsf{b}(\mathsf{c}) = \pi_0 \mathsf{d}\,\mathsf{c}$ and observe $(\mathsf{b}, \omega) \in [\![[\alpha]\phi]\!]$, i.e., $\Gamma \vdash [\alpha]\phi$ is valid.

Case $\dfrac{\Gamma \vdash M : [\alpha \cup \beta]\phi}{\Gamma \vdash [\pi_R M] : [\beta]\phi}$

By $\mathcal{D}$, assume $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$ and $(\mathsf{d}\,\mathsf{c}, \omega) \in [\![[\alpha \cup \beta]\phi]\!]$. Observe $(\mathsf{d}\,\mathsf{c}, \omega) \in [\![[\alpha \cup \beta]\phi]\!]$ iff $\{(\mathsf{d}\,\mathsf{c}, \omega)\}[\![\alpha \cup \beta]\!] \subseteq [\![\phi]\!] \cup \{\top\}$ iff $\{(\mathsf{d}\,\mathsf{c}, \omega)\}_{[0]}[\![\alpha]\!] \cup \{(\mathsf{d}\,\mathsf{c}, \omega)\}_{[1]}[\![\beta]\!] \subseteq [\![\phi]\!] \cup \{\top\}$ which implies $\{(\mathsf{d}\,\mathsf{c}, \omega)\}_{[1]}[\![\beta]\!] \subseteq [\![\phi]\!] \cup \{\top\}$ so $(\pi_1(\mathsf{d}\,\mathsf{c}), \omega) \in [\![[\beta]\phi]\!]$. Now define $\mathsf{b}(\mathsf{c}) = \pi_1 \mathsf{d}\,\mathsf{c}$ and observe $(\mathsf{b}, \omega) \in [\![[\beta]\phi]\!]$, i.e., $\Gamma \vdash [\beta]\phi$ is valid.

Case $\dfrac{\Gamma \vdash M : \phi \quad \Gamma \vdash N : \psi}{\Gamma \vdash \langle M, N \rangle : \langle ?\phi \rangle \psi}$

Assume $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$ and $(\mathsf{d}_1\,\mathsf{c}, \omega) \in [\![\phi]\!]$ and $(\mathsf{d}_2\,\mathsf{c}, \omega) \in [\![\psi]\!]$ by $\mathcal{D}_1$ and $\mathcal{D}_2$. Thus, we have $\{((\mathsf{d}_1\,\mathsf{c}, \mathsf{d}_2\,\mathsf{c}), \omega)\}\langle\!\langle ?\phi \rangle\!\rangle \subseteq [\![\psi]\!] \cup \{\top\}$, thus $((\mathsf{d}_1\,\mathsf{c}, \mathsf{d}_2\,\mathsf{c}), \omega) \in [\![\langle ?\phi \rangle \psi]\!]$ thus, letting $\mathsf{b}(\mathsf{c}) = (\mathsf{d}_1\,\mathsf{c}, \mathsf{d}_2\,\mathsf{c})$, we have $\Gamma \vdash \langle ?\phi \rangle \psi$ valid.

Case $\dfrac{\Gamma \vdash M : [\alpha]\phi \quad \Gamma \vdash N : [\beta]\phi}{\Gamma \vdash [M, N] : [\alpha \cup \beta]\phi}$

Assume $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$ and $(\mathsf{d}_1\,\mathsf{c}, \omega) \in [\![[\alpha]\phi]\!]$ and $(\mathsf{d}_2\,\mathsf{c}, \omega) \in [\![[\beta]\phi]\!]$ by $\mathcal{D}_1$ and $\mathcal{D}_2$, thus $\{(\mathsf{d}_1\,\mathsf{c}, \omega)\}[\![\alpha]\!] \cup \{(\mathsf{d}_2\,\mathsf{c}, \omega)\}[\![\beta]\!] \subseteq [\![\phi]\!] \cup \{\top\}$. Then by letting $\mathsf{b}(\mathsf{c}) = (\mathsf{d}_1\,\mathsf{c}, \mathsf{d}_2\,\mathsf{c})$ we have that $\{(\mathsf{b}, \omega)\}[\![\alpha \cup \beta]\!] = \{(\mathsf{b}, \omega)\}_{[0]}[\![\alpha]\!] \cup \{(\mathsf{b}, \omega)\}_{[1]}[\![\beta]\!] = (\mathsf{d}_1\,\mathsf{c}, \omega)[\![\alpha]\!] \cup (\mathsf{d}_2\,\mathsf{c}, \omega)[\![\beta]\!] \subseteq [\![\phi]\!] \cup \{\top\}$. It then follows that $(\mathsf{b}, \omega) \in [\![[\alpha \cup \beta]\phi]\!]$ and that $\Gamma \vdash [\alpha \cup \beta]\phi$ valid.

Case $\dfrac{\Gamma\frac{y}{x}, p : (x = f\frac{y}{x}) \vdash M : \phi}{\Gamma \vdash \langle f\frac{y}{x} :* p.\ M \rangle : \langle x := * \rangle \phi}$

Assume side condition that y is fresh in Γ, f, and $\langle x := * \rangle \phi$ and that p is fresh in Γ. Assume (G) $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$. Because y is fresh in Γ, assume (RzFV) without loss of generality that $y \notin \mathrm{FV}(\mathsf{c})$. From (G) by Lemma A.5 and Lemma A.4 have (1) $(\mathsf{c}\frac{y}{x}, \omega\frac{y}{x}) \in [\![\bigwedge \Gamma\frac{y}{x}]\!]$. Let $\nu = \omega\frac{y}{x}[x \mapsto [\![f]\!]\omega]$ and $\mathsf{rz} = (\mathsf{c}\frac{y}{x}, \epsilon)$. Note (2) $(\pi_0\mathsf{rz}, \nu) \in [\![\bigwedge \Gamma\frac{y}{x}]\!]$ by (1) and because y is fresh by the side condition, so Lemma 4.11 applies on states $\omega\frac{y}{x}$ and ν which agree on every variable except x, thus they agree on all free variables of $\Gamma\frac{y}{x}$ and, by (RzFV), free variables of $(\pi_0\mathsf{rz}) = (\mathsf{c}\frac{y}{x})$. Note (3) $(\pi_1\mathsf{rz}, \nu) \in [\![x = f\frac{y}{x}]\!]$ follows from $[\![x]\!]\nu = [\![f\frac{y}{x}]\!]\nu$ which holds because $[\![x]\!]\nu = [\![f]\!]\omega$ by construction and $[\![f]\!]\omega = [\![f\frac{y}{x}]\!]\omega\frac{y}{x}$ holds by renaming (Lemma A.5 and Lemma A.4), then applying Lemma 4.11 on term $f\frac{y}{x}$ yields $[\![f\frac{y}{x}]\!]\omega\frac{y}{x} = [\![f\frac{y}{x}]\!]\nu$ since $\omega\frac{y}{x} = \nu$ except on x which is fresh in $f\frac{y}{x}$ by the assumption that variable y is fresh.

From (2) and (3) we have (4) $(\mathsf{rz}, \nu) \in [\![\bigwedge \Gamma\frac{y}{x}, x = f\frac{y}{x}]\!]$ so that the IH on $\mathcal{D}$ can finally be applied, yielding (5) $(\mathsf{d}\,\mathsf{rz}, \nu) \in [\![\phi]\!]$ where we let d be the realizer from the IH on $\mathcal{D}$.

Let $\mathsf{b}(\mathsf{c}) = (\Lambda y : \mathbb{Q}.\ (f, \mathsf{d}\,\mathsf{c}))\,x$. Want to show $(\mathsf{b}\ \mathsf{c}, \omega) \in [\![\langle x := * \rangle \phi]\!]$, so it suffices to show $\{(\mathsf{b}\ \mathsf{c}, \omega)\}\langle\!\langle x := * \rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\}$. Since $\{(\mathsf{b}\ \mathsf{c}, \omega)\}\langle\!\langle x := * \rangle\!\rangle = \{(\pi_1(\mathsf{b}\ \mathsf{c}), \omega[x \mapsto [\![\pi_0(\mathsf{b}\ \mathsf{c})]\!]\omega])\}$, it suffices to show $(\pi_1(\mathsf{b}\ \mathsf{c}), \omega[x \mapsto [\![\pi_0(\mathsf{b}\ \mathsf{c})]\!]\omega]) \in [\![\phi]\!] \cup \{\top\}$, of which we show $(\pi_1(\mathsf{b}\ \mathsf{c}), \omega[x \mapsto [\![\pi_0(\mathsf{b}\ \mathsf{c})]\!]\omega]) \in [\![\phi]\!]$ specifically. By definition of b have $[\![\pi_0(\mathsf{b}\ \mathsf{c})]\!]\omega = [\![f]\!](\omega[y \mapsto \omega(x)])$ and $\pi_1(\mathsf{b}\ \mathsf{c}) = (\mathsf{d}\,\mathsf{c})_y^{\omega(x)}$, so it suffices to show (6)

$$((\mathsf{d}\,\mathsf{c})_y^{\omega(x)}, \omega[x \mapsto [\![f]\!](\omega[y \mapsto \omega(x)])]) \in [\![\phi]\!]$$

We derive (6) from (5).

Note $\nu = \omega[x \mapsto [\![f]\!]\omega][y \mapsto \omega(x)]$ by definition of renaming. By Lemma A.9 we can apply formula substitution on (5), so we have (6a) $((\mathsf{d}\,\mathsf{rz})_y^{\omega(x)}, \omega[x \mapsto [\![f]\!]\omega]) \in [\![\phi]\!]$. Then observe (6b) $[\![f]\!](\omega[y \mapsto \omega(x)]) = [\![f]\!]\omega$ by Lemma 4.11 on term f because variable y is fresh in f. Fact (6) is immediate by replacing (6b) in (6a).

Case $\dfrac{\Gamma \vdash M : \langle\alpha\rangle\phi}{\Gamma \vdash \langle \ell \cdot M\rangle : \langle \alpha \cup \beta\rangle\phi}$

By $\mathcal{D}$, assume $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$ have $(\mathsf{d}\,\mathsf{c}, \omega) \in [\![\langle\alpha\rangle\phi]\!]$ so $\{(\mathsf{d}\,\mathsf{c}, \omega)\}\langle\!\langle\alpha\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\}$. Now let $f = 0$ and $\mathsf{b}(\mathsf{c}) = (f, \mathsf{d}\,\mathsf{c})$, then $\{(\mathsf{b}, \omega)\}\langle\!\langle\alpha \cup \beta\rangle\!\rangle = \{(\mathsf{d}\,\mathsf{c}, \omega)\}\langle\!\langle\alpha\rangle\!\rangle$ and by transitivity $\{(\mathsf{b}, \omega)\}\langle\!\langle\alpha \cup \beta\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\}$, that is $(\mathsf{b}, \omega) \in [\![\langle\alpha \cup \beta\rangle\phi]\!]$ as desired, thus the sequent $\Gamma \vdash \langle\alpha \cup \beta\rangle\phi$ is valid.

Case $\dfrac{\Gamma \vdash M : \langle\beta\rangle\phi}{\Gamma \vdash \langle r \cdot M\rangle : \langle \alpha \cup \beta\rangle\phi}$

By $\mathcal{D}$, assume $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$ have $(\mathsf{d}\,\mathsf{c}, \omega) \in [\![\langle\beta\rangle\phi]\!]$ so $\{(\mathsf{d}\,\mathsf{c}, \omega)\}\langle\!\langle\beta\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\}$. Now let $f = 1$ and $\mathsf{b}(\mathsf{c}) = (f, \mathsf{d}\,\mathsf{c})$, then $\{(\mathsf{b}, \omega)\}\langle\!\langle\alpha \cup \beta\rangle\!\rangle = \{(\mathsf{d}\,\mathsf{c}, \omega)\}\langle\!\langle\beta\rangle\!\rangle$ and by transitivity $\{(\mathsf{b}, \omega)\}\langle\!\langle\alpha \cup \beta\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\}$, that is $(\mathsf{b}, \omega) \in [\![\langle\alpha \cup \beta\rangle\phi]\!]$ as desired, thus the sequent $\Gamma \vdash \langle\alpha \cup \beta\rangle\phi$ is valid.

Case $\dfrac{\Gamma \vdash A : \langle\alpha \cup \beta\rangle\phi \quad \Gamma, \ell : \langle\alpha\rangle\phi \vdash B : \psi \quad \Gamma, r : \langle\beta\rangle\phi \vdash C : \psi}{\Gamma \vdash \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : \psi}$

By $\mathcal{D}_1$ assume $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$ and have $(\mathsf{d}\,\mathsf{c}, \omega) \in [\![\langle\alpha \cup \beta\rangle\phi]\!]$ so $\{(\mathsf{d}\,\mathsf{c}, \omega)\}\langle\!\langle\alpha \cup \beta\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\}$.

In the first case, $[\![\pi_0(\mathsf{d}\,\mathsf{c})]\!]\omega = 0$, so $\{(\pi_1(\mathsf{d}\,\mathsf{c}), \omega)\}\langle\!\langle\alpha\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\}$. Now let $\mathsf{c}_1 = (\mathsf{c}, \pi_1(\mathsf{d}\,\mathsf{c}))$, then $\mathcal{D}_2$ is applicable because $(\mathsf{c}_1, \omega) \in [\![\bigwedge(\Gamma, \ell : \langle\alpha\rangle\phi)]\!]$, so we have some d_1

such that $(\mathsf{d}_1\,\mathsf{c}_1, \omega) \in [\![\psi]\!]$.

In the second case, $[\![\pi_0(\mathsf{d}\,\mathsf{c})]\!]\omega = 1$, so $\{(\pi_1(\mathsf{d}\,\mathsf{c}), \omega)\}\langle\!\langle\beta\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\}$. Now let $\mathsf{c}_2 = (\mathsf{c}, \pi_1(\mathsf{d}\,\mathsf{c}))$, then $\mathcal{D}_3$ is applicable because $(\mathsf{c}_2, \omega) \in [\![\bigwedge(\Gamma, r : \langle\beta\rangle\phi)]\!]$, so we have some d_2 such that $(\mathsf{d}_2\,\mathsf{c}_2, \omega) \in [\![\psi]\!]$.

Lastly, define $\mathsf{b}(\mathsf{c}) = \texttt{if}\ (f = 0)\ \{\mathsf{d}_1\,\mathsf{c}_1\}\ \texttt{else}\ \{\mathsf{d}_2\,\mathsf{c}_2\}$ and observe $(\mathsf{b}\,\mathsf{c}, \omega) \in [\![\psi]\!]$ as desired by the previous two cases.

Case $\dfrac{\Gamma \vdash M : \psi \quad p : \psi \vdash N : [\alpha]\psi \quad p : \psi \vdash O : \phi}{\Gamma \vdash (M \text{ rep } p : \psi.\ N \text{ in } O) : [\alpha^*]\phi}$

Assume some d_1 such that (1) for all $(\mathsf{c}_1, \omega) \in [\![\bigwedge \Gamma]\!]$ have $(\mathsf{d}_1\,\mathsf{c}_1, \omega) \in [\![\psi]\!]$ by $\mathcal{D}_1$, assume some d_2 such that (2) for all $(\mathsf{c}_2, \omega) \in [\![\bigwedge p : \psi]\!]$ have $(\mathsf{d}_2\,\mathsf{c}_2, \omega) \in [\![[\alpha]\psi]\!]$ by $\mathcal{D}_2$, and assume some d_3 such that (3) for all $(\mathsf{c}_3, \omega) \in [\![\bigwedge p : \psi]\!]$ have $(\mathsf{d}_3\,\mathsf{c}_3, \omega) \in [\![\phi]\!]$. Now fix ω and assume c_1 such that $(\mathsf{c}_1, \omega) \in [\![\bigwedge \Gamma]\!]$.

Facts (2) and (3) imply (2a) $(\mathsf{d}_2, \omega) \in [\![\bigwedge \Gamma \to \psi]\!]$ and (3a) $(\mathsf{d}_3, \omega) \in [\![\psi \to \phi]\!]$ by the semantics of implication.

By applying Lemma A.3 to (1), (2a), and (3a) have $(\mathsf{gen}(\mathsf{d}_1\,\mathsf{c}_1,\ \mathsf{d}_2,\ \mathsf{d}_3), \omega) \in [\![[\alpha^*]\phi]\!]$ which completes the case by letting $\mathsf{b}(\mathsf{c}_1) = \mathsf{gen}(\mathsf{d}_1\,\mathsf{c}_1,\ \mathsf{d}_2,\ \mathsf{d}_3)$.

Case $\dfrac{\Gamma \vdash M : [?\phi]\psi \quad \Gamma \vdash N : \phi}{\Gamma \vdash (M\ N) : \psi}$

Assume (1) some d_1 such that for all $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$ have $(\mathsf{d}_1\,\mathsf{c}, \omega) \in [\![[?\phi]\psi]\!]$ by $\mathcal{D}_1$. Assume (2) some d_2 such that for all $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$ have $(\mathsf{d}_2\,\mathsf{c}, \omega) \in [\![\phi]\!]$ by $\mathcal{D}_2$. Assume (3) some $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$. By (1), then $(\mathsf{d}_1\,\mathsf{c}, \omega)[\![?\phi]\!] \subseteq [\![\psi]\!] \cup \{\top\}$. That is for all rz s.t. $(\mathsf{rz}, \omega) \in [\![\phi]\!]$ then $(\mathsf{d}_1\,\mathsf{c}\,\mathsf{rz}, \omega) \in [\![\psi]\!] \cup \{\top\}$. By (2) then let $\mathsf{rz} = \mathsf{d}_2\,\mathsf{c}$ and have $((\mathsf{d}_1\,\mathsf{c})\,(\mathsf{d}_2\,\mathsf{c}), \omega) \in [\![\psi]\!] \cup \{\top\}$. Moreover, $((\mathsf{d}_1\,\mathsf{c})\,(\mathsf{d}_2\,\mathsf{c}), \omega) \in [\![\psi]\!] \cup \{\top\}$ by the semantics of $?\phi$, which could only be $\top$ when there exists *no* d_2 that satisfies the test. Then letting $\mathsf{b}(\mathsf{c}) = (\mathsf{d}_1\,\mathsf{c})\,(\mathsf{d}_2\,\mathsf{c})$, we have $(\mathsf{b}\,\mathsf{c}, \omega) \in [\![\psi]\!]$ as desired.

Case $\dfrac{\Gamma, p : \phi \vdash M : \psi}{\Gamma \vdash (\lambda p : \phi.\ M) : [?\phi]\psi}$

Assume (1) some d such that for all $(\mathsf{c}_2, \omega) \in [\![\bigwedge(\Gamma, p : \phi)]\!]$ have $(\mathsf{d}\,\mathsf{c}_2, \omega) \in [\![\psi]\!]$ by $\mathcal{D}$. Assume (2) some $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$. Let $\mathsf{b}(\mathsf{c})(\mathsf{c}_1) = \mathsf{d}\,(\mathsf{c}, \mathsf{c}_1)$. Suffice to show that assuming (3) $(\mathsf{c}_1, \omega) \in [\![\phi]\!]$ then $\mathsf{b}\,\mathsf{c}\,\mathsf{c}_1 \in [\![\psi]\!]$. This holds by expanding the definition of b, then applying (1), whose assumptions hold by (2) and (3).

Case $\dfrac{\Gamma\frac{y}{x} \vdash M : \phi}{\Gamma \vdash (\lambda x : \mathbb{Q}.\ M) : [x := *]\phi}$

Assume side condition (SC) that y is fresh in Γ and $[x := *]\phi$. Assume (1) some d such that for $(\mathsf{c}_1, \omega) \in [\![\bigwedge \Gamma\frac{y}{x}]\!]$ have $(\mathsf{d}\,\mathsf{c}_1, \omega) \in [\![\phi]\!]$ by $\mathcal{D}$. Assume (2) some $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$. Because y is fresh in Γ by (SC), assume (RzFVC) without loss of generality that $y \notin \mathrm{FV}(\mathsf{c})$. Because y is fresh in ϕ and x fresh in $\Gamma\frac{y}{x}$ by (SC), assume (RzFVD) without loss of generality that $y \notin \mathrm{FV}(\mathsf{d}\,\mathsf{c}_1)$ when $x \notin \mathrm{FV}(\mathsf{c}_1)$. Let $\mathsf{b}(\mathsf{c})(q) = \mathsf{d}\,\mathsf{c}\frac{y}{x}$. Note by renaming on (RzFVC) that $x \notin \mathrm{FV}(\mathsf{c}\frac{y}{x})$ so that by (RzFVD) have (RzFVB) $y \notin \mathrm{FV}(\mathsf{d}\,\mathsf{c}_1)$, i.e., $y \notin \mathrm{FV}(\mathsf{b}(\mathsf{c})(q))$ for all $q \in \mathbb{Q}$. To show the case, suffices to show $[\![\phi]\!] \supseteq \{(\mathsf{b}(\mathsf{c}), \omega)\}[\![x := *]\!] = \{(\mathsf{b}(\mathsf{c})(v), \omega[x \mapsto v]) \mid v \in \mathbb{Q}\}$. Then by Lemma A.4 and Lemma A.5 have $(\mathsf{c}\frac{y}{x}, \omega\frac{y}{x}) \in [\![\bigwedge \Gamma\frac{y}{x}]\!]$. Let $v \in \mathbb{Q}$ be arbitrary. Then by renaming on (RzFVC) and (SC) have x fresh in $\Gamma\frac{y}{x}$ and $\mathsf{c}\frac{y}{x}$ so that Lemma 4.11 applies to give $(\mathsf{c}\frac{y}{x}, \omega\frac{y}{x}[x \mapsto v]) \in [\![\bigwedge \Gamma\frac{y}{x}]\!]$. Then by

(1) have $(\mathsf{d}\,\mathsf{c}\frac{y}{x}, \omega\frac{y}{x}[x \mapsto v]) \in [\![\phi]\!]$. Note that $\mathsf{b}(\mathsf{c})(v) = \mathsf{d}\,\mathsf{c}\frac{y}{x}$ by construction so that $(\mathsf{b}(\mathsf{c})(v), \omega\frac{y}{x}[x \mapsto v]) \in [\![\phi]\!]$.

Next we wish to show we can apply Lemma 4.11 again on formula ϕ with realizer $\mathsf{b}(\mathsf{c})(v)$ and states $\omega\frac{y}{x}[x \mapsto v]$ and $\omega[x \mapsto v]$ which differ only on y. By (SC) and (RzFVB) we have $y \notin \mathrm{FV}(\phi) \cup \mathrm{FV}(\mathsf{b}(\mathsf{c})(v))$. Thus, the precondition of Lemma 4.11 is satisfied, yielding: $(\mathsf{b}(\mathsf{c})(v), \omega[x \mapsto v]) \in [\![\phi]\!]$. The argument was generic in $v \in \mathbb{Q}$ so that $(\mathsf{b}(\mathsf{c})(v), \omega[x \mapsto v]) \in [\![\phi]\!]$ for all $v \in \mathbb{Q}$, which shows the case as we have already argued the case reduces to showing $\{(\mathsf{b}(\mathsf{c})(v), \omega[x \mapsto v]) \mid v \in \mathbb{Q}\} \subseteq [\![\phi]\!]$.

Case $\dfrac{\Gamma \vdash M : [\alpha][\beta]\phi}{\Gamma \vdash [\iota\ M] : [\alpha;\beta]\phi}$

Recall below that rz is a realizer variable name. Assume (1) some d such that for all $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$ have $(\mathsf{d}\,\mathsf{c}, \omega) \in [\![[\alpha][\beta]\phi]\!]$ by $\mathcal{D}$. Assume (2) some $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$. Then by (1) we have that

$$
\begin{aligned}
&(\mathsf{d}\,\mathsf{c}, \omega) \in [\![[\alpha][\beta]\phi]\!] \\
\text{iff } &\{(\mathsf{d}\,\mathsf{c}, \omega)\}[\![\alpha]\!] \subseteq [\![[\beta]\phi]\!] \cup \{\top\} \\
\text{iff } &\{(\mathsf{d}\,\mathsf{c}, \omega)\}[\![\alpha]\!] \subseteq \{(\mathsf{rz}, \nu) \mid \{(\mathsf{rz}, \nu)\}[\![\beta]\!] \subseteq [\![\phi]\!] \cup \{\top\}\} \cup \{\top\} \\
\text{iff } &\Big(\bigcup_{(\mathsf{rz},\nu) \in \{(\mathsf{d}\,\mathsf{c},\omega)\}[\![\alpha]\!]} \{(\mathsf{rz}, \nu)\}[\![\beta]\!]\Big) \subseteq [\![\phi]\!] \cup \{\top\} \\
\text{iff } &(\{(\mathsf{d}\,\mathsf{c}, \omega)\}[\![\alpha]\!])[\![\beta]\!] \subseteq [\![\phi]\!] \cup \{\top\} && \text{by Lemma A.1} \\
\text{iff } &\{(\mathsf{d}\,\mathsf{c}, \omega)\}[\![\alpha;\beta]\!] \subseteq [\![\phi]\!] \cup \{\top\} \\
\text{iff } &(\mathsf{d}\,\mathsf{c}, \omega) \in [\![[\alpha;\beta]\phi]\!]
\end{aligned}
$$

as desired. Recall that Lemma A.1 is a corollary of Scott-continuity (Lemma 4.7); we apply it on β to show that the final region $(\{(\mathsf{d}\,\mathsf{c}, \omega)\}[\![\alpha]\!])[\![\beta]\!]$ which arises from full initial region $(\{(\mathsf{d}\,\mathsf{c}, \omega)\}[\![\alpha]\!])$ is equal to the union of final regions resulting from each singleton initial region $\{(\mathsf{d}\,\mathsf{c}, \omega)\}$ drawn from the full initial region $\{(\mathsf{d}\,\mathsf{c}, \omega)\}[\![\alpha]\!]$.

Case $\dfrac{\Gamma \vdash M : \langle\alpha\rangle\langle\beta\rangle\phi}{\Gamma \vdash \langle\iota\ M\rangle : \langle\alpha;\beta\rangle\phi}$

Assume (1) some d such that for all $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$ have $(\mathsf{d}\,\mathsf{c}, \omega) \in [\![\langle\alpha\rangle\langle\beta\rangle\phi]\!]$ by $\mathcal{D}$. Assume (2) some $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$. Then by (1) we have that

$$
\begin{aligned}
&(\mathsf{d}\,\mathsf{c}, \omega) \in [\![\langle\alpha\rangle\langle\beta\rangle\phi]\!] \\
\text{iff } &\{(\mathsf{d}\,\mathsf{c}, \omega)\}\langle\!\langle\alpha\rangle\!\rangle \subseteq ([\![\langle\beta\rangle\phi]\!] \cup \{\top\}) \\
\text{iff } &\{(\mathsf{d}\,\mathsf{c}, \omega)\}\langle\!\langle\alpha\rangle\!\rangle \subseteq \{(\mathsf{rz}, \nu) \mid \{(\mathsf{rz}, \nu)\}\langle\!\langle\beta\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\}\} \cup \{\top\} \\
\text{iff } &\Big(\bigcup_{(\mathsf{rz},\nu) \in \{(\mathsf{d}\,\mathsf{c},\omega)\}\langle\!\langle\alpha\rangle\!\rangle} \{(\mathsf{rz}, \nu)\}\langle\!\langle\beta\rangle\!\rangle\Big) \subseteq [\![\phi]\!] \cup \{\top\} \\
\text{iff } &\{(\mathsf{d}\,\mathsf{c}, \omega)\}\langle\!\langle\alpha\rangle\!\rangle\langle\!\langle\beta\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\} && \text{by Lemma A.1} \\
\text{iff } &\{(\mathsf{d}\,\mathsf{c}, \omega)\}\langle\!\langle\alpha;\beta\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\} \\
\text{iff } &(\mathsf{d}\,\mathsf{c}, \omega) \in [\![\langle\alpha;\beta\rangle\phi]\!]
\end{aligned}
$$

as desired. Recall that Lemma A.1 is a corollary of Scott-continuity (Lemma 4.7) on β to show that the final region $(\{(\mathsf{d}\,\mathsf{c}, \omega)\}\langle\!\langle\alpha\rangle\!\rangle)\langle\!\langle\beta\rangle\!\rangle$ which arises from full initial region

$(\{(\mathsf{d}\,\mathsf{c},\omega)\}\langle\!\langle\alpha\rangle\!\rangle)$ is equal to the union of final regions resulting from each singleton initial region $\{(\mathsf{d}\,\mathsf{c},\omega)\}$ drawn from the full initial region $\{(\mathsf{d}\,\mathsf{c},\omega)\}\langle\!\langle\alpha\rangle\!\rangle$.

$$\textbf{Case}\ \frac{\Gamma \vdash M : [\alpha]\phi}{\Gamma \vdash \langle\text{yield } M\rangle : \langle\alpha^d\rangle\phi}$$

Assume (1) some d such that for all $(\mathsf{c},\omega) \in [\![\bigwedge \Gamma]\!]$ have $(\mathsf{d}\,\mathsf{c},\omega) \in [\![[\alpha]\phi]\!]$ by $\mathcal{D}$. Assume (2) some $(\mathsf{c},\omega) \in [\![\bigwedge \Gamma]\!]$, then by (1) have $(\mathsf{d}\,\mathsf{c},\omega) \in [\![[\alpha]\phi]\!]$. Observe $(\mathsf{d}\,\mathsf{c},\omega) \in [\![[\alpha]\phi]\!]$ iff $\{(\mathsf{d}\,\mathsf{c},\omega)\}[\![\alpha]\!] \subseteq [\![\phi]\!] \cup \{\top\}$ iff $\{(\mathsf{d}\,\mathsf{c},\omega)\}\langle\!\langle\alpha^d\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\}$ iff $(\mathsf{d}\,\mathsf{c},\omega) \in [\![\langle\alpha^d\rangle\phi]\!]$ as desired.

$$\textbf{Case}\ \frac{\Gamma \vdash M : \langle\alpha\rangle\phi}{\Gamma \vdash [\text{yield } M] : [\alpha^d]\phi}$$

Assume (1) some d such that for all $(\mathsf{c},\omega) \in [\![\bigwedge \Gamma]\!]$ have $(\mathsf{d}\,\mathsf{c},\omega) \in [\![\langle\alpha\rangle\phi]\!]$ by $\mathcal{D}$. Assume (2) some $(\mathsf{c},\omega) \in [\![\bigwedge \Gamma]\!]$, then by (1) have $(\mathsf{d}\,\mathsf{c},\omega) \in [\![\langle\alpha\rangle\phi]\!]$. Observe $(\mathsf{d}\,\mathsf{c},\omega) \in [\![\langle\alpha\rangle\phi]\!]$ iff $\{(\mathsf{d}\,\mathsf{c},\omega)\}\langle\!\langle\alpha\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\}$ iff $\{(\mathsf{d}\,\mathsf{c},\omega)\}[\![\alpha^d]\!] \subseteq [\![\phi]\!] \cup \{\top\}$ iff $(\mathsf{d}\,\mathsf{c},\omega) \in [\![[\alpha^d]\phi]\!]$ as desired.

$$\textbf{Case}\ \frac{\Gamma \vdash M : \langle\alpha\rangle\phi \quad p : \phi \vdash N : \psi}{\Gamma \vdash M \circ_p N : \langle\alpha\rangle\psi}$$

Assume (1) some d_1 such that for all $(\mathsf{c}_1,\omega) \in [\![\bigwedge \Gamma]\!]$ have $(\mathsf{d}_1\,\mathsf{c}_1,\omega) \in [\![\langle\alpha\rangle\phi]\!]$ by $\mathcal{D}_1$. Assume (2) some d_2 such that for all $(\mathsf{c}_2,\nu) \in [\![\bigwedge p : \phi]\!]$ have $(\mathsf{d}_2\,\mathsf{c}_2,\omega) \in [\![\psi]\!]$ by $\mathcal{D}_2$. Assume (3) some $(\mathsf{c},\omega) \in [\![\bigwedge \Gamma]\!]$. From (3) and (1) we have (1a) $(\mathsf{d}_1\,\mathsf{c},\omega) \in [\![\langle\alpha\rangle\phi]\!]$.

We proceed by applying Lemma A.2 on region $S = \mathcal{S}$, i.e., we prove the monotone implication holds everywhere, not only in the final states of executions of α. To apply the lemma, we first rephrase fact (2). Fact (2) equivalently says (2a) $\{\mathsf{d}_2\} \times \mathcal{S} \subseteq [\![\phi \to \psi]\!]$. By applying Lemma A.2 on (2a) and (1a) we have $(\mathsf{mon}_{\langle\alpha\rangle}\ \mathsf{d}_2\ (\mathsf{d}_1\,\mathsf{c}),\omega) \in [\![\langle\alpha\rangle\psi]\!]$.

That is, by letting $\mathsf{b}(\mathsf{c}) = \mathsf{mon}_{\langle\alpha\rangle}\ \mathsf{d}_2\ (\mathsf{d}_1\,\mathsf{c})$ we have $(\mathsf{b}\,\mathsf{c},\omega) \in [\![\langle\alpha\rangle\psi]\!]$ as desired.

Note that the monotonicity rule is often used for both box and diamond modalities in practical proofs, but the box rule is easily derivable from the diamond rule using the equivalence $[\alpha]\phi \leftrightarrow \langle\alpha^d\rangle\phi$.

$$\textbf{Case}\ \frac{\Gamma \vdash M : \phi}{\Gamma \vdash \langle\text{stop } M\rangle : \langle\alpha^*\rangle\phi}$$

Assume (1) some d such that for $(\mathsf{c},\omega) \in [\![\bigwedge \Gamma]\!]$ have $(\mathsf{d}\,\mathsf{c},\omega) \in [\![\phi]\!]$ by $\mathcal{D}$. Assume (2) some $(\mathsf{c},\omega) \in [\![\bigwedge \Gamma]\!]$. Let $\mathsf{b}(\mathsf{c}) = (0, \mathsf{d}\,\mathsf{c})$. Then by (1) have $(\mathsf{d}\,\mathsf{c},\omega) \in [\![\phi]\!]$ and $\{(\mathsf{b}(\mathsf{c}),\omega)\}_{\langle 0\rangle} = \{(\mathsf{d}\,\mathsf{c},\omega)\}$ so (4) $\{(\mathsf{b}(\mathsf{c}),\omega)\}_{\langle 0\rangle} \subseteq [\![\phi]\!] \subseteq [\![\phi]\!] \cup \{\top\}$. Now consider the semantics of loop α^*: $\{(\mathsf{b}(\mathsf{c}),\omega)\}\langle\!\langle\alpha^*\rangle\!\rangle = \bigcap\{Z_{\langle 0\rangle} \subseteq \mathsf{Poss} \mid \{(\mathsf{b}(\mathsf{c}),\omega)\} \cup (Z_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle) \subseteq Z\} = \{(\mathsf{b}(\mathsf{c}),\omega)\}_{\langle 0\rangle}$ since $\{(\mathsf{b}(\mathsf{c}),\omega)\}_{\langle 1\rangle} = \emptyset$ by construction of b. That is, $Z = \{(\mathsf{b}(\mathsf{c}),\omega)\}$ satisfies the fixed-point equation, and must be the solution of the fixed point because no other singleton or empty set satisfies the fixed-point equation. By transitivity with (4), we have $\{(\mathsf{b}\,\mathsf{c},\omega)\}\langle\!\langle\alpha^*\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\}$ which collapses to $(\mathsf{b}\,\mathsf{c},\omega) \in [\![\langle\alpha^*\rangle\phi]\!]$, proving the case.

$$\textbf{Case}\ \frac{\Gamma \vdash M : \langle\alpha\rangle\langle\alpha^*\rangle\phi}{\Gamma \vdash \langle\text{go } M\rangle : \langle\alpha^*\rangle\phi}$$

Assume (1) some d such that for all $(\mathsf{c},\omega) \in [\![\bigwedge \Gamma]\!]$ have $(\mathsf{d}\,\mathsf{c},\omega) \in [\![\langle\alpha\rangle\langle\alpha^*\rangle\phi]\!]$ by $\mathcal{D}$. Assume (2) some $(\mathsf{c},\omega) \in [\![\bigwedge \Gamma]\!]$.

Unfolding definitions in (1) applied to (2), we have

$$\begin{aligned}
&(\mathsf{d}\,\mathsf{c},\omega) \in [\![\langle\alpha\rangle\langle\alpha^*\rangle\phi]\!] \\
\text{iff } &\{(\mathsf{d}\,\mathsf{c},\omega)\}\langle\!\langle\alpha\rangle\!\rangle \subseteq [\![\langle\alpha^*\rangle\phi]\!] \cup \{\top\} \\
\text{iff } &\{(\mathsf{d}\,\mathsf{c},\omega)\}\langle\!\langle\alpha\rangle\!\rangle \subseteq \{(\mathsf{rz},\nu) \mid \{(\mathsf{rz},\nu)\}\langle\!\langle\alpha^*\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\}\} \cup \{\top\} \\
\text{iff } &\{(\mathsf{d}\,\mathsf{c},\omega)\}\langle\!\langle\alpha\rangle\!\rangle\langle\!\langle\alpha^*\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\}
\end{aligned}$$

by Lemma A.1.

Let $D = (\{(\mathsf{d}\,\mathsf{c},\omega)\}\langle\!\langle\alpha\rangle\!\rangle)\langle\!\langle\alpha^*\rangle\!\rangle$ so that (3) $D \subseteq [\![\phi]\!] \cup \{\top\}$.

We now begin using (3) to prove the conclusion $\langle\alpha^*\rangle\phi$. We will define abbreviation $C(X)$ such that $C(X)_{\langle 0\rangle} = X\langle\!\langle\alpha^*\rangle\!\rangle$ for arbitrary region X, i.e., let $C(X) = \bigcap\{Z \subseteq \mathsf{Poss} \mid X \cup (Z_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle) \subseteq Z\}$. From Lemma 4.8, it follows that (4) $C(X) = \bigcup_{k\in\mathbb{N}} X\langle\!\langle\alpha\rangle\!\rangle^k$. Recall also that the solution of a fixed-point construction satisfies the fixed-point inclusion as an exact equality, so (5) $C(X) = X \cup (C_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle)$.

We choose $\mathsf{b}(\mathsf{c}) = (1, \mathsf{d}\,\mathsf{c})$. To show the case, it suffices to show $C(\{\mathsf{b}\} \times \mathcal{S}\})_{\langle 0\rangle} \subseteq [\![\phi]\!] \cup \{\top\}$, or equivalently it suffices to show $C(\{(\mathsf{b},\omega)\})_{\langle 0\rangle} \subseteq [\![\phi]\!] \cup \{\top\}$ for all ω satisfying (2), by Lemma A.1, as we will do.

Consider arbitrary $(\mathsf{rz},\nu) \in C(\{(\mathsf{b},\omega)\})$, then by definition of C either $(\mathsf{rz},\nu) \in \{(\mathsf{b},\omega)\}$ or $(\mathsf{rz},\nu) \in (C(\{(\mathsf{b},\omega)\})_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle)$. In the first case, $\{(\mathsf{b},\omega)\}_{\langle 0\rangle} = \emptyset$ by construction of b, so $\{(\mathsf{rz},\nu)\}_{\langle 0\rangle} \subseteq [\![\phi]\!] \cup \{\top\}$ trivially.

In the latter case: $(\mathsf{rz},\nu) \in (C(\{(\mathsf{b},\omega)\})_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle)$. Note $(\mathsf{rz},\nu) \in (C(\{(\mathsf{b},\omega)\})_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle)$ iff$_{(4)}$ $(\mathsf{rz},\nu) \in ((\bigcup_{k\in\mathbb{N}} \{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle^k)_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle)$ iff $(\mathsf{rz},\nu) \in \bigcup_{k\in\mathbb{N}}(\{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle^k_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle)$, by applying Lemma A.1, call this fact (6).

For all k, have $\{(\mathsf{b},\omega)\}\langle\!\langle\alpha\rangle\!\rangle^k_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle = (\{(\mathsf{b},\omega)\}_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle)\langle\!\langle\alpha\rangle\!\rangle^k$ by Remark 4.1, thus we have $\{(\mathsf{rz},\nu)\}_{\langle 0\rangle} \in (\bigcup_{k\in\mathbb{N}}(\{(\mathsf{b},\omega)\}_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle)\langle\!\langle\alpha\rangle\!\rangle^k)_{\langle 0\rangle} = (\{(\mathsf{b},\omega)\}_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle)\langle\!\langle\alpha^*\rangle\!\rangle$.

Recall $\mathsf{b}(\mathsf{c}) = (1, \mathsf{d}\,\mathsf{c})$ so that $(\{(\mathsf{b},\omega)\}_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle)\langle\!\langle\alpha^*\rangle\!\rangle = (\{(\mathsf{d}\,\mathsf{c},\omega)\}\langle\!\langle\alpha\rangle\!\rangle)\langle\!\langle\alpha^*\rangle\!\rangle = D$ by definition of D, and by fact (3) and transitivity have $\{(\mathsf{rz},\nu)\}_{\langle 0\rangle} \in D \subseteq [\![\phi]\!] \cup \{\top\}$ as desired, completing case 2, which completes the case for this rule.

Case $\dfrac{\Gamma \vdash A : \langle\alpha^*\rangle\phi \quad \Gamma, s : \phi \vdash B : \psi \quad \Gamma, g : \langle\alpha\rangle\langle\alpha^*\rangle\phi \vdash C : \psi}{\Gamma \vdash \langle \mathrm{case}_*\ A \text{ of } s \Rightarrow\ B \mid g \Rightarrow C\rangle : \psi}$

Assume (1) some d_1 such that for all $(\mathsf{c},\omega) \in [\![\bigwedge \Gamma]\!]$ have $(\mathsf{d}_1\,\mathsf{c},\omega) \in [\![\langle\alpha^*\rangle\phi]\!]$ by $\mathcal{D}_1$. Assume (2) some d_2 such that for all $(\mathsf{c}_2,\omega) \in [\![\bigwedge(\Gamma, s : \phi)]\!]$ have $(\mathsf{d}_2\,\mathsf{c}_2,\omega) \in [\![\psi]\!]$ by $\mathcal{D}_2$. Assume (3) some d_3 such that for all $(\mathsf{c}_3,\omega) \in [\![\bigwedge(\Gamma, g : \langle\alpha\rangle\langle\alpha^*\rangle\phi)]\!]$ have $(\mathsf{d}_3\,\mathsf{c}_3,\omega) \in [\![\psi]\!]$ by $\mathcal{D}_3$. Assume (4) some $(\mathsf{c},\omega) \in [\![\bigwedge \Gamma]\!]$.

We will define realizer $\mathsf{rz}(\mathsf{c}) = (\mathsf{c}, \pi_1\mathsf{d}_1\,\mathsf{c})$ which happens to realize the contexts for both $\mathcal{D}_2$ and $\mathcal{D}_3$. That is, first apply (1) to (4) to get $(\mathsf{d}_1\,\mathsf{c},\omega) \in [\![\langle\alpha^*\rangle\phi]\!]$ which expands to $\{(\mathsf{d}_1\,\mathsf{c},\omega)\}\langle\!\langle\alpha^*\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\}$ which by Lemma 4.8 gives $\bigcup_{k\in\mathbb{N}} \{(\mathsf{d}_1\,\mathsf{c},\omega)\}\langle\!\langle\alpha\rangle\!\rangle^k_{\langle 0\rangle} \subseteq [\![\phi]\!] \cup \{\top\}$. The case $k = 0$ expands by definition of $\cdot\langle\!\langle\alpha\rangle\!\rangle^k$ and by Remark 4.1 into $\{(\mathsf{d}_1\,\mathsf{c},\omega)\}_{\langle 0\rangle} \subseteq [\![\phi]\!] \cup \{\top\}$ and specifically (2a) $\{(\mathsf{d}_1\,\mathsf{c},\omega)\}_{\langle 0\rangle} \subseteq [\![\phi]\!]$ since $\cdot_{\langle 0\rangle}$ never introduces $\top$. The $k > 0$ case expands to $(\{(\mathsf{b},\omega)\}_{\langle 1\rangle}\langle\!\langle\alpha\rangle\!\rangle)\langle\!\langle\alpha^*\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\}$ so that (3a) $\{(\mathsf{b},\omega)\}_{\langle 1\rangle} \subseteq [\![\langle\alpha\rangle\langle\alpha^*\rangle\phi]\!]$.

Coming back to rz, note when $\pi_0\mathsf{d}_1\,\mathsf{c} = 0$ then $\{(\mathsf{d}_1\,\mathsf{c},\omega)\}_{\langle 0\rangle} = \{(\pi_1(\mathsf{d}_1\,\mathsf{c}),\omega)\}$ so by (2a) have (2b) $\{(\pi_1(\mathsf{d}_1\,\mathsf{c}),\omega)\} \subseteq [\![\phi]\!]$, likewise when $\pi_0(\mathsf{d}_1\,\mathsf{c}) = 1$ have (3b) $\{(\pi_1(\mathsf{d}_1\,\mathsf{c}),\omega)\} \subseteq [\![\langle\alpha\rangle\langle\alpha^*\rangle\phi]\!]$ by (3a).

In both (2b) and (3b), the same realizer $\pi_1(\mathsf{d}_1\,\mathsf{c})$ realizes the assumption used by $\mathcal{D}_2$ or $\mathcal{D}_3$ respectively.

In the former case $(\mathsf{rz}\,\mathsf{c},\omega)\in[\![\bigwedge(\Gamma,s:\phi)]\!]$ by (4) and (2b), so that by fact (2) we have (2c) $(\mathsf{d}_2\,(\mathsf{rz}\,\mathsf{c}),\omega)\in[\![\psi]\!]$. In the latter case $(\mathsf{rz}\,\mathsf{c},\omega)\in[\![\bigwedge(\Gamma,g:\langle\alpha\rangle\langle\alpha^*\rangle\phi)]\!]$ by (4) and (3b) so by (3) have (3c) $(\mathsf{d}_3\,(\mathsf{rz}\,\mathsf{c}),\omega)\in[\![\psi]\!]$.

Thus, we define $\mathsf{b}(\mathsf{c})=\mathtt{if}\,(\pi_0(\mathsf{d}_1\,\mathsf{c})=0)\,\{\mathsf{d}_2\,(\mathsf{rz}\,\mathsf{c})\}\,\mathtt{else}\,\{\mathsf{d}_3\,(\mathsf{rz}\,\mathsf{c})\}$ and we immediately have $(\mathsf{b}\,\mathsf{c},\omega)\in[\![\psi]\!]$ by (2c) and (3c) as desired.

Case $\dfrac{\Gamma\vdash M:\phi\wedge[\alpha][\alpha^*]\phi}{\Gamma\vdash[\text{roll }M]:[\alpha^*]\phi}$

Assume (1) some d such that for all $(\mathsf{c},\omega)\in[\![\bigwedge\Gamma]\!]$ have $(\mathsf{d}\,\mathsf{c},\omega)\in[\![\phi\wedge[\alpha][\alpha^*]\phi]\!]$ by $\mathcal{D}$. Assume (2) some $(\mathsf{c},\omega)\in[\![\bigwedge\Gamma]\!]$, so by (1) have $(\mathsf{d}\,\mathsf{c},\omega)\in[\![\phi\wedge[\alpha][\alpha^*]\phi]\!]$ giving (3a) $(\pi_0(\mathsf{d}\,\mathsf{c}),\omega)\in[\![\phi]\!]$ and (3b) $(\pi_1(\mathsf{d}\,\mathsf{c}),\omega)\in[\![[\alpha][\alpha^*]\phi]\!]$.

Suffice to show $(\mathsf{d}\,\mathsf{c},\omega)\in[\![[\alpha^*]\phi]\!]$, so note $(\mathsf{d}\,\mathsf{c},\omega)\in[\![[\alpha^*]\phi]\!]$ iff $\{(\mathsf{d}\,\mathsf{c},\omega)\}[\![\alpha^*]\!]\subseteq[\![\phi]\!]\cup\{\top\}$ iff$_{(*)}$ (G1) $\{(\mathsf{d}\,\mathsf{c},\omega)\}_{[0]}\subseteq[\![\phi]\!]\cup\{\top\}$ and (G2) $\{(\mathsf{d}\,\mathsf{c},\omega)\}[\![\alpha]\!][\![\alpha^*]\!]\subseteq[\![\phi]\!]\cup\{\top\}$).

The crucial step is step (*), which holds by Remark 4.1 and Lemma 4.8, by an argument analogous to those in cases $\langle*\rangle$G and $\langle*\rangle$S. Then goal (G1) is direct by (3a) and (G2) holds from (3b) since $(\pi_1\mathsf{d}\,\mathsf{c},\omega)\in[\![[\alpha][\alpha^*]\phi]\!]$ iff $\{(\pi_1\mathsf{d}\,\mathsf{c},\omega)\}[\![\alpha]\!][\![\alpha^*]\!]\subseteq[\![\phi]\!]\cup\{\top\}$.

Case $\dfrac{\Gamma\vdash M:[\alpha^*]\phi}{\Gamma\vdash[\text{unroll }M]:\phi\wedge[\alpha][\alpha^*]\phi}$

Assume (1) some d such that for $(\mathsf{c},\omega)\in[\![\bigwedge\Gamma]\!]$ have $(\mathsf{d}\,\mathsf{c},\omega)\in[\![[\alpha^*]\phi]\!]$ by $\mathcal{D}$. Assume (2) some $(\mathsf{c},\omega)\in[\![\bigwedge\Gamma]\!]$, so by (1) have $(\mathsf{d}\,\mathsf{c},\omega)\in[\![[\alpha^*]\phi]\!]$ iff$_{(*)}$ (3a) $(\pi_0\mathsf{d}\,\mathsf{c},\omega)\in[\![\phi]\!]$ and (3b) $(\pi_1\mathsf{d}\,\mathsf{c},\omega)\in[\![[\alpha][\alpha^*]\phi]\!]$. The crucial step is the step marked (*), which holds by Remark 4.1 and Lemma 4.8, by an argument analogous to those in the $(\langle\text{case}_*\ \cdot\ \text{of}\ s\Rightarrow\ \cdot\mid g\Rightarrow\cdot\rangle)$ branch of the proof.

Suffice to show $(\mathsf{d}\,\mathsf{c},\omega)\in[\![\phi\wedge[\alpha][\alpha^*]\phi]\!]$ iff (G1) $\{(\pi_0\mathsf{d}\,\mathsf{c},\omega)\}[\![\alpha^*]\!]\subseteq[\![\phi]\!]\cup\{\top\}$ and (G2) $\{(\pi_1\mathsf{d}\,\mathsf{c},\omega)\}[\![\alpha]\!][\![\alpha^*]\!]\subseteq[\![\phi]\!]\cup\{\top\}$. Goal (G1) is direct by (3a) and (G2) holds from (3b) since $(\pi_1\mathsf{d}\,\mathsf{c},\omega)\in[\![[\alpha][\alpha^*]\phi]\!]$ iff $\{(\pi_1\mathsf{d}\,\mathsf{c},\omega)\}[\![\alpha]\!][\![\alpha^*]\!]\subseteq[\![\phi]\!]\cup\{\top\}$.

Case $\dfrac{\Gamma\frac{y}{x},p:(x=f\frac{y}{x})\vdash M:\phi}{\Gamma\vdash[x:=f\frac{y}{x}\text{ in }p.\ M]:[x:=f]\phi}$

Assume (1) some d such that for all $(\mathsf{c}_2,\nu)\in[\![\bigwedge(\Gamma\frac{y}{x},p:(x=f\frac{y}{x}))]\!]$ have $(\mathsf{d}\,\mathsf{c}_2,\nu)\in[\![\phi]\!]$ by $\mathcal{D}$. Assume (2) some $(\mathsf{c},\omega)\in[\![\bigwedge\Gamma]\!]$. Assume side condition (SC) that y is fresh in Γ and $[x:=f]\phi$. By (SC), we lose no generality in assuming (RzB) $y\notin\mathrm{FV}(\mathsf{c})$. By Lemma A.5 on (2) have (2a) $(\mathsf{c}\frac{y}{x},\omega\frac{y}{x})\in[\![\bigwedge\Gamma\frac{y}{x}]\!]$. Let $v=[\![f]\!]\omega=[\![f\frac{y}{x}]\!]\omega\frac{y}{x}$ by Lemma A.5 and let $\nu=\omega\frac{y}{x}[x\mapsto v]$. Apply Lemma 4.11 on (2a) (changing only the state) to get (2b) $(\mathsf{c}\frac{y}{x},\nu)\in[\![\bigwedge\Gamma\frac{y}{x}]\!]$ since ν and $\omega\frac{y}{x}$ disagree only on x. Disagreement on x is allowed because x does not appear as a free variable in $\Gamma\frac{y}{x}$ nor, by renaming on (RzB), in $\mathsf{c}\frac{y}{x}$.

Now let $\mathsf{c}_2=(\mathsf{c}\frac{y}{x},\epsilon)$. Now get (3) $(\mathsf{c}_2,\nu)\in[\![\bigwedge\Gamma\frac{y}{x},p:(x=f\frac{y}{x})]\!]$ by (2b) and by construction of v,ν. Apply (3) on (1) to get (4) $(\mathsf{d}\,\mathsf{c}_2,\nu)\in[\![\phi]\!]$.

Let[5] $\mathsf{b}(\mathsf{c})=(\Lambda y:\mathbb{Q}.\ \mathsf{d}\,\mathsf{c}_2)\,[\![x]\!]\omega$.

[5]The use of $[\![x]\!]\omega$ here is a slight abuse which says that the realizer forces the value of x at state ω, even if b is not forced until a later state. This abuse of the realizer language could be avoided by changing the semantics of assignment to explicitly supply the old value $[\![x]\!]\omega$ as an argument to the realizer, but such a semantics would be non-obvious for all purposes except this soundness proof, so we choose to abuse the

Want to show $(\mathsf{b}\ \mathsf{c},\omega) \in [\![[x := f]\phi]\!]$, so it suffices to show $\{(\mathsf{b}\ \mathsf{c},\omega)\}[\![x := f]\!] \subseteq [\![\phi]\!] \cup \{\top\}$. Since $\{(\mathsf{b}\ \mathsf{c},\omega)\}[\![x := f]\!] = \{((\mathsf{b}\ \mathsf{c}),\omega[x \mapsto [\![f]\!]\omega])\}$, it suffices to show $((\mathsf{b}\ \mathsf{c}),\omega[x \mapsto [\![f]\!]\omega]) \in [\![\phi]\!] \cup \{\top\}$, of which we show $((\mathsf{b}\ \mathsf{c}),\omega[x \mapsto [\![f]\!]\omega]) \in [\![\phi]\!]$ specifically. Recall $[\![f]\!]\omega = [\![f]\!](\omega[y \mapsto \omega(x)])$ by freshness and $\mathsf{b}\ \mathsf{c} = (\mathsf{d}\,\mathsf{c}_2)_y^{\omega(x)}$, so it suffices to show (5) $((\mathsf{d}\,\mathsf{c}_2)_y^{\omega(x)}, \omega[x \mapsto [\![f]\!](\omega[y \mapsto \omega(x)])]) \in [\![\phi]\!]$. We derive (5) from (4).

Note $\nu = \omega[x \mapsto [\![f]\!]\omega][y \mapsto \omega(x)]$ by definition of renaming. By formula substitution Lemma A.9 on (4), we have (5a) $((\mathsf{d}\,\mathsf{c}_2)_y^{\omega(x)}, \omega[x \mapsto [\![f]\!]\omega]) \in [\![\phi_y^{\omega(x)}]\!] = [\![\phi]\!]$ since y is fresh in ϕ. Then recall (5b) $[\![f]\!](\omega[y \mapsto \omega(x)]) = [\![f]\!]\omega$ by Lemma 4.11 on term f because variable y is fresh in f. Fact (5) is immediate by replacing (5b) in (5a).

Case $\dfrac{\Gamma\frac{y}{x}, p : (x = f\frac{y}{x}) \vdash M : \phi}{\Gamma \vdash \langle x := f\frac{y}{x} \text{ in } p.\ M\rangle : \langle x := f\rangle\phi}$

Assume (1) some d such that for all $(\mathsf{c}_2,\nu) \in [\![\bigwedge(\Gamma\frac{y}{x}, p : (x = f\frac{y}{x}))]\!]$ have $(\mathsf{d}\,\mathsf{c}_2,\nu) \in [\![\phi]\!]$ by $\mathcal{D}$. Assume (2) some $(\mathsf{c},\omega) \in [\![\bigwedge\Gamma]\!]$. By Lemma A.5 have (2a) $(\mathsf{c}\frac{y}{x}, \omega\frac{y}{x}) \in [\![\bigwedge\Gamma\frac{y}{x}]\!]$. Let $v = [\![f]\!]\omega = [\![f\frac{y}{x}]\!]\omega\frac{y}{x}$ by Lemma A.5 and let $\nu = \omega\frac{y}{x}[x \mapsto v]$. Apply Lemma 4.11 on (2a) (changing only the state) to get (2b) $(\mathsf{c}\frac{y}{x}, \nu) \in [\![\bigwedge\Gamma\frac{y}{x}]\!]$ since ν and $\omega\frac{y}{x}$ disagree only on x fresh in $\Gamma\frac{y}{x}$.

Now let $\mathsf{c}_2 = (\mathsf{c}\frac{y}{x}, \epsilon)$. Now get (3) $(\mathsf{c}_2,\nu) \in [\![\bigwedge\Gamma\frac{y}{x}, p : (x = f\frac{y}{x})]\!]$ by (2b) and by construction of v,ν. Apply (3) on (1) to get (4) $(\mathsf{d}\,\mathsf{c}_2,\nu) \in [\![\phi]\!]$.

Let $\mathsf{b}(\mathsf{c}) = (\Lambda y : \mathbb{Q}.\ \mathsf{d}\,\mathsf{c}_2)\,[\![x]\!]\omega$.

Want to show $(\mathsf{b}\ \mathsf{c},\omega) \in [\![\langle x := f\rangle\phi]\!]$, so it suffices to show $\{(\mathsf{b}\ \mathsf{c},\omega)\}\langle\!\langle x := f\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\}$. Since $\{(\mathsf{b}\ \mathsf{c},\omega)\}\langle\!\langle x := f\rangle\!\rangle = \{((\mathsf{b}\ \mathsf{c}),\omega[x \mapsto [\![f]\!]\omega])\}$, it suffices to show $((\mathsf{b}\ \mathsf{c}),\omega[x \mapsto [\![f]\!]\omega]) \in [\![\phi]\!] \cup \{\top\}$, of which we show $((\mathsf{b}\ \mathsf{c}),\omega[x \mapsto [\![f]\!]\omega]) \in [\![\phi]\!]$ specifically. Recall $[\![f]\!]\omega = [\![f]\!]\omega[y \mapsto \omega(x)]$ by freshness and $\mathsf{b}\ \mathsf{c} = (\mathsf{d}\,\mathsf{c}_2)_y^{\omega(x)}$, so it suffices to show (5) $((\mathsf{d}\,\mathsf{c}_2)_y^{\omega(x)}, \omega[x \mapsto [\![f]\!]\omega[y \mapsto \omega(x)]]) \in [\![\phi]\!]$. We derive (5) from (4).

Note $\nu = \omega[x \mapsto [\![f]\!]\omega][y \mapsto \omega(x)]$ by definition of renaming. By formula substitution Lemma A.9 on (4), we have (5a) $((\mathsf{d}\,\mathsf{c}_2)_y^{\omega(x)}, \omega[x \mapsto [\![f]\!]\omega]) \in [\![\phi_y^{\omega(x)}]\!] = [\![\phi]\!]$ since y was fresh in ϕ. Then recall (5b) $[\![f]\!]\omega[y \mapsto \omega(x)] = [\![f]\!]\omega$ by Lemma 4.11 on term f because variable y is fresh in f. Fact (5) is immediate by replacing (5b) in (5a).

Case $\dfrac{\Gamma \vdash M : \langle x := *\rangle\phi \quad \Gamma\frac{y}{x}, p : \phi \vdash N : \psi}{\Gamma \vdash \text{unpack}(M, py.\ N) : \psi}$

Assume side condition (SC) that y is fresh in $\Gamma, \langle x := *\rangle\phi, \psi$ and $x \notin \text{FV}(\psi)$.

Assume (1) some d_1 such that for all $(\mathsf{c},\omega) \in [\![\bigwedge\Gamma]\!]$ have $(\mathsf{d}_1\,\mathsf{c},\omega) \in [\![\langle x := *\rangle\phi]\!]$ by $\mathcal{D}_1$. Assume (2) some d_2 such that for all $(\mathsf{c}_2,\nu) \in [\![\bigwedge(\Gamma\frac{y}{x}, p : \phi)]\!]$ have $(\mathsf{d}_2\,\mathsf{c}_2,\nu) \in [\![\psi]\!]$ by $\mathcal{D}_2$. Assume (3) some $(\mathsf{c},\omega) \in [\![\bigwedge\Gamma]\!]$. By (SC) we have that y is fresh in Γ, so that no generality is lost in assuming (RzB) that $y \notin \text{FV}(\mathsf{c})$. From (1) and (3) have $(\mathsf{d}_1\,\mathsf{c},\omega) \in [\![\langle x := *\rangle\phi]\!]$. Note $(\mathsf{d}_1\,\mathsf{c},\omega) \in [\![\langle x := *\rangle\phi]\!]$ iff $\{(\mathsf{d}_1\,\mathsf{c},\omega)\}\langle\!\langle x := *\rangle\!\rangle \subseteq [\![\phi]\!] \cup \{\top\}$. Then $\{(\mathsf{d}_1\,\mathsf{c},\omega)\}\langle\!\langle x := *\rangle\!\rangle$ expands to $(\pi_1(\mathsf{d}_1\,\mathsf{c}), \omega[x \mapsto \pi_0(\mathsf{d}_1\,\mathsf{c})]) \in [\![\phi]\!]$, then by freshness of y in ϕ and by Lemma 4.11 (changing only the term) have:

(4a) $(\pi_1(\mathsf{d}_1\,\mathsf{c}), \omega\frac{y}{x}[x \mapsto \pi_0(\mathsf{d}_1\,\mathsf{c})]) \in [\![\phi]\!]$. Also (4b) $(\mathsf{c}\frac{y}{x}, \omega\frac{y}{x}[x \mapsto \pi_0(\mathsf{d}_1\,\mathsf{c})]) \in [\![\bigwedge\Gamma\frac{y}{x}]\!]$ by Lemma A.5 on (3) and by Lemma 4.11 on the context, whose precondition is met because $x \notin \text{FV}(\Gamma\frac{y}{x})$ by freshness of y in Γ and because $x \notin \text{FV}(\mathsf{c}\frac{y}{x})$ by renaming and (RzB). Then

semantics here instead.

from (4a) and (4b) have $((\mathsf{c}\frac{y}{x}, \pi_1(\mathsf{d}_1\,\mathsf{c})), \omega\frac{y}{x}[x \mapsto \pi_0(\mathsf{d}_1\,\mathsf{c})]) \in [\![\bigwedge \Gamma\frac{y}{x}, p : \phi]\!]$ so from fact (2) we have the fact (5) $(\mathsf{d}_2\,(\mathsf{c}\frac{y}{x}, \pi_1(\mathsf{d}_1\,\mathsf{c})), \omega\frac{y}{x}[x \mapsto \pi_0\mathsf{d}_1\,\mathsf{c}]) \in [\![\psi]\!]$. Lastly note $\omega = \omega\frac{y}{x}[x \mapsto \pi_0(\mathsf{d}_1\,\mathsf{c})]$ on $\mathrm{FV}(\psi) \subseteq \{x, y\}^{\complement}$, so let $\mathsf{b}(\mathsf{c}) = \mathsf{d}_2\,(\mathsf{c}\frac{y}{x}, \pi_1(\mathsf{d}_1\,\mathsf{c}))$ then $(\mathsf{b}, \omega) \in [\![\psi]\!]$ as desired.

Case $\dfrac{\Gamma \vdash A : \langle\alpha^*\rangle\phi \quad s : \phi \vdash B : \psi \quad g : \langle\alpha\rangle\psi \vdash C : \psi}{\Gamma \vdash FP(A, s.\ B, g.\ C) : \psi}$

Assume (1) some d_1 such that for $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$ have $(\mathsf{d}_1\,\mathsf{c}, \omega) \in [\![\langle\alpha^*\rangle\phi]\!]$ by $\mathcal{D}_1$. Assume (2) some d_2 such that for $(\mathsf{c}_2, \nu) \in [\![\bigwedge(s : \phi)]\!]$ have $(\mathsf{d}_2\,\mathsf{c}_2, \nu) \in [\![\psi]\!]$ by $\mathcal{D}_2$. Assume (3) some d_3 such that for $(\mathsf{c}_3, \nu) \in [\![\bigwedge(g : \langle\alpha\rangle\psi)]\!]$ have $(\mathsf{d}_3\,\mathsf{c}_3, \nu) \in [\![\psi]\!]$ by $\mathcal{D}_3$. Assume (4) some $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$, so by (1) have (1a) $(\mathsf{d}_1\,\mathsf{c}, \omega) \in [\![\langle\alpha^*\rangle\phi]\!]$.

Let $\mathsf{rz}(\mathsf{c}) = \mathsf{mon}_{\langle\alpha^*\rangle}\ \mathsf{d}_2\ (\mathsf{d}_1\,\mathsf{c})$, then by Lemma A.2 on (1a) and (2) have (5) $(\mathsf{rz}\,\mathsf{c}, \omega) \in [\![\langle\alpha^*\rangle\phi]\!]$. In the application of Lemma A.2, we let $S = \mathcal{S}$, meaning we prove that the monotone implication holds everywhere, not only in states resulting from executing α^*.

Now define $\mathsf{b}(\mathsf{rz})(\mathsf{c}) = \mathtt{if}\ (\pi_0(\mathsf{rz}\,\mathsf{c}) = 0)\ \pi_1(\mathsf{rz}\,\mathsf{c})\ \mathtt{else}\ \mathsf{d}_3\,\mathsf{mon}_{\langle\alpha\rangle}\ (\mathsf{b}\ \mathsf{c})\ (\pi_1(\mathsf{rz}\,\mathsf{c}))$, which is an infinite recursive definition that will nonetheless terminate when executed by unrolling the inductive structure of $\langle\alpha^*\rangle\psi$.

We want to show $(\mathsf{b}\ \mathsf{rz}\ \mathsf{c}, \omega) \in [\![\psi]\!]$. To do so, we start by expanding (5) according to Lemma 4.8 to get (5a) $\left(\bigcup_{k\in\mathbb{N}} \{(\mathsf{rz}\,\mathsf{c}, \omega)\}\langle\!\langle\alpha\rangle\!\rangle^k\right)_{\langle 0\rangle} \subseteq [\![\psi]\!] \cup \{\top\}$.

We proceed by induction. We will split by cases on k from (5a), but formally speaking the induction is on the fixed-point construction of the loop semantics.

As the first case consider $k = 0$, then $\pi_0(\mathsf{rz}\,\mathsf{c}) = 0$, i.e., the loop terminates immediately. In that case $\{(\mathsf{rz}, \omega)\}\langle\!\langle\alpha\rangle\!\rangle^0_{\langle 0\rangle} = \{(\mathsf{rz}\,\mathsf{c}, \omega)\}_{\langle 0\rangle} = \{(\pi_1\mathsf{rz}\,\mathsf{c}, \omega)\}$ and by (5a) $\{(\pi_1\mathsf{rz}\,\mathsf{c}, \omega)\} \subseteq [\![\psi]\!] \cup \{\top\}$, specifically $[\![\psi]\!]$ since Demon has not failed a test. Since the $\pi_0(\mathsf{rz}\,\mathsf{c}) = 0$ case of b just calls $\pi_1(\mathsf{rz}\,\mathsf{c})$, then $(\mathsf{b}\ \mathsf{rz}\ \mathsf{c}, \omega) \in [\![\psi]\!]$ as desired.

In the second case $k = i + 1$, then (6) $\pi_0(\mathsf{rz}\,\mathsf{c}) = 1$, i.e., the loop progresses. Consider (b, ω) and work backwards to show $(\mathsf{b}\ \mathsf{rz}\ \mathsf{c}, \omega) \in [\![\psi]\!]$:

$$\begin{aligned}
&(\mathsf{b}\ \mathsf{rz}\ \mathsf{c}, \omega) \in [\![\psi]\!] \\
\text{iff }_{(6)} &(\mathsf{d}_3\,(\mathsf{mon}_{\langle\alpha\rangle}\ (\mathsf{b}\ \mathsf{c})\ (\pi_1(\mathsf{rz}\,\mathsf{c}))), \omega) \in [\![\psi]\!] \\
\text{iff }_{(3)} &(\mathsf{mon}_{\langle\alpha\rangle}\ (\mathsf{b}\ \mathsf{c})\ (\pi_1(\mathsf{rz}\,\mathsf{c})), \omega) \in [\![\langle\alpha\rangle\psi]\!]
\end{aligned}$$

It now suffices to show $(\mathsf{mon}_{\langle\alpha\rangle}\ (\mathsf{b}\ \mathsf{c})\ (\pi_1(\mathsf{rz}\,\mathsf{c})), \omega) \in [\![\langle\alpha\rangle\psi]\!]$, for which we apply Lemma A.2. Let R be the state projection of $\{(\pi_1(\mathsf{rz}\,\mathsf{c}), \omega)\}\langle\!\langle\alpha\rangle\!\rangle$, then we must prove the preconditions (PC1) $\{\mathsf{b}\ \mathsf{c}\} \times R \subseteq [\![\langle\alpha^*\rangle\psi \to \psi]\!]$ and (PC2) $(\pi_1(\mathsf{rz}\,\mathsf{c}), \omega) \in [\![\langle\alpha\rangle\langle\alpha^*\rangle\psi]\!]$.

Precondition (PC2) follows from (5) by applying the typical loop unfolding definition using (6) and Lemma 4.8. Precondition (PC1) holds because it is the inductive hypothesis from induction on the fixed-point construction. It is the inductive hypothesis because the starting region R of (PC1) was constructed to unfold one iteration of the fixed-point semantics. Then (PC1) is immediate from the induction hypothesis.

Case $\dfrac{\Gamma \vdash A : \varphi \quad p : \varphi, q : \mathcal{M}_0{=}\mathcal{M}{\succ}\mathbf{0} \vdash B : \langle\alpha\rangle(\varphi \wedge \mathcal{M}_0{\succ}\mathcal{M}) \quad p : \varphi, q : \mathbf{0}{\succcurlyeq}\mathcal{M} \vdash C : \phi}{\Gamma \vdash \mathrm{for}(p : \varphi(\mathcal{M}) = A; q; B)\,\{\alpha\}\,C : \langle\alpha^*\rangle\phi}$

Assume (1) some d_1 s.t. for all $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$ have $(\mathsf{d}_1\,\mathsf{c}, \omega) \in [\![\varphi]\!]$ by $\mathcal{D}_1$. Assume (2) some d_2 s.t. for all $(\mathsf{c}_2, \nu) \in [\![\bigwedge(p : \varphi, q : (\mathcal{M}_0{=}\mathcal{M}{\succ}\mathbf{0}))]\!]$ have $(\mathsf{d}_2\,\mathsf{c}_2, \nu) \in [\![\langle\alpha\rangle(\varphi \wedge \mathcal{M}_0{\succ}\mathcal{M})]\!]$ by $\mathcal{D}_2$. Assume (3) some d_3 s.t. for all $(\mathsf{c}_3, \nu) \in [\![\bigwedge(p : \varphi, q : (\mathbf{0}{\succcurlyeq}\mathcal{M}))]\!]$ have $(\mathsf{d}_3\,\mathsf{c}_3, \nu) \in [\![\phi]\!]$ by $\mathcal{D}_3$. Assume (4) some $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$, so by (1) have $(\mathsf{d}_1\,\mathsf{c}, \omega) \in [\![\varphi]\!]$.

In order to apply Lemma A.3, we restate (2) as

$$(\{\mathsf{d}_2\} \times \mathcal{S}) \subseteq [\![(\varphi \wedge (\mathcal{M}_0 = \mathcal{M} \wedge \mathcal{M} \succ \mathbf{0}) \to \langle\alpha\rangle(\varphi \wedge \mathcal{M}_0 \succ \mathcal{M}))]\!]$$

which generalizes to (2a)

$$(\{\Lambda\mathcal{M}_0 : \mathbb{Q}.\ \mathsf{d}_2\} \times \mathcal{S}) \subseteq [\![\forall\mathcal{M}_0\,(\varphi \wedge (\mathcal{M}_0 = \mathcal{M} \wedge \mathcal{M} \succ \mathbf{0}) \to \langle\alpha\rangle(\varphi \wedge \mathcal{M}_0 \succ \mathcal{M}))]\!]$$

We also restate (3) as (3a)

$$(\{\mathsf{d}_3\} \times \mathcal{S}) \subseteq [\![\varphi \wedge \mathbf{0} \succcurlyeq \mathcal{M} \to \phi]\!]$$

Then the assumptions of Lemma A.3 are discharged by (4), (2a), (3a), so

$$(\mathsf{ind}(\mathsf{d}_1\,\mathsf{c},\ (\Lambda\mathcal{M}_0 : \mathbb{Q}.\ \mathsf{d}_2),\ \mathsf{d}_3)_{\mathcal{M}}, \omega) \in [\![\langle\alpha^*\rangle\phi]\!]$$

and the case follows immediately by letting $\mathsf{b}(\mathsf{c}) = (\mathsf{ind}(\mathsf{d}_1\,\mathsf{c},\ (\Lambda\mathcal{M}_0 : \mathbb{Q}.\ \mathsf{d}_2),\ \mathsf{d}_3)_{\mathcal{M}}$.

Case $\Gamma, p : \phi \vdash p : \phi$: We prove the case for the hypothesis rule. Assume (1) some c such that $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma, p : \phi]\!]$. Let $\mathsf{b}(\mathsf{c}) = \pi_1\mathsf{c}$ then by (1) and semantics of conjunction have $(\mathsf{b}(\mathsf{c}), \omega) \in [\![\phi]\!]$ as desired.

Case $\dfrac{\Gamma \vdash M : \rho}{\Gamma \vdash \mathsf{FO}[\phi](M) : \phi}$

Assume (1) some d such that for all $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$ have $(\mathsf{d}\,\mathsf{c}, \omega) \in \rho$ by $\mathcal{D}$. Assume (2) some c such that $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$. Assume by side condition (SC) some b such that $\{\mathsf{b}\} \times \mathcal{S} \subseteq [\![\rho \to \phi]\!]$.

Then by (2) and (1) have $(\mathsf{d}\,\mathsf{c}, \omega) \in \rho$ and by (SC) have $(\mathsf{b}\,(\mathsf{d}\,\mathsf{c}), \omega) \in \phi$ which proves the case. Soundness of rules Dec and split are immediate corollaries.

Case $\dfrac{\Gamma \vdash M : [x := *]\phi}{\Gamma \vdash (M\ f) : \phi^f_x}$

Assume (1) some d such that for all $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$ have $(\mathsf{d}\,\mathsf{c}, \omega) \in [\![[x := *]\phi]\!]$ by $\mathcal{D}$. Assume (2) some c such that $(\mathsf{c}, \omega) \in [\![\bigwedge \Gamma]\!]$ so that (1a) $(\mathsf{d}\,\mathsf{c}, \omega) \in [\![[x := *]\phi]\!]$. Assume the side condition that (SC) ϕ^f_x is admissible. Fact (1a) expands: $(\mathsf{d}\,\mathsf{c}, \omega) \in [\![[x := *]\phi]\!]$ iff $\{(\mathsf{d}\,\mathsf{c}, \omega)\}[\![x := *]\!] \subseteq [\![\phi]\!] \cup \{\top\}$ iff$_\top$ $\{(\mathsf{d}\,\mathsf{c}, \omega)\}[\![x := *]\!] \subseteq [\![\phi]\!]$ iff $\{(\mathsf{d}\,\mathsf{c}\,v), \omega[x \mapsto v] \mid v \in \mathbb{Q}\} \subseteq [\![\phi]\!]$, call this fact (1b). The line marked $\top$ holds because the $x := *$ semantics never introduce $\top$. In (1b), choose $v = [\![f]\!]\omega$ to get (3) $(\mathsf{d}\,\mathsf{c}\,[\![f]\!]\omega, \omega[x \mapsto [\![f]\!]\omega]) \in [\![\phi]\!]$. By (SC) we can apply Lemma A.9 on (3) to get (4) $(\mathsf{d}\,\mathsf{c}\,[\![f]\!]\omega, \omega) \in [\![\phi^f_x]\!]$, then let $\mathsf{b}(\mathsf{c}) = (\mathsf{d}\,\mathsf{c}\,f)$ so the case holds by (4). □

A.4.7 Proof Theory

The following theorems concern proof terms, normal forms, progress, and preservation.

Lemma A.10 (Normal forms)**.** *If $\cdot \vdash M : \rho$ and M normal, then either M is $\langle$case A of $\ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$ for simple A and normal B and C, else M starts with the canonical introduction rule(s) for the top-level connective of ρ, all subterms are well-typed, and all subterms not under a binder are simple.*

For example, M has form:

- $\lambda p : \phi.\ N$ *if* $\rho \equiv \phi \to \psi$
- $\langle A, B\rangle$ *if* $\rho \equiv \langle ?\phi\rangle\psi$
- $\langle \ell \cdot N\rangle$ *or* $\langle r \cdot N\rangle$ *if* $\rho \equiv \langle \alpha \cup \beta\rangle\phi$
- $\langle go\ N\rangle$ *or* $\langle stop\ N\rangle$ *if* $\rho \equiv \langle \alpha^*\rangle\phi$
- $[roll\ N]$ *if* $\rho \equiv [\alpha^*]\phi$

Proof. Assume M is normal, then by definition, no reduction rule can be applied. First case-analyze the top-level rule application of $\cdot \vdash M : \rho$. In each case, M must be either an introduction rule for ρ, some elimination rule, or the monotonicity rule. In the first case, we are done. Otherwise, proceed by cases on the remaining rules: monotonicity and elimination rules. Every rule except $\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$ has a reduction rule, contradicting the assumption that M is normal. In the remaining case that M has form $\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$, the claim already holds since case analyses are normal. □

Lemma 4.16 (Progress)**.** *If* $\cdot \vdash M : \phi$*, then either* M *is normal or* $M \mapsto M'$ *for some* M'*.*

Proof. By induction on the derivation.

Case $[:*]$I, $\langle:*\rangle$I, $[?]$I, or $\langle\!\langle:=\rangle\!\rangle$I: In these cases, M is already normal by definition.

Case $\langle?\rangle$I, term $\langle M, N\rangle$: By first IH1, M normal or $M \mapsto M'$. If $M \mapsto M'$ then $\langle M, N\rangle \mapsto \langle M', N\rangle$ by rule $\langle\cup\rangle$S1. Else, by the IH2, N normal or $N \mapsto N'$. If $N \mapsto N'$ then $\langle M, N\rangle \mapsto \langle M, N'\rangle$ by rule $\langle\cup\rangle$S2. Else, M normal and N normal. If $M \equiv \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$ then $\langle\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle, N\rangle \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \langle B, N\rangle \mid r \Rightarrow\ \langle C, N\rangle\rangle$ by rule $\langle\cup\rangle$C1. Else if $N \equiv \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$ then $\langle M, \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rangle \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \langle M, B\rangle \mid r \Rightarrow\ \langle M, C\rangle\rangle$ by rule $\langle\cup\rangle$C2. Else M simp and N simp so $\langle\!\langle M, N\rangle\!\rangle$ simp and $\langle\!\langle M, N\rangle\!\rangle$ normal.

Case $[\cup]$I is symmetric.

Case $\langle\cup\rangle$I1, term $\langle \ell \cdot M\rangle$: By the IH, either $M \mapsto M'$ or M normal. If $M \mapsto M'$, then $\langle \ell \cdot M\rangle \mapsto \langle \ell \cdot M'\rangle$ by rule $\langle\!\langle \ell\cdot\rangle\!\rangle$S. Else, either M simp or $M \equiv \langle \text{case } A \text{ of } p \Rightarrow\ B \mid q \Rightarrow\ C\rangle$. If $M \equiv \langle \text{case } A \text{ of } p \Rightarrow\ B \mid q \Rightarrow\ C\rangle$ then $\langle \ell \cdot \langle \text{case } A \text{ of } p \Rightarrow\ B \mid q \Rightarrow\ C\rangle\rangle \mapsto \langle \text{case } A \text{ of } p \Rightarrow\ \langle \ell \cdot B\rangle \mid q \Rightarrow\ \langle \ell \cdot C\rangle\rangle$ by rule $\langle \ell\cdot\rangle$C. Else M simp so $\langle \ell \cdot M\rangle$ simp.

Case $\langle*\rangle$S is symmetric.

Case $\langle\cup\rangle$I2, term $\langle r \cdot M\rangle$: By the IH, either $M \mapsto M'$ or M normal. If $M \mapsto M'$, then $\langle r \cdot M\rangle \mapsto \langle r \cdot M'\rangle$ by rule $\langle\!\langle r\cdot\rangle\!\rangle$S. Else, either M simp or $M \equiv \langle \text{case } A \text{ of } p \Rightarrow\ B \mid q \Rightarrow\ C\rangle$. If $M \equiv \langle \text{case } A \text{ of } p \Rightarrow\ B \mid q \Rightarrow\ C\rangle$ then $\langle r \cdot \langle \text{case } A \text{ of } p \Rightarrow\ B \mid q \Rightarrow\ C\rangle\rangle \mapsto \langle \text{case } A \text{ of } p \Rightarrow\ \langle r \cdot B\rangle \mid q \Rightarrow\ \langle r \cdot C\rangle\rangle$ by rule $\langle r\cdot\rangle$C. Else M simp so $\langle r \cdot M\rangle$ simp.

Case $\langle*\rangle$G is symmetric.

Case $\langle\!\langle;\rangle\!\rangle$I (both box and diamond): By the IH, either $M \mapsto M'$ or M normal. If $M \mapsto M'$, then $\langle\!\langle \iota\ M\rangle\!\rangle \mapsto \langle\!\langle \iota\ M'\rangle\!\rangle$ by rule $[\iota]$S or $\langle\iota\rangle$S. Else, either M simp or $M \equiv \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$. If $M \equiv \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$ then $\langle\!\langle \iota\ \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rangle\!\rangle \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \langle\!\langle \iota\ B\rangle\!\rangle \mid r \Rightarrow\ \langle\!\langle \iota\ C\rangle\!\rangle\rangle$ by rule $[\iota]$C or $\langle\iota\rangle$C. Else M simp so $\langle\!\langle \iota\ M\rangle\!\rangle$ is normal.

Case $\langle\!\langle^d\rangle\!\rangle$I (both box and diamond): By the IH, either $M \mapsto M'$ or M normal. If $M \mapsto M'$, then $\langle\!\langle \text{yield } M\rangle\!\rangle \mapsto \langle\!\langle \text{yield } M'\rangle\!\rangle$ by rule $[^d]$S or $\langle^d\rangle$S. Else, either M simp or $M \equiv \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$. If $M \equiv \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$ then

$\langle\!|\iota\ \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle|\!\rangle \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \langle\!|\text{yield } B|\!\rangle \mid r \Rightarrow\ \langle\!|\text{yield } C|\!\rangle\rangle$ by rule $[^d]$C or $\langle^d\rangle$C. Else M simp so $\langle\!|\text{yield } M|\!\rangle$ is normal.

Case $\langle*\rangle$I, term $\text{for}(p:\varphi(\mathcal{M})=A;q;B)\,\{\alpha\}\,C$: Abbreviate term

$$M = [A, \langle rr, r\rangle/p, q]B\circ_t\text{for}(p:\varphi(\mathcal{M})=\langle\pi_L t\rangle;q;B)\,\{\alpha\}\,C$$

By the IH, either $A \mapsto A'$ or A normal. If $A \mapsto A'$ then we are able to apply rule forS to get $\text{for}(p:\varphi(\mathcal{M})=A;q;B)\,\{\alpha\}\,C \mapsto \text{for}(p:\varphi(\mathcal{M})=A';q;B)\,\{\alpha\}\,C$. Otherwise we have A normal so A simp or $A \equiv \langle \text{case } D \text{ of } \ell \Rightarrow\ E \mid r \Rightarrow\ F\rangle$. If A simp then

$$\begin{aligned}&\text{for}(p:\varphi(\mathcal{M})=A;q;B)\,\{\alpha\}\,C\\ \mapsto\ &\langle\text{case split } [\mathcal{M}\sim 0] \text{ of}\\ &\quad \ell \Rightarrow\ \langle\text{stop } [(A,\ell)/(p,q)]C\rangle\\ &\quad \mid r \Rightarrow\ \text{Ghost}[\mathcal{M}_0=\mathcal{M}](rr.\ \langle\text{go } M\rangle)\rangle\end{aligned}$$

by rule forβ. Otherwise by rule forC we have

$$\begin{aligned}&\text{for}(p:\varphi(\mathcal{M})=\langle \text{case } D \text{ of } \ell \Rightarrow\ E \mid r \Rightarrow\ F\rangle;q;B)\,\{\alpha\}\,C\\ \mapsto\ &\langle \text{case } D \text{ of } \ell \Rightarrow\ \text{for}(p:\varphi(\mathcal{M})=E;q;B)\,\{\alpha\}\,C \mid r \Rightarrow\ \text{for}(p:\varphi(\mathcal{M})=F;q;B)\,\{\alpha\}\,C\rangle\end{aligned}$$

Case FP, term $FP(A, s.\ B, g.\ C)$: By the IH, either $A \mapsto A'$ or A normal. If $A \mapsto A'$ then $FP(A, s.\ B, g.\ C) \mapsto FP(A', s.\ B, g.\ C)$ by FPS. Else, since A normal then $A \equiv \langle \text{case } D \text{ of } \ell \Rightarrow\ E \mid r \Rightarrow\ F\rangle$ or A simp. If $A \equiv \langle \text{case } D \text{ of } \ell \Rightarrow\ E \mid r \Rightarrow\ F\rangle$ then $FP(\langle \text{case } D \text{ of } \ell \Rightarrow\ E \mid r \Rightarrow\ F\rangle, s.\ B, g.\ C) \mapsto \langle \text{case } D \text{ of } \ell \Rightarrow\ FP(E, s.\ B, g.\ C) \mid r \Rightarrow\ FP(F, s.\ B, g.\ C)\rangle$ by rule FPC. If A simp then by rule FPβ

$$FP(A, s.\ B, g.\ C) \mapsto (\langle \text{case}_*\ A \text{ of } s \Rightarrow\ B \mid g \Rightarrow [r\circ_t FP(t, s.\ B, g.\ C)/q]C\rangle)$$

Case [?]E, term $M\ N$: By the IH, either $M \mapsto M'$ or M normal. If $M \mapsto M'$ then $M\ N \mapsto M'\ N$ by rule appS1. Else if $M \equiv \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$ then $\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\ N \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ B\ N \mid r \Rightarrow\ C\ N\rangle$ by rule app1C. Else M simp. By the IH, either $N \mapsto N'$ or N normal. If $N \mapsto N'$ then $M\ N \mapsto M\ N'$ by rule appS2. Else if $N \equiv \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$ then $M\ \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ M\ B \mid r \Rightarrow\ M\ C\rangle$ by rule app2C. Else N simp. By Lemma A.10 $M = (\lambda p:\phi.\ D)$ then $(\lambda p:\phi.\ D)\ N \mapsto [N/p]D$ by rule $\lambda\phi\beta$.

Case [:*]E, term $M\ f$: By the IH, $M \mapsto M'$ or M normal. If $M \mapsto M'$ then $M\ f \mapsto M'\ f$ by rule unroll. Else if $M \equiv \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$ then $\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\ f \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ B\ f \mid r \Rightarrow\ C\ f\rangle$ by rule appC. Else M simp and by Lemma A.10 then $M \equiv (\lambda x:\mathbb{Q}.\ D)$ so $(\lambda x:\mathbb{Q}.\ D)\ f \mapsto D^f_x$ by rule $\lambda\beta$.

Case $\langle?\rangle$E1, term $\langle\pi_L M\rangle$: By IH, $M \mapsto M'$ or M normal. If $M \mapsto M'$ then we have $\langle\pi_L M\rangle \mapsto \langle\pi_L M'\rangle$ by rule $\langle\!|\pi_L|\!\rangle$S. Else if $M \equiv \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$ then we have that $\langle\pi_L\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rangle \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \langle\pi_L B\rangle \mid r \Rightarrow\ \langle\pi_L C\rangle\rangle$ by rule $\langle\!|\pi_L|\!\rangle$C. Else M simp so by Lemma A.10 have $M \equiv \langle A, B\rangle$ so $\langle\pi_L\langle A, B\rangle\rangle \mapsto A$ by rule $\pi_L\beta$. The case for $[\cup]$E1 is symmetric.

Case $\langle ?\rangle$E2, term $\langle \pi_R M\rangle$: By IH, $M \mapsto M'$ or M normal. If $M \mapsto M'$ then we have $\langle \pi_R M\rangle \mapsto \langle \pi_R M'\rangle$ by rule $\langle\!|\pi_R|\!\rangle$S. Else if $M \equiv \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$ then we have that $\langle \pi_R\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rangle \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \langle \pi_R B\rangle \mid r \Rightarrow\ \langle \pi_R C\rangle\rangle$ by rule $\langle\!|\pi_R|\!\rangle$C. Else M simp so by Lemma A.10 have $M \equiv \langle A, B\rangle$ so $\langle \pi_R\langle\!|A, B|\!\rangle\rangle \mapsto B$ by rule $\pi_R\beta$. The case for rule $[\cup]$E2 is symmetric.

Case $[*]$E, term $[\text{unroll } M]$: By the IH, $M \mapsto M'$ or M normal. If $M \mapsto M'$ then $[\text{unroll } M] \mapsto [\text{unroll } M']$ by rule unroll. Else if $M \equiv \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$ then $[\text{unroll } \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle] \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ [\text{unroll } B] \mid r \Rightarrow\ [\text{unroll } C]\rangle$ by rule unrollC. Else M simp so by Lemma A.10 have $M \equiv [\text{roll } A]$ so $[\text{unroll } [\text{roll } A]] \mapsto A$ by rule unrollβ.

Case $\langle *\rangle$C, term $\langle \text{case}_*\ A \text{ of } s \Rightarrow\ B \mid g \Rightarrow C\rangle$: By the IH, $A \mapsto A'$ or A normal. If $A \mapsto A'$ then $\langle \text{case}_*\ A \text{ of } s \Rightarrow\ B \mid g \Rightarrow C\rangle \mapsto \langle \text{case}_*\ A' \text{ of } s \Rightarrow\ B \mid g \Rightarrow C\rangle$ by rule caseS. Else if $M \equiv \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$ then $\langle \text{case}_*\ \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \text{ of } s \Rightarrow\ D \mid g \Rightarrow E\rangle \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \langle \text{case}_*\ B \text{ of } s \Rightarrow\ D \mid g \Rightarrow E\rangle \mid r \Rightarrow\ \langle \text{case}_*\ C \text{ of } s \Rightarrow\ D \mid g \Rightarrow E\rangle\rangle$ by rule caseC. Else M simp so by Lemma A.10 have $M \equiv \langle \text{stop } A\rangle$ or $M \equiv \langle \text{go } A\rangle$ so $\langle \text{case}_*\ \langle \text{stop } A\rangle \text{ of } s \Rightarrow\ B \mid g \Rightarrow C\rangle \mapsto [A/s]B$ and $\langle \text{case}_*\ \langle \text{go } A\rangle \text{ of } s \Rightarrow\ B \mid g \Rightarrow C\rangle \mapsto [A/g]C$ by rules caseβL and caseβR.

Case $[\cup]$I, term $[M, N]$: By the IH, $M \mapsto M'$ or M normal. If $M \mapsto M'$ then $[M, N] \mapsto [M', N]$ by rule $[\cup]$S1. Else if $M \equiv \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$ then $[\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle, N] \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ [B, N] \mid r \Rightarrow\ [C, N]\rangle$ by rule $[\cup]$C1. Else M simp. By the IH, $N \mapsto N'$ or N normal. If $N \mapsto N'$ then $[M, N] \mapsto [M, N']$ by rule $[\cup]$S2. Else if $N \equiv \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$ then $[M, \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle] \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ [M, B] \mid r \Rightarrow\ [M, C]\rangle$ by rule $[\cup]$C2. Else N simp, so $[M, N]$ normal.

Case $\langle\cup\rangle$E, term $\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$: By the IH, $A \mapsto A'$ or A normal. If $A \mapsto A'$ then $\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \mapsto \langle \text{case } A' \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$ by rule caseS. Else A normal. If $A \equiv \langle \text{case } D \text{ of } \ell \Rightarrow\ E \mid r \Rightarrow\ F\rangle$ then $\langle \text{case } \langle \text{case } D \text{ of } \ell \Rightarrow\ E \mid r \Rightarrow\ F\rangle \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \mapsto \langle \text{case } D \text{ of } \ell \Rightarrow\ \langle \text{case } E \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \mid r \Rightarrow\ \langle \text{case } F \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rangle$ by rule caseC. Else A simp thus by Lemma A.10, $A \equiv \langle \ell \cdot M\rangle$ or $A \equiv \langle r \cdot M\rangle$. In the first case, $\langle \text{case } \langle \ell \cdot M\rangle \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \mapsto [M/\ell]B$ by rule caseβL else $\langle \text{case } \langle r \cdot M\rangle \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \mapsto [M/r]C$ by rule caseβR.

Case $[*]$I, term $(M \text{ rep } p : \psi.\ N \text{ in } O)$: By the IH, $M \mapsto M'$ or M normal. If $M \mapsto M'$ then $(M \text{ rep } p : \psi.\ N \text{ in } O) \mapsto (M' \text{ rep } p : \psi.\ N \text{ in } O)$ by rule repS. Else if $M \equiv \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$ then $(\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \text{ rep } p : \psi.\ N \text{ in } O) \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ (B \text{ rep } p : \psi.\ N \text{ in } O) \mid r \Rightarrow\ (C \text{ rep } p : \psi.\ N \text{ in } O)\rangle$ by rule repC. Else M simp meaning $(M \text{ rep } p : \psi.\ N \text{ in } O) \mapsto [\text{roll } \langle M, ([M/p]N)\circ_q(q \text{ rep } p : \psi.\ N \text{ in } O)\rangle]$ by rule repβ.

Case $\langle :*\rangle$E, term $\text{unpack}(M, py.\ N)$: By the IH, $M \mapsto M'$ or M normal. If $M \mapsto M'$ then $\text{unpack}(M, py.\ N) \mapsto \text{unpack}(M, py.\ N)$ by rule $\langle :*\rangle$S. Else if $M \equiv \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$ then $\text{unpack}(\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle, py.\ N) \mapsto \langle \text{case } A \text{ of } \ell \Rightarrow\ \text{unpack}(B, py.\ N) \mid r \Rightarrow\ \text{unpack}(C, py.\ N)\rangle$ by rule $\langle :*\rangle$C. Else M simp and by Lemma A.10 then $M \equiv \langle f\frac{y}{x} :* p.\ M'\rangle$ so rule unpackβ gives $\text{unpack}(\langle f\frac{y}{x} :* q.\ M\rangle, py.\ N) \mapsto (\text{Ghost}[x = f\frac{y}{x}](q.\ [M/p]N))$.

Case hyp, term x: By inversion on $\cdot \vdash p : \phi$ then $p \in \cdot$, so the case holds by absurdity.

Case M, term $M\circ_p N$: By the IH, $M \mapsto M'$ or M normal. If $M \mapsto M'$ then $M\circ_p N \mapsto$

$M' \circ_p N$ by rule $\circ$S, else if $M \equiv \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C \rangle$ then by rule $\circ$C

$$\begin{aligned} &\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C \rangle \circ_p N \\ \mapsto\ &\langle \text{case } A \text{ of } \ell \Rightarrow\ B \circ_p N \mid r \Rightarrow\ C \circ_p N \rangle \end{aligned}$$

Else M normal. We proceed by cases on M. By Lemma A.10, it suffices to show the cases for introduction forms. The termination argument is by lexicographic induction on the postcondition formula ϕ and on the derivation M.

Case $\langle ? \rangle$I, term $(\lambda p : \phi.\ M)$: By rule $\lambda\phi\circ$ have $(\lambda p : \phi.\ M) \circ_p N \mapsto (\lambda p : \phi.\ M \circ_p N)$

Case $[:*]$I, term $(\lambda x : \mathbb{Q}.\ M)$: By rule $\lambda\circ$ have $(\lambda x : \mathbb{Q}.\ M) \circ_p N \mapsto (\lambda x : \mathbb{Q}.\ (M \circ_p N))$

Case $[\cup]$I, term $[A, B]$: By rule $[\cup]\circ$ have $[A, B] \circ_q N \mapsto [A \circ_p (N^{\vec{a}}_{\vec{a}'}), B \circ_p (N^{\vec{b}}_{\vec{b}'})]$ where $\vec{a}$ and $\vec{a}'$ are defined as in rule $[\cup]\circ$.

Case $\langle ? \rangle$I, term $\langle A, B \rangle$: By rule $\langle\cup\rangle\circ$ have $\langle A, B \rangle \circ_p N \mapsto \langle A, [B/p]N \rangle$

Case $\langle :* \rangle$I, term $\langle f \frac{y}{x} :* p.\ M \rangle$: By rule $\langle :* \rangle\circ$, $\langle f \frac{y}{x} :* q.\ M \rangle \circ_p N \mapsto \langle f \frac{y}{x} :* q.\ [M/p]N \rangle$

Case $\langle\!\langle^d \rangle\!\rangle$I , term $\langle \text{yield } M \rangle$: By rule $\langle^d\rangle\circ$ have $\langle \text{yield } M \rangle \circ_p N \mapsto \langle \text{yield } M \circ_p N \rangle$

Case $\langle\!\langle^d \rangle\!\rangle$I , term $[\text{yield } M]$: By rule $[^d]\circ$ have $[\text{yield } M] \circ_p N \mapsto [\text{yield } M \circ_p N]$

Case $\langle\cup\rangle$I1, term $\langle \ell \cdot M \rangle$: By rule $\langle \ell \cdot \rangle\circ$ have $(\langle \ell \cdot M \rangle) \circ_p N \mapsto \langle \ell \cdot (M \circ_p N) \rangle$

Case $\langle\cup\rangle$I2, term $\langle r \cdot M \rangle$: By rule $\langle r \cdot \rangle\circ$ have $(\langle r \cdot M \rangle) \circ_p N \mapsto \langle r \cdot (M \circ_p N) \rangle$

Case $[*]$R, term $[\text{roll } M]$: By rule $[*]\circ$ have

$$[\text{roll } M] \circ_p N \mapsto [\text{roll } \langle ([\pi_L M]) \circ_p (N^{\vec{a}}_{\vec{a}'}), (\pi_R M) \circ_t (t \circ_p N) \rangle]$$

where $\vec{a} = \text{BV}(\alpha)$ for loop body α and where $\vec{y}$ are corresponding ghosts.

Case $\langle * \rangle$S, term $\langle \text{stop } M \rangle$: By rule stop$\circ$ have $\langle \text{stop } M \rangle \circ_p N \mapsto \langle \text{stop } ([M/p](N^{\vec{a}}_{\vec{a}'})) \rangle$ where $\vec{a}$ and $\vec{a}'$ are defined as in rule stop$\circ$.

Case $\langle * \rangle$G, term $\langle \text{go } M \rangle$: By rule go$\circ$ have $\langle \text{go } M \rangle \circ_p N \mapsto \langle \text{go } (M \circ_p N) \rangle$

Case $\langle\!\langle := \rangle\!\rangle$I (diamond), term $\langle x := f \frac{y}{x} \text{ in } p.\ M \rangle$: By rule $\langle := \rangle\circ$ have

$$\langle y := f \tfrac{x}{y} \text{ in } p.\ M \rangle \circ_p N \mapsto \langle y := f \tfrac{x}{y} \text{ in } p.\ [M/p]N \rangle$$

Case $[\![:=]\!]$I (box), term $[x := f \frac{y}{x} \text{ in } p.\ M]$: By rule $[:=]\circ$ have

$$[y := f \tfrac{x}{y} \text{ in } p.\ M] \circ_p N \mapsto [y := f \tfrac{x}{y} \text{ in } p.\ [M/p]N]$$

Case $[\![;]\!]$I (box), term $[\iota\ M]$: By rule $[\iota]\circ$ have $([\iota\ M]) \circ_p N \mapsto [\iota\ (M \circ_q q \circ_t [t/p]N)]$

Case $\langle\!\langle ; \rangle\!\rangle$I (diamond), term $\langle \iota\ M \rangle$: By rule $\langle \iota \rangle\circ$ have $(\langle \iota\ M \rangle) \circ_p N \mapsto \langle \iota\ (M \circ_q q \circ_t [t/p]N) \rangle$

□

Lemma 4.9 (Structurality)**.** *The structural rules W, X, and C are admissible, i.e., the conclusions are provable using existing rules whenever the premises are provable.*

$$\text{(W)}\quad \frac{\Gamma \vdash M : \phi}{\Gamma, p : \psi \vdash M : \phi} \qquad \text{(X)}\quad \frac{\Gamma, p : \phi, q : \psi \vdash M : \rho}{\Gamma, q : \psi, p : \phi \vdash M : \rho} \qquad \text{(C)}\quad \frac{\Gamma, p : \phi, q : \phi \vdash M : \rho}{\Gamma, p : \phi \vdash [p/q]M : \rho}$$

Proof. To show each rule admissible, prove that when the premises are provable, the conclusion is provable. Each proof is by induction on the proof term M.

Observe that the only premises regarding Γ are of the form $\Gamma(x) = \phi$. Such premises are preserved when extending Γ with a fresh proof variable $p : \phi$, as needed by weakening. Premises are also preserved under exchange, because contexts are understood as sets to begin with, i.e., the premise $\Gamma(p) = \phi$ is ignorant of the position of x. Premises are preserved modulo renaming under contraction because the assumption $\Gamma(p) = \phi$ is as good as the assumption $\Gamma(q) = \phi$.

We note briefly that the inductive hypothesis can always be applied when needed. The context Γ is kept universally quantified in the induction over M. When proving $\Gamma \vdash M : \phi$, we sometimes have assumptions in an empty or singleton context; for weakening we need not apply the IH at all in these cases, and for exchange and contraction we simply vary the context as we apply the inductive hypothesis. We also vary the context in the inductive hypothesis of the assignment rule. □

Lemma 4.10 (Uniform renaming)**.** *Let $M\frac{y}{x}$ be the renaming of program variable x to y (and vice-versa by transposition) within M, even if x and y are not fresh. If $\Gamma \vdash M : \phi$ then $\Gamma\frac{y}{x} \vdash M\frac{y}{x} : \phi\frac{y}{x}$.*

Proof. Induction on M analogous to the proof of Lemma 4.14. □

Lemma 4.13 (Arithmetic-term substitution)**.** *If $\Gamma \vdash M : \phi$ and the program variable substitution σ is admissible for Γ, M, and ϕ, then $\sigma(\Gamma) \vdash \sigma(M) : \sigma(\phi)$.*

Proof. Induction on M analogous to the proof of Lemma 4.14.

We present the case for [?]I, which demonstrates the use of substitution in each of contexts, proof terms, and formulas.

Case $\dfrac{\Gamma, p : \phi \vdash M : \psi}{\Gamma \vdash (\lambda p : \phi.\ M) : [?\phi]\psi}$

By the IH, have $\sigma(\Gamma, p : \phi) \vdash \sigma(M) : \sigma(\psi)$. Unfold the definition of substitution to get $\sigma(\Gamma, p : \phi) = (\sigma(\Gamma), p : \sigma(\phi))$, that is $\sigma(\Gamma), p : \sigma(\phi) \vdash \sigma(M) : \sigma(\psi)$. so that by [?]I have $\sigma(\Gamma) \vdash (\lambda p : \sigma(\phi).\ \sigma(M)) : [?\sigma(\phi)]\sigma(\psi)$. To complete the case, simply observe $\sigma(\lambda p : \phi.\ M) = (\lambda p : \sigma(\phi).\ \sigma(M))$ and $\sigma([?\phi]\psi) = [?\sigma(\phi)]\sigma(\psi)$ from which we conclude $\sigma(\Gamma) \vdash \sigma(\lambda p : \phi.\ M) : \sigma([?\phi]\psi)$.

In general, the substitution $\sigma(M)$ recursively applies the substitution $\sigma(\cdot)$ to every subderivation or CGL expression mentioned in M, with respective examples from the [?]I case being the subderivation for the body (written M in the case) and the annotation ϕ appearing in $(\lambda p : \phi.\ M)$. Another example of a CGL expression that gets substituted is the witness term in an Angelic nondeterministic assignment proof. Each case's conclusion $\Gamma^f_x \vdash M^f_x : \phi^f_x$ follows by the same CGL proof rule that was applied at the top level of M. □

Lemma 4.14 (Proof term substitution)**.** *Let $[N/p]M$ by the result of (proof term) substitution of proof term N for p in M. Proof term substitution implicitly renames proof variables if necessary to avoid (proof) variable capture. If $\Gamma, p : \psi \vdash M : \phi$ and $\Gamma \vdash N : \psi$ then $\Gamma \vdash [N/p]M : \phi$.*

Proof. By induction on the derivation $\Gamma, p:\psi \vdash M:\phi$, generalizing the context.

Case $\langle\!\![:=]\!\!\rangle$I (diamond): In this case to avoid a clash we say the substituted proof variable was t, not p. By premise, have $\Gamma\frac{y}{x}, t:\psi\frac{y}{x}, p:(x=f\frac{y}{x}) \vdash M:\phi$. Now regroup the context as $(\Gamma\frac{y}{x}, p:(x=f\frac{y}{x}), t:\psi\frac{y}{x})$. By Lemma 4.10 on N and weakening have $\Gamma\frac{y}{x}, p:(x=f\frac{y}{x}) \vdash N\frac{y}{x}:\psi\frac{y}{x}$. Then we can apply the IH on M while varying the context to $\Gamma\frac{y}{x}, p:(x=f\frac{y}{x})$ and varying N to equally-complex $N\frac{y}{x}$, which yields $\Gamma\frac{y}{x}, p:(x=f\frac{y}{x}) \vdash [N\frac{y}{x}/t]M:\phi$, then by $\langle\!\![:=]\!\!\rangle$I have $\Gamma \vdash \langle x:=f\frac{y}{x} \text{ in } p.\ [N\frac{y}{x}/t]M\rangle : [x:=f]\phi$. Note to avoid capture, the substitution case for assignment does a renaming, i.e., $[N/t]\langle x:=f\frac{y}{x} \text{ in } p.\ M\rangle = \langle x:=f\frac{y}{x} \text{ in } p.\ [N\frac{y}{x}/t]M\rangle$.

Case $\langle:*\rangle$I: By premise have $\Gamma\frac{y}{x}, t:\psi\frac{y}{x}, p:(x=f\frac{y}{x}) \vdash M:\phi$ for some term f. Now regroup the context as $(\Gamma\frac{y}{x}, p:(x=f\frac{y}{x}), t:\psi\frac{y}{x})$. By Lemma 4.10 on N and weakening have $\Gamma\frac{y}{x}, p:(x=f\frac{y}{x}) \vdash N\frac{y}{x}:\psi\frac{y}{x}$. Then apply IH on M, varying context to $\Gamma\frac{y}{x}, p:(x=f\frac{y}{x})$ and varying N to equally-complex $N\frac{y}{x}$, which yields $\Gamma\frac{y}{x}, p:(x=f\frac{y}{x}) \vdash [N\frac{y}{x}/t]M:\phi$, then by $\langle:*\rangle$I have $\Gamma \vdash \langle f\frac{y}{x} :* p.\ [N/t]M\rangle : \langle x:=*\rangle\phi$. The case holds since $[N/t]\langle f\frac{y}{x} :* p.\ M\rangle = \langle f\frac{y}{x} :* p.\ [N/t]M\rangle$.

Case $\langle:*\rangle$E, postcondition ρ, term $\text{unpack}(A, py.\ B)$, call the substituted variable t. Assume the side condition that $x \notin \text{FV}(\rho)$. By premise have $\mathcal{D}_1$ is $\Gamma, t:\psi \vdash A:\langle x:=*\rangle\phi$ and $\mathcal{D}_2$ is $(\Gamma, t:\psi)\frac{z}{x}, p:\phi \vdash B:\rho$. By the IH on $\mathcal{D}_1$ have $\Gamma \vdash [N/t]A:\langle x:=*\rangle\phi$. In $\mathcal{D}_2$ reorder the context as $(\Gamma\frac{z}{x}, p:\phi), t:\psi\frac{z}{x}$, then observe by Lemma 4.10 have $\Gamma\frac{z}{x} \vdash N\frac{z}{x}:\psi\frac{z}{x}$ and by weakening, $\Gamma\frac{z}{x}, p:\phi \vdash N\frac{z}{x}:\psi\frac{z}{x}$ so by the IH on $\mathcal{D}_2$ have $\Gamma\frac{z}{x}, p:\phi \vdash [N\frac{z}{x}/t]B:\rho$, so by $\langle:*\rangle$E have $\Gamma \vdash \text{unpack}([N/t]A, py.\ [N\frac{z}{x}/t]B):\rho$. Note that this case of substitution renames for Ns within B, i.e., $[N/t]\text{unpack}(A, py.\ B) = \text{unpack}([N/t]A, py.\ [N\frac{z}{x}/t]B)$, which is as desired.

Case $\langle*\rangle$I, variable t: By the premises, have $\mathcal{D}_1$ is $\Gamma, t:\psi \vdash A:\varphi$ and have $\mathcal{D}_2$ is $p:\varphi, q:(\mathcal{M}_0 = \mathcal{M} \succ \mathbf{0}) \vdash B:\langle\alpha\rangle(\varphi \wedge \mathcal{M}_0 \succ \mathcal{M})$ for fresh $\mathcal{M}_0$ and have $\mathcal{D}_3$ is $p:\varphi, q:(\mathbf{0} \succcurlyeq \mathcal{M}) \vdash C:\phi$. By the IH on $\mathcal{D}_1$ have $\Gamma \vdash [N/t]A:\varphi$, then reusing $\mathcal{D}_2$ and $\mathcal{D}_3$ as-is, apply rule $\langle*\rangle$I giving $\Gamma \vdash \text{for}(p:\varphi(\mathcal{M}) = [N/t]A; q; B)\,\{\alpha\}\,C : \langle\alpha^*\rangle\phi$. This completes the case because

$$[N/t](\text{for}(p:\varphi(\mathcal{M}) = A; q; B)\,\{\alpha\}\,C) = \text{for}(p:\varphi(\mathcal{M}) = [N/t]A; q; B)\,\{\alpha\}\,C$$

Note substitution does not substitute in B or C , which cannot access t to begin with.

Case FP, postcondition ρ: By the premises, have $\mathcal{D}_1$ is $\Gamma, t:\psi \vdash A:\langle\alpha^*\rangle\phi$, $\mathcal{D}_2$ is $p:\phi \vdash B:\rho$, and $\mathcal{D}_3$ is $q:\langle\alpha\rangle\rho \vdash C:\rho$. By the IH on $\mathcal{D}_1$ have $\Gamma \vdash [N/t]A:\langle\alpha^*\rangle\phi$, then apply rule FP reusing $\mathcal{D}_2$ and $\mathcal{D}_3$ as-is: yielding $\Gamma \vdash FP([N/t]A, s.\ B, g.\ C):\rho$, which completes the case since $[N/t]FP(A, s.\ B, g.\ C) = FP([N/t]A, s.\ B, g.\ C)$. Note that it does not substitute in the B and C branches because they cannot access t in the first place.

Case $\langle*\rangle$S, postcondition $\langle\alpha^*\rangle\phi$: By premise, $\Gamma, p:\psi \vdash M:\phi$. By the IH, $\Gamma \vdash [N/p]M:\phi$ and by $\langle*\rangle$S, $\Gamma \vdash \langle\text{stop } [N/p]M\rangle : \langle\alpha^*\rangle\phi$, then the case holds because $[N/p]\langle\text{stop } M\rangle = \langle\text{stop } [N/p]M\rangle$.

Case $\langle*\rangle$G, postcondition $\langle\alpha^*\rangle\phi$: By premise, $\Gamma, p:\psi \vdash M:\phi$. By the IH, $\Gamma \vdash [N/p]M:\langle\alpha\rangle\langle\alpha^*\rangle\phi$ and by $\langle*\rangle$G, $\Gamma \vdash \langle\text{go } [N/p]M\rangle : \langle\alpha^*\rangle\phi$, then note $[N/p]\langle\text{go } M\rangle = \langle\text{go } [N/p]M\rangle$ to complete the case.

Case $\langle*\rangle$C, postcondition ρ, assume without loss of generality p has been uniformly renamed so $p \notin \{s, g\}$. By $\mathcal{D}_1$, $\Gamma, p:\psi \vdash A:\langle\alpha^*\rangle\phi$. By $\mathcal{D}_2$, $\Gamma, p:\psi, s:\phi \vdash B:\rho$. By

$\mathcal{D}_3$, $\Gamma, p:\psi, g:\langle\alpha\rangle\langle\alpha^*\rangle\phi \vdash C:\rho$. By weakening on N and IH1, $\Gamma \vdash [N/p]A:\langle\alpha^*\rangle\phi$, by weakening on N and IH2, $\Gamma, s:\psi \vdash [N/p]B:\rho$, and by IH3, $\Gamma, g:\langle\alpha\rangle\langle\alpha^*\rangle\psi \vdash [N/p]C:\rho$, then by $\langle*\rangle$C, $\Gamma \vdash \langle \text{case}_* \ [N/p]A \text{ of } s \Rightarrow \ [N/p]B \mid g \Rightarrow [N/p]C\rangle : \rho$.

Case $\langle\cup\rangle$I1, postcondition $\langle\alpha\cup\beta\rangle\phi$: By premise, $\Gamma, p:\psi \vdash M:\langle\alpha\rangle\phi$. By the IH, $\Gamma \vdash [N/p]M:\langle\alpha\rangle\phi$, and by $\langle\cup\rangle$I1 $\Gamma \vdash \langle\ell\cdot[N/p]M\rangle:\langle\alpha\cup\beta\rangle\phi$, then the case holds because $[N/p]\langle\ell\cdot M\rangle = \langle\ell\cdot[N/p]M\rangle$.

Case $\langle\cup\rangle$I2, postcondition $\langle\alpha\cup\beta\rangle\phi$: By premise, $\Gamma, p:\psi \vdash M:\langle\beta\rangle\phi$. By the IH, $\Gamma \vdash [N/p]M:\langle\beta\rangle\phi$, and by $\langle\cup\rangle$I2 $\Gamma \vdash \langle r\cdot[N/p]M\rangle:\langle\alpha\cup\beta\rangle\phi$, then the case holds because $[N/p]\langle r\cdot M\rangle = \langle r\cdot[N/p]M\rangle$.

Case $\langle\cup\rangle$E, postcondition ϕ, term $\langle \text{case } A \text{ of } \ell \Rightarrow \ B \mid r \Rightarrow \ C\rangle$: By premises, $\mathcal{D}_1$ is $\Gamma, p:\psi \vdash A:\langle\alpha\cup\beta\rangle\rho$, $\mathcal{D}_2$ is $\Gamma, p:\psi, \ell:\langle\alpha\rangle\rho \vdash B:\phi$, and $\mathcal{D}_3$ is $\Gamma, p:\psi, r:\langle\beta\rangle\rho \vdash C:\phi$. By weakening, $\Gamma, \ell:\langle\alpha\rangle\rho \vdash N:\psi$ and $\Gamma, r:\langle\beta\rangle\rho \vdash N:\psi$. By the IH then $\Gamma \vdash [N/p]A:\langle\alpha\cup\beta\rangle\rho$ and $\Gamma, \ell:\langle\alpha\rangle\rho \vdash [N/p]B:\phi$ and $\Gamma, r:\langle\beta\rangle\rho \vdash [N/p]C:\phi$. Then by $\langle\cup\rangle$E, have $\Gamma \vdash \langle \text{case } [N/p]A \text{ of } \ell \Rightarrow \ [N/p]B \mid r \Rightarrow \ [N/p]C\rangle : \psi$, then since $p \notin \{\ell, r\}$ (by α-varying it if necessary) have $[N/p]\langle \text{case } A \text{ of } \ell \Rightarrow \ B \mid r \Rightarrow \ C\rangle = \langle \text{case } [N/p]A \text{ of } \ell \Rightarrow \ [N/p]B \mid r \Rightarrow \ [N/p]C\rangle$ which completes the case.

Case $\langle?\rangle$I, postcondition $\langle?\rho\rangle\phi$, term $\langle A, B\rangle$: By premises have $\mathcal{D}_1$ that $\Gamma, p:\psi \vdash A:\rho$ and $\mathcal{D}_2$ that $\Gamma, p:\psi \vdash B:\phi$. By the IH have $\Gamma \vdash [N/p]A:\rho$ and $\Gamma \vdash [N/p]B:\phi$. By $\langle?\rangle$I have $\Gamma \vdash \langle[N/p]A, [N/p]B\rangle : \langle?\rho\rangle\phi$. Then since $[N/p]\langle A, B\rangle$ equals $\langle[N/p]A, [N/p]B\rangle$ this completes the case.

Case $\langle?\rangle$E1, postcondition ρ, term $\langle\pi_L M\rangle$: By premise, have $\Gamma, p:\psi \vdash M:\langle?\rho\rangle\phi$. By the IH, have $\Gamma \vdash [N/p]M:\langle?\rho\rangle\phi$, then by $\langle?\rangle$E1 have $\Gamma \vdash \langle\pi_L([N/p]M)\rangle:\rho$, then $[N/p](\langle\pi_L M\rangle) = \langle\pi_L([N/p]M)\rangle$ which completes the case.

Case $\langle?\rangle$E2, postcondition ϕ, term $\langle\pi_R M\rangle$: By premise, have $\Gamma, p:\psi \vdash M:\langle?\rho\rangle\phi$. By the IH, have $\Gamma \vdash [N/p]M:\langle?\rho\rangle\phi$, then by $\langle?\rangle$E2 have $\Gamma \vdash \langle\pi_R([N/p]M)\rangle:\phi$, then $[N/p](\langle\pi_R M\rangle) = \langle\pi_R([N/p]M)\rangle$ which completes the case.

Case $\langle\!\langle;\rangle\!\rangle$I: By premise, have $\Gamma, p:\psi \vdash M:\langle\!\langle\alpha\rangle\!\rangle\langle\!\langle\beta\rangle\!\rangle\phi$. By the IH, have $\Gamma \vdash [N/p]M:\langle\!\langle\alpha\rangle\!\rangle\langle\!\langle\beta\rangle\!\rangle\phi$, then by $\langle\!\langle;\rangle\!\rangle$I have $\Gamma \vdash \langle\!\langle\iota \ [N/p]M\rangle\!\rangle : \langle\!\langle\alpha;\beta\rangle\!\rangle\phi$, then $[N/p]\langle\!\langle\iota \ M\rangle\!\rangle = \langle\!\langle\iota \ ([N/p]M)\rangle\!\rangle$ which completes the case. The case works for both box and diamond proofs.

Case $\langle\!\langle^d\rangle\!\rangle$I: By premise, have $\Gamma, p:\psi \vdash M:[\![\alpha]\!]\phi$. By the IH, have $\Gamma \vdash [N/p]M:[\![\alpha]\!]\phi$, then by $\langle\!\langle^d\rangle\!\rangle$I have $\Gamma \vdash \langle\!\langle\text{yield } [N/p]M\rangle\!\rangle : \langle\!\langle\alpha^d\rangle\!\rangle\phi$, then $[N/p]\langle\!\langle\text{yield } M\rangle\!\rangle = \langle\!\langle\text{yield } ([N/p]M)\rangle\!\rangle$ which completes the case. The case works for both box and diamond proofs.

Case $\langle\!\langle:=\rangle\!\rangle$I (box): In this case to avoid confusion we say the substituted proof variable was t, not p. By premise, have $\Gamma\frac{y}{x}, t:\psi\frac{y}{x}, p:(x = f\frac{y}{x}) \vdash M:\phi$. Now regroup the context as $(\Gamma\frac{y}{x}, p:(x = f\frac{y}{x}), t:\psi\frac{y}{x})$. By Lemma 4.10 on N and weakening have $\Gamma\frac{y}{x}, p:(x = f\frac{y}{x}) \vdash N\frac{y}{x}:\psi\frac{y}{x}$. Then we can apply the IH on M while varying the context to $\Gamma\frac{y}{x}, p:(x = f\frac{y}{x})$ and varying N to equally-complex $N\frac{y}{x}$, which yields $\Gamma\frac{y}{x}, p:(x = f\frac{y}{x}) \vdash ([(N\frac{y}{x})/t]M):\phi$, then by $\langle\!\langle:=\rangle\!\rangle$I have $\Gamma \vdash [x := f\frac{y}{x} \text{ in } p.\ ([(N\frac{y}{x})/t]M)]:[x := f]\phi$. Note that to avoid capture, this case of substitution renames, i.e., $[N/t][x := f\frac{y}{x} \text{ in } p.\ M] = [x := f\frac{y}{x} \text{ in } p.\ [N\frac{y}{x}/t]M]$.

Case $[:*]$I, here call the substituted variable t: By premise, have $(\Gamma, t:\psi)\frac{y}{x} \vdash M:\phi$. Regroup the context as $(\Gamma\frac{y}{x}), t:\psi\frac{y}{x}$. By Lemma 4.10 on N have $\Gamma\frac{y}{x} \vdash N\frac{y}{x}:\phi\frac{y}{x}$. Then apply IH for M and varying to equally-complex $\Gamma\frac{y}{x}$ and $N\frac{y}{x}$, yielding $\Gamma\frac{y}{x} \vdash [N\frac{y}{x}/t]M:\phi$, then by $[:*]$I have $\Gamma \vdash (\lambda x:\mathbb{Q}.\ [N\frac{y}{x}/t]M):[x := *]\phi$. Note that to avoid capture, this case of substitution renames, i.e., $[N/t](\lambda x:\mathbb{Q}.\ M) = \lambda x:\mathbb{Q}.\ ([(N\frac{y}{x})/t]M)$.

Case [:∗]E: By premise, have $\Gamma, t : \psi \vdash M : [x := *]\phi$. By the IH, have $\Gamma \vdash [N/t]M : [x := *]\phi$. By [:∗]E have $\Gamma \vdash ([N/t]M)\ f : \phi^f_x$, then the case holds as a result of the equality $[N/t](M\ f) = ([N/t]M)\ f$.

Case [∗]I: The loop invariant is usually called ψ, but here we use $\tilde{\psi}$ to avoid colliding with t. By premises, have $\mathcal{D}_1$ is $\Gamma, t : \psi \vdash A : \tilde{\psi}$ and $\mathcal{D}_2$ is $q : \tilde{\psi} \vdash B : [\alpha]\tilde{\psi}$ and $\mathcal{D}_3$ is $q : \tilde{\psi} \vdash C : \phi$. By the IH on $\mathcal{D}_1$ have $\Gamma \vdash [N/t]A : \tilde{\psi}$, then applying [∗]I without changing $\mathcal{D}_2$ or $\mathcal{D}_3$, have $\Gamma \vdash ([N/t]A)$ rep $q : \tilde{\psi}.\ B$ in $C : [\alpha^*]\phi$. This concludes the case since $[N/t](A \text{ rep } q : \tilde{\psi}.\ B \text{ in } C) = ([N/t]A) \text{ rep } q : \tilde{\psi}.\ B \text{ in } C$, i.e., no substitution occurs in B or C since they cannot access t anyway.

Case [∗]R, postcondition $[\alpha^*]\phi$: By premise, $\Gamma, p : \psi \vdash M : \phi \wedge [\alpha][\alpha^*]\phi$. By the IH, $\Gamma \vdash [N/p]M : \phi \wedge [\alpha][\alpha^*]\phi$ and by [∗]R, $\Gamma \vdash [\text{roll } [N/p]M] : [\alpha^*]\phi$, then the case holds because $[N/p][\text{roll } M] = [\text{roll } [N/p]M]$.

Case [∗]E, postcondition $\phi \wedge [\alpha][\alpha^*]\phi$: By premise, $\Gamma, p : \psi \vdash M : [\alpha^*]\phi$. By the IH, $\Gamma \vdash [N/p]M : [\alpha^*]\phi$ and by [∗]E, $\Gamma \vdash [\text{unroll } [N/p]M] : \phi \wedge [\alpha][\alpha^*]\phi$, then the case holds because $[N/p][\text{unroll } M] = [\text{unroll } [N/p]M]$.

Case [∪]I, postcondition $[\alpha \cup \beta]\phi$, term $[A, B]$: By premises have $\mathcal{D}_1$ that $\Gamma, p : \psi \vdash A : [\alpha]\phi$ and $\mathcal{D}_2$ that $\Gamma, p : \psi \vdash B : [\beta]\phi$. By the IH have $\Gamma \vdash [N/p]A : [\alpha]\phi$ and $\Gamma \vdash [N/p]B : [\beta]\phi$. By [∪]I have $\Gamma \vdash [[N/p]A, [N/p]B] : [\alpha \cup \beta]\phi$. Then since $[N/p][A, B] = [[N/p]A, [N/p]B]$ this completes the case.

Case [∪]E1, postcondition $[\alpha]\phi$, term $[\pi_L M]$: By premise, have $\Gamma, p : \psi \vdash M : [\alpha \cup \beta]\phi$. By the IH, have $\Gamma \vdash [N/p]M : [\alpha \cup \beta]\phi$, then by [∪]E1 have $\Gamma \vdash [\pi_L([N/p]M)] : [\alpha]\phi$, then $[N/p][\pi_L M] = [\pi_L([N/p]M)]$ which completes the case.

Case [∪]E2, postcondition $[\beta]\phi$, term $[\pi_R M]$: By premise, have $\Gamma, p : \psi \vdash M : [\alpha \cup \beta]\phi$. By the IH, have $\Gamma \vdash [N/p]M : [\alpha \cup \beta]\phi$, then by [∪]E2 have $\Gamma \vdash [\pi_R([N/p]M)] : [\beta]\phi$, then $[N/p][\pi_R M] = [\pi_R([N/p]M)]$ which completes the case.

Case [?]I, postcondition $[?\rho]\phi$, term $(\lambda q : \rho.\ M)$: By premise, have $\Gamma, p : \psi, q : \rho \vdash M : \phi$. By weakening, have $\Gamma, q : \rho \vdash N : \psi$. By the IH, have $\Gamma, q : \rho \vdash M : \phi$ so by [?]I have $\lambda q : \rho.\ ([N/p]M)$. Then (after α-varying q to avoid capture if needed) $[N/p](\lambda q : \rho.\ M) = \lambda q : \rho.\ ([N/p]M)$ as desired.

Case [?]E, postcondition ϕ, term $(A\ B)$: By premise, have $\Gamma, p : \psi \vdash A : [?\rho]\phi$ and $\Gamma, p : \psi \vdash B : \rho$. By the IH, have $\Gamma \vdash [N/p]A : [?\rho]\phi$ and $\Gamma \vdash [N/p]B : \rho$, then $[N/p](A\ B) = [N/p]A\ [N/p]B$ as desired.

Case M, postcondition $\langle\alpha\rangle\rho$, term $A \circ_q B$: Let $\vec{x} = \text{BV}(\alpha)$ and $\vec{y}$ be a vector of fresh variables of the same length. By $\mathcal{D}_1$ have $\Gamma, p : \psi \vdash A : \langle\alpha\rangle\phi$ and by $\mathcal{D}_2$ have $(\Gamma, p : \psi)^{\vec{y}}_{\vec{x}}, q : \phi \vdash B : \rho$. Then by the IH on $\mathcal{D}_1$, have (1) $\Gamma \vdash [N/p]A : \langle\alpha\rangle\phi$. By Lemma 4.10 ($|\vec{x}|$ times) on N and weakening, have $\Gamma^{\vec{y}}_{\vec{x}}, q : \phi \vdash N^{\vec{y}}_{\vec{x}} : \phi^{\vec{y}}_{\vec{x}}$, then because $(((\Gamma, p : \psi)^{\vec{y}}_{\vec{x}}), q : \phi = ((\Gamma)^{\vec{y}}_{\vec{x}}, q : \phi), p : \psi^{\vec{y}}_{\vec{x}})$, we can apply the IH on $\mathcal{D}_2$, giving (2) $(\Gamma)^{\vec{y}}_{\vec{x}}, q : \phi \vdash [N^{\vec{y}}_{\vec{x}}/p]B : \rho$. Then apply monotonicity rule M to (1) and (2), giving $\Gamma \vdash ([N/p]A) \circ_q [N^{\vec{y}}_{\vec{x}}/p]B : \langle\alpha\rangle\rho$. Substitution avoids capture in the B branch, so $[N/p](A \circ_q B) = ([N/p]A) \circ_q [N^{\vec{y}}_{\vec{x}}/p]B$, so the case holds.

Case hyp, postcondition ϕ: That is, the proof term is some variable q. If $q = p$, then $[N/p]q = N$ and $\Gamma \vdash N : \psi$ by assumption. Moreover $\phi = \psi$ since $\Gamma, p : \phi = \Gamma, p : \psi$ assigns only one type to p. Thus, trivially $\Gamma \vdash [N/p]q : \phi$ as desired. Else $q \neq p$, so $[N/p]q = q$ and $q \in \Gamma$, so that $\Gamma \vdash q : \phi$ by hyp and thus $\Gamma \vdash [N/p]q : \phi$ as well. □

Lemma 4.17 (Preservation)**.** *Let $\mapsto^*$ be the reflexive, transitive closure of the $\mapsto$ relation. If $\cdot \vdash M : \phi$ and $M \mapsto^* M'$, then $\cdot \vdash M' : \phi$.*

Proof. By outer induction on the derivation $M \mapsto^* M'$, then inner induction on $M \mapsto M'$. The base case of the outer induction is trivial, so preservation reduces to the inner induction which shows that if $\cdot \vdash M : \phi$ and $M \mapsto M'$ then $\cdot \vdash M' : \phi$.

We split the cases of the proof according to the four main kinds of rules: Beta rules, structural rules, monotonicity-conversion rules, and commuting-conversion rules. While the cases within each class are not necessarily symmetric, they usually share a similar proof approach.

Beta rule cases:

Case $\lambda\phi\beta$, rule $(\lambda p : \psi.\ M)\ N \mapsto [N/p]M$, postcondition ϕ: By inversion, $\cdot \vdash M : [?\psi]\phi$ and $\cdot \vdash N : \psi$. Then by Lemma 4.14, have $\cdot \vdash [N/p]M : \phi$ as desired.

Case $\lambda\beta$, rule $(\lambda x : \mathbb{Q}.\ M)\ f \mapsto M^f_x$, postcondition ϕ^f_x: By inversion $\cdot \frac{y}{x} \vdash M : \phi$, Then by Lemma A.9 have $\cdot \frac{y}{x} \frac{f}{x} \vdash M^f_x : \phi^f_x$. Then note $\cdot \frac{y}{x} \frac{f}{x} = \cdot \frac{y}{x} = \cdot$ so that $\cdot \frac{y}{x} \frac{f}{x} \vdash M^f_x : \phi^f_x$ simplifies to $\cdot \vdash M^f_x : \phi^f_x$.

Case $\pi_L\beta$, rule $\langle\!\langle \pi_L \langle\!\langle A, B \rangle\!\rangle \rangle\!\rangle \mapsto A$, postcondition ϕ: By inversion twice, $\cdot \vdash A : \phi$, which completes the case.

Case $\pi_R\beta$: By inversion twice, $\cdot \vdash B : \phi$ as desired.

Case caseβL, rule $\langle \text{case } \langle \ell \cdot A \rangle \text{ of } p \Rightarrow\ B \mid q \Rightarrow\ C \rangle \mapsto [A/p]B$, postcondition ϕ: We give the case for choices, the case for Angelic loops is symmetric. By inversion $\cdot \vdash A : \langle\alpha\rangle\psi$, for some α and ψ. By inversion, $p : \langle\alpha\rangle\psi \vdash B : \phi$, then by Lemma 4.14 $\cdot \vdash [A/p]B : \phi$, which completes the case.

Case caseβR, rule $\langle \text{case } \langle r \cdot A \rangle \text{ of } p \Rightarrow\ B \mid q \Rightarrow\ C \rangle \mapsto [A/q]C$, postcondition ϕ: We give the case for choices, the case for Angelic loops is symmetric. By inversion $\cdot \vdash A : \langle\beta\rangle\psi$, for some β, ψ. By inversion, $q : \langle\beta\rangle\psi \vdash C : \phi$, then by Lemma 4.14 $\cdot \vdash [A/q]C : \phi$ as desired.

Case unpackβ, rule $\text{unpack}(\langle f\frac{y}{x} :* q.\ M \rangle, py.\ N) \mapsto (\text{Ghost}[x = f\frac{y}{x}](q.\ [M/p]N))$, postcondition ϕ:

By inversion, $(q : x = f\frac{y}{x}) \vdash M : \psi$ and $\cdot \frac{z}{x}, p : \psi \vdash N : \phi$ which simplifies to $p : \psi \vdash N : \phi$. Then by Lemma 4.14 have (1) $(q : x = f\frac{y}{x}) \vdash [M/p]N : \phi$. Then rule iG is applicable because x is fresh in both $f\frac{y}{x}$ (by definition) and ϕ (by side condition). By iG have $\cdot \vdash (\text{Ghost}[x = f\frac{y}{x}](q.\ [M/p]N)) : \phi$ as desired.

Case FPβ, $FP(A, s.\ B, g.\ C) \mapsto \langle \text{case}_*\ A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow [r \circ_t FP(t, s.\ B, g.\ C)/g]C \rangle$, postcondition ψ: By inversion, $\mathcal{D}_1$ is $\cdot \vdash A : \langle\alpha^*\rangle\phi$, $\mathcal{D}_2$ is $(1) s : \phi \vdash B : \psi$, and $\mathcal{D}_3$ is $g : \langle\alpha\rangle\psi \vdash C : \psi$. On the other branch, note $t : \langle\alpha^*\rangle\phi \vdash FP(t, s.\ B, g.\ C) : \psi$, by FP then by hyp and by $\mathcal{D}_2$ and $\mathcal{D}_3$. Then by monotonicity rule M, have $r : \langle\alpha\rangle\langle\alpha^*\rangle\phi \vdash r \circ_t FP(t, s.\ B, g.\ C) : \langle\alpha\rangle\psi$ by the previous line and by weakening and hyp. Next, apply Lemma 4.14 to $\mathcal{D}_3$, getting (2) $r : \langle\alpha\rangle\langle\alpha^*\rangle\phi \vdash [r \circ_t FP(t, s.\ B, g.\ C)/g]C : \psi$. Now apply rule $\langle*\rangle$C to facts (1) and (2), which yields $\cdot \vdash \langle \text{case}_*\ A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow [r \circ_t FP(t, s.\ B, g.\ C)/g]C \rangle : \psi$ as desired.

Case repβ, rule $(M \text{ rep } p : \psi.\ N \text{ in } O) \mapsto [\text{roll } \langle M, ([M/p]N) \circ_q (q \text{ rep } p : \psi.\ N \text{ in } O) \rangle]$: By inversion have $\mathcal{D}_1$ is $\cdot \vdash M : \psi$ and $\mathcal{D}_2$ is $p : \psi \vdash N : [\alpha]\psi$ and $\mathcal{D}_3$ is $p : \psi \vdash O : \phi$.

Note (1) $q : \psi \vdash (q \text{ rep } p : \psi.\ N \text{ in } O) : [\alpha^*]\phi$ by [*]I since $q : \psi \vdash q : \psi$ by hyp and $p : \psi \vdash N : [\alpha]\psi$ by $\mathcal{D}_2$. Then (2) $\cdot \vdash [M/p]N : [\alpha]\psi$ by Lemma 4.14 and $\mathcal{D}_1$ and $\mathcal{D}_2$ and weakening. Then monotonicity rule M on (1) and (2) gives $\cdot \vdash ([M/p]N) \circ_q (q \text{ rep } p : \psi.\ N \text{ in } O)$

$: [\alpha][\alpha^*]\phi$ Then by $\langle ? \rangle$I have $\cdot \vdash \langle M, ([M/p]N) \circ_q (q \text{ rep } p : \psi.\ N \text{ in } O) \rangle : \phi \wedge [\alpha][\alpha^*]\phi$ so by $[*]$R have $\cdot \vdash [\text{roll } \langle M, ([M/p]N) \circ_q (q \text{ rep } p : \psi.\ N \text{ in } O) \rangle] : [\alpha^*]\phi$ as desired.

Case unrollβ, rule $[\text{unroll } [\text{roll } M]] \mapsto M$: By inversion twice, $\cdot \vdash M : \phi \wedge [\alpha][\alpha^*]\phi$, which completes the case.

Case caseβL, rule $\langle \text{case } \langle \ell \cdot A \rangle \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C \rangle \mapsto [A/\ell]B$: By inversion, $\cdot \vdash A : \phi$ and by substitution $\cdot \vdash [A/\ell]B : \psi$ as desired. The case for diamond loops is symmetric.

Case caseβR, rule $\langle \text{case } \langle r \cdot A \rangle \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C \rangle \mapsto [A/r]C$: By inversion, $\cdot \vdash A : \langle \alpha \rangle \langle \alpha^* \rangle \phi$ and by substitution $\cdot \vdash [A/r]C : \psi$ as desired. The case for diamond loops holds by symmetry.

Case forβ, rule

$$\begin{array}{l}
\text{for}(p : \varphi(\mathcal{M}) = A; q; B)\ \{\alpha\}\ C \mapsto \\
\langle \text{case split } [\mathcal{M} \sim 0] \text{ of} \\
\quad \ell \Rightarrow \langle \text{stop } [(A, \ell)/(p, q)]C \rangle \\
\quad \mid r \Rightarrow \text{Ghost}[\mathcal{M}_0 = \mathcal{M}](rr.\ \langle \text{go } (([A, \langle rr, r \rangle / p, q]B) \circ_t (\text{for}(p : \varphi(\mathcal{M}) = \langle \pi_L t \rangle; q; B)\ \{\alpha\}\ C)) \rangle) \rangle
\end{array}$$

By inversion, $\mathcal{D}_1$ is $\cdot \vdash A : \varphi$ and $\mathcal{D}_2$ is $p : \varphi, q : (\mathcal{M}_0 = \mathcal{M} \succ \mathbf{0}) \vdash B : \langle \alpha \rangle (\varphi \wedge \mathcal{M}_0 \succ \mathcal{M})$ and $\mathcal{D}_3$ is $p : \varphi, q : (\mathbf{0} \succcurlyeq \mathcal{M}) \vdash C : \phi$. By split, (1) $\cdot \vdash \text{split } [\mathcal{M} \sim \mathbf{0}] : (\mathbf{0} \succcurlyeq \mathcal{M} \vee \mathcal{M} \succ \mathbf{0})$. Consider the first branch: $\ell : \mathbf{0} \succcurlyeq \mathcal{M} \vdash \langle A, \ell \rangle : \varphi \wedge \mathbf{0} \succcurlyeq \mathcal{M}$ by $\langle ? \rangle$I, hyp, and weakening. Then by $\langle * \rangle$S have (left) $\ell : \mathbf{0} \succcurlyeq \mathcal{M} \vdash \langle \text{stop } [(A, \ell)/(p, q)]C \rangle : \langle \alpha^* \rangle \phi$. Consider the second branch: By $\langle ? \rangle$I have (x) $rr : \mathcal{M}_0 = \mathcal{M}, r : \mathcal{M} \succ \mathbf{0} \vdash \langle rr, r \rangle : \mathcal{M}_0 = \mathcal{M} \succ \mathbf{0}$. then by (x) and $\mathcal{D}_2$ and weakening and Lemma 4.14 have (3a) $r : \mathcal{M} \succ \mathbf{0}, rr : \mathcal{M}_0 = \mathcal{M} \vdash [A, \langle rr, r \rangle / p, q]B : \langle \alpha \rangle (\varphi \wedge \mathcal{M}_0 \succ \mathcal{M})$. Separately, (3b) $t : \varphi \wedge \mathcal{M}_0 \succ \mathcal{M} \vdash \text{for}(p : \varphi(\mathcal{M}) = \langle \pi_L t \rangle; q; B)\ \{\alpha\}\ C : \langle \alpha^* \rangle \phi$. Monotonicity rule M on (3a) and (3b) gives $rr : \mathcal{M}_0 = \mathcal{M}, r : \mathcal{M} \succ \mathbf{0} \vdash [A, \langle rr, r \rangle / p, q]B \circ_t \text{for}(p : \varphi(\mathcal{M}) = \langle \pi_L t \rangle; q; B)\ \{\alpha\}\ C : \langle \alpha \rangle \langle \alpha^* \rangle (\phi)$. Thus, in one more step, by applying rule $\langle * \rangle$G, we have

$$rr : \mathcal{M}_0 = \mathcal{M}, r : \mathcal{M} \succ \mathbf{0} \vdash \langle \text{go } [A, \langle rr, r \rangle / p, q]B \circ_t \text{for}(p : \varphi(\mathcal{M}) = \langle \pi_L t \rangle; q; B)\ \{\alpha\}\ C \rangle : \langle \alpha^* \rangle \phi$$

Then by rule iG have (3)

$$\begin{array}{l}
r : \mathcal{M} \succ \mathbf{0} \\
\quad \vdash \text{Ghost}[\mathcal{M}_0 = \mathcal{M}](rr.\ \langle \text{go } [A, \langle rr, r \rangle / p, q]B \circ_t \text{for}(p : \varphi(\mathcal{M}) = \langle \pi_L t \rangle; q; B)\ \{\alpha\}\ C \rangle) \\
\quad : \langle \alpha^* \rangle \phi
\end{array}$$

Lastly, combining (1), (2), and (3), then by $\langle \cup \rangle$E have

$$\begin{array}{l}
\cdot \vdash \langle \text{case split } [\mathcal{M} \sim 0] \text{ of} \\
\quad \ell \Rightarrow\ \langle \text{stop } [(A, \ell)/(p, q)]C \rangle \\
\quad \mid r \Rightarrow\ \text{Ghost}[\mathcal{M}_0 = \mathcal{M}](rr.\ \langle \text{go } [A, \langle rr, r \rangle / p, q]B \circ_t \text{for}(p : \varphi(\mathcal{M}) = \langle \pi_L t \rangle; q; B)\ \{\alpha\}\ C \rangle) \rangle \\
: \langle \alpha^* \rangle \phi
\end{array}$$

as desired.

Structural rules:

We do not write out the premises of the stepping rules explicitly in each case, because they are very similar. For unary operations applied to a term M, as well as the left structural rule of any binary operator, we assume $M \mapsto M'$, or for the right structural rule of any binary operator, we assume M simp for the operator's left operand and $N \mapsto N'$ for the operator's right operand.

Case $[\cup]$S1: By the IH, $\cdot \vdash M' : [\alpha]\phi$, so by case assumption and by rule $[\cup]$I then $\cdot \vdash [M', N] : [\alpha \cup \beta]\phi$.

Case $[\cup]$S2, $[M, N] \mapsto [M, N']$, postcondition $[\alpha \cup \beta]\phi$: By the IH, $\cdot \vdash N' : [\beta]\phi$, so by case assumption and by rule $[\cup]$I then $\cdot \vdash [M, N'] : [\alpha \cup \beta]\phi$.

Case $\langle\cup\rangle$S1 (diamond): By the IH, $\cdot \vdash M' : \phi$, so by case assumption and by rule $\langle ? \rangle$I then $\cdot \vdash \langle M', N\rangle : \langle ?\phi\rangle\psi$.

Case $\langle\cup\rangle$S2 (diamond): By the IH, $\cdot \vdash N' : \psi$, so by case assumption and by rule $\langle ? \rangle$I then $\cdot \vdash \langle M, N'\rangle : \langle ?\phi\rangle\psi$.

Case $\langle\!|\pi_L|\!\rangle$S: By the IH, $\cdot \vdash M' : \langle ?\phi\rangle\psi$ so by rule $\langle ? \rangle$E1 then $\cdot \vdash \langle \pi_L M'\rangle : \phi$. The case for $[\pi_L M']$ is symmetric.

Case $\langle\!|\pi_R|\!\rangle$S: By the IH, $\cdot \vdash M' : \langle ?\phi\rangle\psi$ so by rule $\langle ? \rangle$E2 then $\cdot \vdash \langle \pi_R M'\rangle : \psi$. The case for $[\pi_R M']$ is symmetric.

Case $\langle\!|\ell\cdot|\!\rangle$S: We give the case for Angelic choices, the case for Angelic loops is symmetric. By the IH, $\cdot \vdash M' : \langle\alpha\rangle\phi$ so by rule $\langle\cup\rangle$I1 then $\cdot \vdash \langle \ell \cdot M'\rangle : \langle\alpha \cup \beta\rangle\phi$.

Case $\langle\!|r\cdot|\!\rangle$S: We give the case for Angelic choices, the case for Angelic loops is symmetric. By the IH, $\cdot \vdash M' : \langle\beta\rangle\phi$ so by rule $\langle\cup\rangle$I2 then $\cdot \vdash \langle r \cdot M'\rangle : \langle\alpha \cup \beta\rangle\phi$.

Case repS: By the IH, $\cdot \vdash M' : \psi$ so by rule $[*]$I then $\cdot \vdash (M' \text{ rep } p : \psi.\ N \text{ in } O) : [\alpha^*]\phi$.

Case appS1: By the IH, $\cdot \vdash M' : [?\phi]\psi$ so by rule $[?]$E then $\cdot \vdash M'\ N : \psi$.

Case appS2; By the IH, $\cdot \vdash N' : \phi$ so by rule $[?]$E then $\cdot \vdash M\ N' : \psi$.

Case $[\iota]$S: By the IH, $\cdot \vdash M' : [\alpha][\beta]\phi$ so by rule $\langle\!|;|\!\rangle$I then $\cdot \vdash [\iota\ M'] : [\alpha; \beta]\phi$.

Case $\langle\iota\rangle$S: By the IH, $\cdot \vdash M' : \langle\alpha\rangle\langle\beta\rangle\phi$ so by rule $\langle\!|;|\!\rangle$I then $\cdot \vdash \langle\iota\ M'\rangle : \langle\alpha; \beta\rangle\phi$.

Case $[^d]$S: By the IH, $\cdot \vdash M' : \langle\alpha\rangle\phi$ so by rule $\langle\!|^d|\!\rangle$I then $\cdot \vdash [\text{yield } M'] : [\alpha^d]\phi$.

Case $\langle^d\rangle$S: By the IH, $\cdot \vdash M' : [\alpha]\phi$ so by rule $\langle\!|^d|\!\rangle$I then $\cdot \vdash \langle\text{yield } M'\rangle : \langle\alpha^d\rangle\phi$.

Case $\circ$S: By the IH, $\cdot \vdash M' : \langle\alpha\rangle\phi$ so by monotonicity rule M then $\cdot \vdash M' \circ_p N : \langle\alpha\rangle\psi$.

Case FPS: By the IH, $\cdot \vdash M' : \langle\alpha^*\rangle\phi$ so by rule FP then $\cdot \vdash FP(M', s.\ N, g.\ O) : \psi$.

Case caseS: By the inductive hypothesis, $\cdot \vdash M' : \langle\alpha \cup \beta\rangle\phi$ so applying rule $\langle\cup\rangle$E results in $\cdot \vdash \langle\text{case } M' \text{ of } \ell \Rightarrow\ N \mid r \Rightarrow\ O\rangle : \psi$.

Case $\langle*\rangle$C (diamond loops): By the IH, $\cdot \vdash M' : \langle\alpha^*\rangle\phi$ so applying rule $\langle*\rangle$C results in

$$\cdot \vdash \langle\text{case}_*\ M' \text{ of } s \Rightarrow\ N \mid g \Rightarrow O\rangle : \psi$$

Case $\langle:*\rangle$S: By the IH, $\cdot \vdash M' : \langle x := *\rangle\phi$ so rule $\langle:*\rangle$E gives $\cdot \vdash \text{unpack}(M', py.\ N) : \psi$.

Case forS: By the inductive hypothesis, have $\cdot \vdash M' : \varphi$ so rule $\langle*\rangle$I results in $\cdot \vdash \text{for}(p : \varphi(\mathcal{M}) = M'; q; N)\ \{\alpha\}\ O : \langle\alpha^*\rangle\phi$.

Case unroll: By the IH, $\cdot \vdash M' : [\alpha^*]\phi$ so by rule $[*]$E then $\cdot \vdash [\text{unroll } M'] : \phi \wedge [\alpha][\alpha^*]\phi$.

Monotonicity conversion rule cases:

In each case, let $\vec{x}$ denote the bound variables of the initial program on which monotonicity is being applied and $\vec{y}$ be a fresh variable vector of the same size. For rules which mention vectors $\vec{a}, \vec{a}', \vec{b}, \vec{b}'$, their definitions are as in the main text. Note that the definitions of vectors $\vec{a}, \vec{a}', \vec{b}, \vec{b}'$ are not identical between all cases.

Case $\lambda\phi\circ$, rule $(\lambda p:\phi.\ M)\circ_q N \mapsto (\lambda p:\phi.\ ([M/q]N))$: Note $\cdot\frac{\vec{y}}{\vec{x}} = \cdot$. By premises, have $\mathcal{D}_1$ is $\cdot \vdash (\lambda p:\phi.\ M) : [?\phi]\psi$ and $\mathcal{D}_2$ is $\cdot\frac{\vec{y}}{\vec{x}}, q:\psi \vdash N:\rho$, i.e., $q:\psi \vdash N:\rho$. By inversion on $\mathcal{D}_1$ have $p:\phi \vdash M:\psi$ then $p:\phi \vdash [M/q]N:\rho$ by Lemma 4.14 and weakening, then by rule $\langle?\rangle$I have $\cdot \vdash (\lambda p:\phi.\ ([M/q]N)) : [?\phi]\rho$.

Case $\lambda\circ$, rule $(\lambda x:\mathbb{Q}.\ M)\circ_q N \mapsto (\lambda x:\mathbb{Q}.\ [M/q]N)$: By premises, have $\mathcal{D}_1$ is $\cdot \vdash (\lambda x:\mathbb{Q}.\ M) : [x:=*]\phi$ and $\mathcal{D}_2$ is $\cdot\frac{\vec{y}}{\vec{x}}, q:\phi \vdash N:\psi$. By inversion on $\mathcal{D}_1$ have $\cdot\frac{\vec{y}}{\vec{x}} \vdash M:\phi$ since $\vec{x} = x$. then $\cdot\frac{\vec{y}}{\vec{x}} \vdash [M/q]N:\rho$ by Lemma 4.14 and weakening, then by rule $\langle:*\rangle$I have $\cdot \vdash (\lambda x:\mathbb{Q}.\ ([M/q]N)) : [x:=*]\psi$.

Case $[\cup]\circ$, rule $[A,B]\circ_q N \mapsto [A\circ_p(N^{\vec{a}}_{\vec{a'}}), B\circ_p(N^{\vec{b}}_{\vec{b'}})]$: By premises, have $\mathcal{D}_1$ is $\cdot \vdash [A,B] : [\alpha\cup\beta]\phi$ and $\mathcal{D}_2$ is $\cdot\frac{\vec{y}}{\vec{x}}, p:\phi \vdash N:\psi$ where $\vec{x}$ is $\mathrm{BV}(\alpha)\cup\mathrm{BV}(\beta)$ as a vector and $\vec{y}$ is the corresponding ghost variable vector. Let $\vec{A} = \mathrm{BV}(\alpha)$ and $\vec{B} = \mathrm{BV}(\beta)$ as vectors and let $\vec{A'}$ and $\vec{B'}$ be the corresponding ghosts. Let $\vec{a},\vec{b},\vec{a'}$, and $\vec{b'}$ have their usual meanings such that, for example, $\vec{a}$ contains a subset of variables of $\vec{A}$.

By inversion on $\mathcal{D}_1$ have (1) $\cdot \vdash A:[\alpha]\phi$ and (2) $\cdot \vdash B:[\beta]\phi$. By Lemma 4.13 on $\mathcal{D}_2$ have $\cdot\frac{\vec{y}\,\vec{a'}}{\vec{x}\,\vec{a}}, p:\phi^{\vec{a'}}_{\vec{a}} \vdash N^{\vec{a}}_{\vec{a'}}:\psi$ which by construction of $\vec{a}$ simplifies to $\cdot\frac{\vec{A'}}{\vec{A}}, p:\phi^{\vec{a'}}_{\vec{a}} \vdash N^{\vec{a}}_{\vec{a'}}:\psi$ which then, since $\vec{a'}$ are fresh in ψ and ϕ, simplifies to (3a) $\cdot\frac{\vec{A'}}{\vec{A}}, p:\phi \vdash N:\psi$. By Lemma 4.13 on $\mathcal{D}_2$ again have $\cdot\frac{\vec{y}\,\vec{b'}}{\vec{x}\,\vec{b}}, p:\phi^{\vec{b'}}_{\vec{b}} \vdash N^{\vec{b}}_{\vec{b'}}:\psi$ which by construction of $\vec{b}$ simplifies to $\cdot\frac{\vec{B'}}{\vec{B}}, p:\phi^{\vec{b'}}_{\vec{b}} \vdash N^{\vec{b}}_{\vec{b'}}:\psi$ which then, since $\vec{b'}$ are fresh in ψ and ϕ, simplifies to (3b) $\cdot\frac{\vec{B'}}{\vec{B}}, p:\phi \vdash N:\psi$.

By monotonicity rule M on (1) and (3a) have $\cdot \vdash A\circ_p N^{\vec{a}}_{\vec{a'}} : [\alpha]\psi$ and by monotonicity rule M on (2) and (3b) $\mathcal{D}_2$ have $\cdot \vdash B\circ_p N^{\vec{b}}_{\vec{b'}} : [\beta]\psi$, lastly apply rule $[\cup]$I in order to conclude $\cdot \vdash [A\circ_p N^{\vec{a}}_{\vec{a'}}, B\circ_p N^{\vec{b}}_{\vec{b'}}] : [\alpha\cup\beta]\psi$.

Case $\langle\cup\rangle\circ$, rule $\langle A,B\rangle\circ_p N \mapsto \langle A,[B/p]N\rangle$: Note $(\cdot\frac{\vec{y}}{\vec{x}}) = (\cdot)$. By premises, have $\mathcal{D}_1$ is $\cdot \vdash \langle A,B\rangle : \langle?\phi\rangle\psi$ and $\mathcal{D}_2$ is $p:\psi \vdash N:\rho$. By inversion on $\mathcal{D}_1$ have (1) $\cdot \vdash A:\phi$ and (2) $\cdot \vdash B:\psi$. By Lemma 4.14, have $\cdot \vdash [B/p]N:\rho$. By rule $\langle?\rangle$I have $\cdot \vdash \langle A,[B/p]N\rangle : \langle?\phi\rangle\rho$.

Case $\langle:*\rangle\circ$: By premises, have $\mathcal{D}_1$ is $\cdot \vdash \langle f\frac{y}{x} :* p.\ M\rangle : \langle x:=*\rangle\phi$, and $\mathcal{D}_2$ is $\cdot\frac{\vec{y}}{\vec{x}}, q:\phi \vdash N:\psi$. then by inversion on $\mathcal{D}_1$ have (1) $\cdot\frac{\vec{y}}{\vec{x}}, p:(x = f\frac{y}{x}) \vdash M:\phi$. By weakening on $\mathcal{D}_2$ have $\cdot\frac{\vec{y}}{\vec{x}}, q:\phi, p:(x = f\frac{y}{x}) \vdash N:\psi$, then by Lemma 4.14 have $\cdot\frac{\vec{y}}{\vec{x}}, p:(x = f\frac{y}{x}) \vdash [M/q]N:\psi$, and lastly by rule $\langle:*\rangle$I have $\cdot \vdash \langle f\frac{y}{x} :* p.\ [M/q]N\rangle : \langle x:=*\rangle\psi$.

Case $\langle^d\rangle\circ$: By premises, have $\mathcal{D}_1$ is $\cdot \vdash \langle\text{yield } M\rangle : \langle\alpha^d\rangle\phi$, and $\mathcal{D}_2$ is $\cdot\frac{\vec{y}}{\vec{x}}, q:\phi \vdash N:\psi$. By inversion on $\mathcal{D}_1$ have (1) $\cdot \vdash M:[\alpha]\phi$. By monotonicity rule M have $\cdot \vdash M\circ_p N : [\alpha]\psi$ so by rule $\langle\!\langle^d\rangle\!\rangle$I have $\cdot \vdash \langle\text{yield } (M\circ_p N)\rangle : \langle\alpha^d\rangle\psi$ as desired.

Case $[^d]\circ$: By premises, have $\mathcal{D}_1$ is $\cdot \vdash [\text{yield } M] : [\alpha^d]\phi$, and $\mathcal{D}_2$ is $\cdot\frac{\vec{y}}{\vec{x}}, q:\phi \vdash N:\psi$. By inversion on $\mathcal{D}_1$ have (1) $\cdot \vdash M:\langle\alpha\rangle\phi$. By monotonicity rule M have $\cdot \vdash M\circ_p N : \langle\alpha\rangle\psi$ so by rule $\langle\!\langle^d\rangle\!\rangle$I have $\cdot \vdash [\text{yield } (M\circ_p N)] : [\alpha^d]\psi$ as desired.

Case $\langle\ell\cdot\rangle\circ$ (for $\cup$): By premises, have $\mathcal{D}_1$ is $\cdot \vdash \langle\ell\cdot M\rangle : \langle\alpha\cup\beta\rangle\phi$, then by inversion have (1) $\cdot \vdash M:\langle\alpha\rangle\phi$. By monotonicity rule M have $\cdot \vdash M\circ_p N : \langle\alpha\rangle\psi$ so by rule $\langle\cup\rangle$I1 have $\cdot \vdash \langle\ell\cdot(M\circ_p N)\rangle : \langle\alpha\cup\beta\rangle\psi$ as desired.

Case $\langle r\cdot\rangle\circ$ (for $\cup$): By premises, have $\mathcal{D}_1$ is $\cdot \vdash \langle r\cdot M\rangle : \langle\alpha\cup\beta\rangle\phi$, then by inversion have (1) $\cdot \vdash M:\langle\beta\rangle\phi$. By monotonicity rule M have $\cdot \vdash M\circ_p N : \langle\beta\rangle\psi$ so by rule $\langle\cup\rangle$I2 have $\cdot \vdash \langle r\cdot(M\circ_p N)\rangle : \langle\alpha\cup\beta\rangle\psi$ as desired.

Case $\langle:=\rangle\circ$ (diamond): By premises, have $\mathcal{D}_1$ is $\cdot \vdash \langle x := f\frac{y}{x} \text{ in } p.\ M\rangle : \langle x := f\rangle\phi$, and $\mathcal{D}_2$ is $\cdot\frac{\vec{y}}{\vec{x}}, q : \phi \vdash N : \psi$. then by inversion on $\mathcal{D}_1$ have (1) $\cdot\frac{\vec{y}}{\vec{x}}, p : (x = f\frac{y}{x}) \vdash M : \phi$. By weakening on $\mathcal{D}_2$ have $\cdot\frac{\vec{y}}{\vec{x}}, q : \phi, p : (x = f\frac{y}{x}) \vdash N : \psi$, then by Lemma 4.14 have $\cdot\frac{\vec{y}}{\vec{x}}, p : (x = f\frac{y}{x}) \vdash [M/q]N : \psi$, and lastly apply rule $\langle:*\rangle$I in order to conclude $\cdot \vdash \langle x := f\frac{y}{x} \text{ in } p.\ [M/q]N\rangle : \langle x := f\rangle\psi$.

Case $[:=]\circ$ (box): By premises, have $\mathcal{D}_1$ is $\cdot \vdash [x := f\frac{y}{x} \text{ in } p.\ M] : [x := f]\phi$, and $\mathcal{D}_2$ is $\cdot\frac{\vec{y}}{\vec{x}}, q : \phi \vdash N : \psi$. then by inversion on $\mathcal{D}_1$ have (1) $\cdot\frac{\vec{y}}{\vec{x}}, p : (x = f\frac{y}{x}) \vdash M : \phi$. By weakening on $\mathcal{D}_2$ have $\cdot\frac{\vec{y}}{\vec{x}}, q : \phi, p : (x = f\frac{y}{x}) \vdash N : \psi$, then by Lemma 4.14 have $\cdot\frac{\vec{y}}{\vec{x}}, p : (x = f\frac{y}{x}) \vdash [M/q]N : \psi$, and lastly by rule $\langle:*\rangle$I have $\cdot \vdash [x := f\frac{y}{x} \text{ in } p.\ [M/q]N] : [x := f]\psi$.

Case $[\iota]\circ$: By premises have $\mathcal{D}_1$ is $\cdot \vdash [\iota\ M] : [\alpha;\beta]\phi$, and $\mathcal{D}_2$ is $p : \phi \vdash N : \psi$. By inversion on $\mathcal{D}_1$ have (1) $\cdot \vdash M : [\alpha][\beta]\phi$. By Lemma 4.14 on $\mathcal{D}_2$ have $t : \phi \vdash [t/p]N : \psi$, then by monotonicity rule M have $q : [\beta]\phi \vdash q\circ_t([t/p]N) : [\beta]\psi$, then by monotonicity rule M again have $\cdot \vdash M\circ_q q\circ_t([t/p]N) : [\alpha][\beta]\psi$ and finally by $\langle;\rangle$I have $\cdot \vdash [\iota\ M\circ_q(q\circ_t([t/p]N))] : [\alpha;\beta]\psi$, or equivalently $\cdot \vdash [\iota\ M\circ_q(q\circ_p N)] : [\alpha;\beta]\psi$.

Case $\langle\iota\rangle\circ$: By premises have $\mathcal{D}_1$ is $\cdot \vdash \langle\iota\ M\rangle : \langle\alpha;\beta\rangle\phi$, and $\mathcal{D}_2$ is $p : \phi \vdash N : \psi$. By inversion on $\mathcal{D}_1$ have (1) $\cdot \vdash M : \langle\alpha\rangle\langle\beta\rangle\phi$. By Lemma 4.14 on $\mathcal{D}_2$ have $t : \phi \vdash [t/p]N : \psi$, then by monotonicity rule M have $q : \langle\beta\rangle\phi \vdash q\circ_t([t/p]N) : \langle\beta\rangle\psi$, then by monotonicity rule M again have $\cdot \vdash M\circ_q q\circ_t([t/p]N) : \langle\alpha\rangle\langle\beta\rangle\psi$ and finally by rule $\langle;\rangle$I have $\cdot \vdash \langle\iota\ M\circ_q(q\circ_t([t/p]N))\rangle : \langle\alpha;\beta\rangle\psi$, or equivalently $\cdot \vdash \langle\iota\ M\circ_q(q\circ_p N)\rangle : \langle\alpha;\beta\rangle\psi$.

Case $[*]\circ$: Recall that for this rule we let $\vec{a} = \mathrm{BV}(\alpha)$ as a vector and let $\vec{a}'$ be the corresponding ghost variables. By premises have $\mathcal{D}_1$ is $\cdot \vdash [\text{roll } M] : [\alpha^*]\phi$, and $\mathcal{D}_2$ is $\cdot\frac{\vec{a}'}{\vec{a}}, p : \phi \vdash N : \psi$. Then have (2a) $p : \phi \vdash N^{\vec{a}}_{\vec{a}'} : \psi$ by Lemma 4.13 since $\vec{a}'$ are fresh in ϕ and ψ. By inversion on $\mathcal{D}_1$ have (1) $\cdot \vdash M : \phi \wedge [\alpha][\alpha^*]\phi$. Then by $\langle?\rangle$E1 and $\langle?\rangle$E2 have (1a) $\cdot \vdash \langle\pi_L M\rangle : \phi$ and (1b) $\cdot \vdash \langle\pi_R M\rangle : [\alpha][\alpha^*]\phi$ By monotonicity rule M on (1a) and (2a) have (L) $\cdot \vdash (\langle\pi_L M\rangle)\circ_p N^{\vec{a}}_{\vec{a}'} : \psi$. By monotonicity rule M on $\mathcal{D}_2$ and weakening then have $\cdot\frac{\vec{a}'}{\vec{a}}, t : [\alpha^*]\phi \vdash t\circ_p N : [\alpha^*]\psi$, then by monotonicity rule M again and (1b) have (R) $\cdot \vdash (\langle\pi_R M\rangle)\circ_p t\circ_p N : [\alpha][\alpha^*]\psi$. Finally, from (L) and (R) have by rule $\langle?\rangle$I, $\cdot \vdash \langle(\langle\pi_L M\rangle)\circ_p(N^{\vec{a}}_{\vec{a}'}), (\langle\pi_R M\rangle)\circ_t t\circ_p N\rangle : \psi \wedge [\alpha][\alpha^*]\psi$ and immediately by rule $[*]$R $\cdot \vdash [\text{roll } \langle(\langle\pi_L M\rangle)\circ_p(N^{\vec{a}}_{\vec{a}'}), (\langle\pi_R M\rangle)\circ_t t\circ_p N\rangle] : [\alpha^*]\psi$.

Case stop$\circ$: Recall that in rule stop$\circ$ we let $\vec{a} = \mathrm{BV}(\alpha)$ as a vector and let $\vec{a}'$ be the corresponding ghost variables. By premises have $\mathcal{D}_1$ is $\cdot \vdash \langle\text{stop } M\rangle : \langle\alpha^*\rangle\phi$, and $\mathcal{D}_2$ is $\cdot\frac{\vec{a}'}{\vec{a}}, p : \phi \vdash N : \psi$. By inversion on $\mathcal{D}_1$ have (1) $\cdot \vdash M : \phi$. By Lemma 4.13 on $\mathcal{D}_2$ have $p : \phi \vdash N^{\vec{a}}_{\vec{a}'} : \psi$ after simplifying using the fact that $\vec{a}'$ are fresh in ϕ and ψ. Then by Lemma 4.14 then have $\cdot \vdash [M/p](N^{\vec{a}}_{\vec{a}'}) : \psi$ so by rule $\langle*\rangle$S have $\cdot \vdash \langle\text{stop } [M/p](N^{\vec{a}}_{\vec{a}'})\rangle : \langle\alpha^*\rangle\psi$ as desired.

Case go$\circ$: Recall that in rule go$\circ$ we let $\vec{a} = \mathrm{BV}(\alpha)$ as a vector and let $\vec{a}'$ be the corresponding ghost variables. By premises have $\mathcal{D}_1$ is $\cdot \vdash \langle\text{go } M\rangle : \langle\alpha^*\rangle\phi$, and $\mathcal{D}_2$ is $\cdot\frac{\vec{a}'}{\vec{a}}, p : \phi \vdash N : \psi$. By inversion on $\mathcal{D}_1$ have (1) $\cdot \vdash M : \langle\alpha\rangle\langle\alpha^*\rangle\phi$. By monotonicity rule M have $\cdot\frac{\vec{a}'}{\vec{a}}, t : \langle\alpha^*\rangle\phi \vdash t\circ_q N : \langle\alpha\rangle\psi$. Note this application checks even with $\cdot\frac{\vec{a}'}{\vec{a}}$ because in checking N we can exploit $\cdot\frac{\vec{a}'}{\vec{a}}\frac{\vec{z}}{\vec{a}} = \cdot\frac{\vec{a}'}{\vec{a}}$, where $\vec{z}$ is the additional vector of fresh variables used in the second application. Then by monotonicity rule M again have $q : \langle\alpha\rangle\langle\alpha^*\rangle\phi \vdash q\circ_t t\circ_p N : \langle\alpha\rangle\langle\alpha^*\rangle\psi$. Then by Lemma 4.14 and (1)have $\cdot \vdash [M/q](q\circ_t t\circ_p N) : \langle\alpha\rangle\langle\alpha^*\rangle\psi$ so by rule $\langle*\rangle$G have $\cdot \vdash \langle\text{go } [M/q](q\circ_t t\circ_p N)\rangle : \langle\alpha^*\rangle\psi$ which simplifies to $\cdot \vdash \langle\text{go } M\circ_t t\circ_p N\rangle : \langle\alpha^*\rangle\psi$ as desired.

Commuting conversion rule cases:

Case $\langle\iota\rangle$C: From premises, have $\cdot \vdash \langle\iota\ \langle\text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rangle : \langle\alpha;\beta\rangle\phi$ and by inversion (1) $\cdot \vdash \langle\text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : \langle\alpha\rangle\langle\beta\rangle\phi$, (A) $\cdot \vdash A : \psi \vee \rho$, (B) $\ell : \psi \vdash B : \langle\alpha\rangle\langle\beta\rangle\phi$, and (C) $r : \rho \vdash C : \langle\alpha\rangle\langle\beta\rangle\phi$. Then by rule $\langle[;]\rangle$I have (B1) $\ell : \psi \vdash \langle\iota\ B\rangle : \langle\alpha;\beta\rangle\phi$ and (C1) $r : \rho \vdash \langle\iota\ C\rangle : \langle\alpha;\beta\rangle\phi$, then by rule $\langle\cup\rangle$E on (A) have $\cdot \vdash \langle\text{case } A \text{ of } \ell \Rightarrow\ \langle\iota\ B\rangle \mid r \Rightarrow\ \langle\iota\ C\rangle\rangle : \langle\alpha;\beta\rangle\phi$.

Case $[\iota]$C: From premises, have $\cdot \vdash [\iota\ \langle\text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle] : [\alpha;\beta]\phi$ and by inversion (1) $\cdot \vdash \langle\text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : [\alpha][\beta]\phi$, (A) $\cdot \vdash A : \psi \vee \rho$, (B) $\ell : \psi \vdash B : [\alpha][\beta]\phi$, and (C) $r : \rho \vdash C : [\alpha][\beta]\phi$. Then by rule $\langle[;]\rangle$I have (B1) $\ell : \psi \vdash \langle\iota\ B\rangle : [\alpha;\beta]\phi$ and (C1) $r : \rho \vdash \langle\iota\ C\rangle : [\alpha;\beta]\phi$, then by rule $\langle\cup\rangle$E on (A) have $\cdot \vdash \langle\text{case } A \text{ of } \ell \Rightarrow\ [\iota\ B] \mid r \Rightarrow\ [\iota\ C]\rangle : [\alpha;\beta]\phi$.

Case $[\cup]$C1: From premises, have $\cdot \vdash [\langle\text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle, N] : [\alpha \cup \beta]\phi$ and by inversion (1) $\cdot \vdash \langle\text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : [\alpha]\phi$, (A) $\cdot \vdash A : \psi \vee \rho$, (B) $\ell : \psi \vdash B : [\alpha]\phi$, and (C) $r : \rho \vdash C : [\alpha]\phi$. Then by rule $[\cup]$I and weakening on N have (B1) $\ell : \psi \vdash [B, N] : [\alpha \cup \beta]\phi$ and (C1) $r : \rho \vdash [C, N] : [\alpha \cup \beta]\phi$, then by rule $\langle\cup\rangle$E on (A) have $\cdot \vdash \langle\text{case } A \text{ of } \ell \Rightarrow\ [B, N] \mid r \Rightarrow\ [C, N]\rangle : [\alpha \cup \beta]\phi$.

Case $[\cup]$C2: From premises, have $\cdot \vdash [M, \langle\text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle] : [\alpha \cup \beta]\phi$ and by inversion (1) $\cdot \vdash \langle\text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : [\beta]\phi$, (A) $\cdot \vdash A : \psi \vee \rho$, (B) $\ell : \psi \vdash B : [\beta]\phi$, and (C) $r : \rho \vdash C : [\beta]\phi$. Then by rule $[\cup]$I and weakening on M have (B1) $\ell : \psi \vdash [M, B] : [\alpha \cup \beta]\phi$ and (C1) $r : \rho \vdash [M, C] : [\alpha \cup \beta]\phi$, then by rule $\langle\cup\rangle$E on (A) have $\cdot \vdash \langle\text{case } A \text{ of } \ell \Rightarrow\ [M, B] \mid r \Rightarrow\ [M, C]\rangle : [\alpha \cup \beta]\phi$.

Case $\langle\cup\rangle$C1: From premises, have $\cdot \vdash \langle\langle\text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle, N\rangle : \langle?\psi\rangle\phi$ and by inversion (1) $\cdot \vdash \langle\text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : \psi$, (A) $\cdot \vdash A : \zeta \vee \rho$, (B) $\ell : \zeta \vdash B : \psi$, and (C) $r : \rho \vdash C : \psi$. Then by rule $\langle?\rangle$I and weakening on N have (B1) $\ell : \zeta \vdash \langle B, N\rangle : \langle?\psi\rangle\phi$ and (C1) $r : \rho \vdash \langle C, N\rangle : \langle?\psi\rangle\phi$, then by rule $\langle\cup\rangle$E on (A) have $\cdot \vdash \langle\text{case } A \text{ of } \ell \Rightarrow\ \langle B, N\rangle \mid r \Rightarrow\ \langle C, N\rangle\rangle : \langle?\psi\rangle\phi$.

Case $\langle\cup\rangle$C2: From premises, haves $\cdot \vdash \langle M, \langle\text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rangle : \langle?\psi\rangle\phi$ and by inversion (1) $\cdot \vdash \langle\text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : \psi$, (A) $\cdot \vdash A : \zeta \vee \rho$, (B) $\ell : \zeta \vdash B : \psi$, and (C) $r : \rho \vdash C : \psi$. Then by rule $\langle?\rangle$I and weakening on M have (B1) $\ell : \zeta \vdash \langle M, B\rangle : \langle?\psi\rangle\phi$ and (C1) $r : \rho \vdash \langle M, C\rangle : \langle?\psi\rangle\phi$, then by rule $\langle\cup\rangle$E on (A) have $\cdot \vdash \langle\text{case } A \text{ of } \ell \Rightarrow\ \langle M, B\rangle \mid r \Rightarrow\ \langle M, C\rangle\rangle : \langle?\psi\rangle\phi$.

Case $\langle[\pi_L]\rangle$C: We give the diamond case. The box case is analogous. From premises, have $\cdot \vdash \langle\pi_L\langle\text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rangle : \psi$ and by inversion (1) $\cdot \vdash \langle\text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : \langle?\psi\rangle\phi$, (A) $\cdot \vdash A : \zeta \vee \rho$, (B) $\ell : \zeta \vdash B : \langle?\psi\rangle\phi$, and (C) $r : \rho \vdash C : \langle?\psi\rangle\phi$. Then by rule $\langle?\rangle$E1 have (B1) $\ell : \zeta \vdash \langle\pi_L B\rangle : \psi$ and (C1) $r : \rho \vdash \langle\pi_L C\rangle : \psi$, then by rule $\langle\cup\rangle$E on (A) have $\cdot \vdash \langle\text{case } A \text{ of } \ell \Rightarrow\ \langle\pi_L B\rangle \mid r \Rightarrow\ \langle\pi_L C\rangle\rangle : \psi$.

Case $\langle[\pi_R]\rangle$C: We give the diamond case. The box case is analogous. From premises, have $\cdot \vdash \langle\pi_R\langle\text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rangle : \phi$ and by inversion (1) $\cdot \vdash \langle\text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : \langle?\psi\rangle\phi$, (A) $\cdot \vdash A : \zeta \vee \rho$, (B) $\ell : \zeta \vdash B : \langle?\psi\rangle\phi$, and (C) $r : \rho \vdash C : \langle?\psi\rangle\phi$. Then by rule $\langle?\rangle$E2 have (B1) $\ell : \zeta \vdash \langle\pi_R B\rangle : \phi$ and (C1) $r : \rho \vdash \langle\pi_R C\rangle : \phi$, then by rule $\langle\cup\rangle$E on (A) have $\cdot \vdash \langle\text{case } A \text{ of } \ell \Rightarrow\ \langle\pi_R B\rangle \mid r \Rightarrow\ \langle\pi_R C\rangle\rangle : \phi$.

Case $\langle\ell\cdot\rangle$C: From premises, have $\cdot \vdash \langle\ell \cdot \langle\text{case } A \text{ of } p \Rightarrow\ B \mid q \Rightarrow\ C\rangle\rangle : \langle\alpha \cup \beta\rangle\phi$ and by inversion (1) $\cdot \vdash \langle\text{case } A \text{ of } p \Rightarrow\ B \mid q \Rightarrow\ C\rangle : \langle\alpha\rangle\phi$, (A) $\cdot \vdash A : \zeta \vee \rho$, (B) $p : \zeta \vdash B : \langle\alpha\rangle\phi$, and (C) $q : \rho \vdash C : \langle\alpha\rangle\phi$. Then by rule $\langle\cup\rangle$I1 have (B1) $q : \zeta \vdash \langle\ell \cdot B\rangle : \langle\alpha \cup \beta\rangle\phi$ and

(C1) $q : \rho \vdash \langle \ell \cdot C\rangle : \langle \alpha \cup \beta\rangle\phi$, then by rule $\langle\cup\rangle$E on (A) have $\cdot \vdash \langle$case A of $p \Rightarrow\ \langle \ell \cdot B\rangle \mid q \Rightarrow\ \langle \ell \cdot C\rangle\rangle : \langle \alpha \cup \beta\rangle\phi$. The case for stopC (for $\langle \alpha^*\rangle\phi$) is symmetric.

Case $\langle r\cdot\rangle$C: From premises, have $\cdot \vdash \langle r \cdot \langle$case A of $p \Rightarrow\ B \mid q \Rightarrow\ C\rangle\rangle : \langle \alpha \cup \beta\rangle\phi$ and by inversion (1) $\cdot \vdash \langle$case A of $p \Rightarrow\ B \mid q \Rightarrow\ C\rangle : \langle \beta\rangle\phi$, (A) $\cdot \vdash A : \zeta \vee \rho$, (B) $p : \zeta \vdash B : \langle \beta\rangle\phi$, and (C) $q : \rho \vdash C : \langle \beta\rangle\phi$. Then by rule $\langle\cup\rangle$I2 have (B1) $p : \zeta \vdash \langle r \cdot B\rangle : \langle \alpha \cup \beta\rangle\phi$ and (C1) $q : \rho \vdash \langle r \cdot C\rangle : \langle \alpha \cup \beta\rangle\phi$, then by rule $\langle\cup\rangle$E on (A) have $\cdot \vdash \langle$case A of $p \Rightarrow\ \langle r \cdot B\rangle \mid q \Rightarrow\ \langle r \cdot C\rangle\rangle : \langle \alpha \cup \beta\rangle\phi$. The case for goC (for $\langle \alpha^*\rangle\phi$) is symmetric.

Case repC: From premises, have $\cdot \vdash (\langle$case A of $\ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$ rep $p : \psi.\ N$ in $O) : [\alpha^*]\phi$ and by inversion (1) $\cdot \vdash \langle$case A of $\ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : \psi$, (A) $\cdot \vdash A : \zeta \vee \rho$, (B) $\ell : \zeta \vdash B : \psi$, and (C) $r : \rho \vdash C : \psi$. Then by rule $[*]$I and because N is being applied in the same context as before have (B1) $\ell : \zeta \vdash (B$ rep $p : \psi.\ N$ in $O) : [\alpha^*]\phi$ and (C1) $r : \rho \vdash (C$ rep $p : \psi.\ N$ in $O) : [\alpha^*]\phi$, then by rule $\langle\cup\rangle$E on (A) have $\cdot \vdash \langle$case A of $\ell \Rightarrow\ (B$ rep $p : \psi.\ N$ in $O) \mid r \Rightarrow\ (C$ rep $p : \psi.\ N$ in $O)\rangle : [\alpha^*]\phi$.

Case app1C: From premises, have $\cdot \vdash \langle$case A of $\ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\ N : \phi$ and by inversion (1) $\cdot \vdash \langle$case A of $\ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : [?\psi]\phi$, (A) $\cdot \vdash A : \zeta \vee \rho$, (B) $\ell : \zeta \vdash B : [?\psi]\phi$, and (C) $r : \rho \vdash C : [?\psi]\phi$. Then by rule $[?]$E and weakening on N have (B1) $\ell : \zeta \vdash B\ N : \phi$ and (C1) $r : \rho \vdash C\ N : \phi$, then by rule $\langle\cup\rangle$E on (A) have $\cdot \vdash \langle$case A of $\ell \Rightarrow\ B\ N \mid r \Rightarrow\ C\ N\rangle : \phi$.

Case app2C: From premises, have $\cdot \vdash M\ \langle$case A of $\ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : \phi$ and by inversion (1) $\cdot \vdash \langle$case A of $\ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : \psi$ (A) $\cdot \vdash A : \zeta \vee \rho$, (B) $\ell : \zeta \vdash B : \psi$, and (C) $r : \rho \vdash C : \psi$. Then by rule $[?]$E and weakening on M have (B1) $\ell : \zeta \vdash M\ B : \phi$ and (C1) $r : \rho \vdash M\ C : \phi$, then by rule $\langle\cup\rangle$E on (A) have $\cdot \vdash \langle$case A of $\ell \Rightarrow\ M\ B \mid r \Rightarrow\ M\ C\rangle : \phi$.

Case $\langle^d\rangle$C: From premises, have $\cdot \vdash \langle$yield $\langle$case A of $\ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\rangle : \langle \alpha^d\rangle\phi$ and by inversion (1) $\cdot \vdash \langle$case A of $\ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : [\alpha]\phi$, (A) $\cdot \vdash A : \zeta \vee \rho$, (B) $\ell : \zeta \vdash B : [\beta]\phi$, and (C) $r : \rho \vdash C : [\beta]\phi$. Then by rule $\langle\!\langle^d\rangle\!\rangle$I have (B1) $\ell : \zeta \vdash \langle$yield $B\rangle : \langle \alpha^d\rangle\phi$ and (C1) $r : \rho \vdash \langle$yield $C\rangle : \langle \alpha^d\rangle\phi$, then by rule $\langle\cup\rangle$E on (A) have $\cdot \vdash \langle$case A of $\ell \Rightarrow\ \langle$yield $B\rangle \mid r \Rightarrow\ \langle$yield $C\rangle\rangle : \langle \alpha^d\rangle\phi$.

Case $[^d]$C: From premises, have $\cdot \vdash [$yield $\langle$case A of $\ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle] : [\alpha^d]\phi$ and by inversion (1) $\cdot \vdash \langle$case A of $\ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : \langle \alpha\rangle\phi$, (A) $\cdot \vdash A : \zeta \vee \rho$, (B) $\ell : \zeta \vdash B : \langle \beta\rangle\phi$, and (C) $r : \rho \vdash C : \langle \beta\rangle\phi$. Then by rule $\langle\!\langle^d\rangle\!\rangle$I have (B1) $\ell : \zeta \vdash [$yield $B] : [\alpha^d]\phi$ and (C1) $r : \rho \vdash [$yield $C] : [\alpha^d]\phi$, then by rule $\langle\cup\rangle$E on (A) have $\cdot \vdash \langle$case A of $\ell \Rightarrow\ [$yield $B] \mid r \Rightarrow\ [$yield $C]\rangle : [\alpha^d]\phi$.

Case appC: From premises, have $\cdot \vdash \langle$case A of $\ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle\ f : \phi^f_x$ and by inversion (1) $\cdot \vdash \langle$case A of $\ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : [x := *]\phi$, (A) $\cdot \vdash A : \zeta \vee \rho$, (B) $\ell : \zeta \vdash B : [x := *]\phi$, and (C) $r : \rho \vdash C : [x := *]\phi$. Then rule $[:*]$E gives (B1) $\ell : \zeta \vdash B\ f : \phi^f_x$ and (C1) $r : \rho \vdash B\ f : \phi^f_x$, then rule $\langle\cup\rangle$E on (A) gives $\cdot \vdash \langle$case A of $\ell \Rightarrow\ B\ f \mid r \Rightarrow\ C\ f\rangle : \phi^f_x$.

Case $\circ$C: From premises, have $\cdot \vdash \langle$case A of $\ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \circ_p N : \langle \alpha\rangle\psi$ and by inversion (1) $\cdot \vdash \langle$case A of $\ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : \langle \alpha\rangle\phi$, (A) $\cdot \vdash A : \zeta \vee \rho$, (B) $\ell : \zeta \vdash B : \langle \alpha\rangle\phi$, and (C) $r : \rho \vdash C : \langle \alpha\rangle\phi$. Then by monotonicity rule M and by weakening N with ζ^z_x and ρ^z_x, then (B1) $\ell : \zeta \vdash B \circ_p N : \langle \alpha\rangle\psi$ and (C1) $r : \rho \vdash C \circ_p N : \langle \alpha\rangle\psi$, then by rule $\langle\cup\rangle$E on (A) have $\cdot \vdash \langle$case A of $\ell \Rightarrow\ B \circ_p N \mid r \Rightarrow\ C \circ_p N\rangle : \langle \alpha\rangle\psi$.

Case FPC: From premises, have $\cdot \vdash FP(\langle$case A of $\ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle, s.\ D, g.\ E) : \psi$ and by inversion (1) $\cdot \vdash \langle$case A of $\ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : \langle \alpha^*\rangle\phi$, (A) $\cdot \vdash A : \zeta \vee \rho$, (B) $\ell : \zeta \vdash B : \langle \alpha^*\rangle\phi$, and (C) $r : \rho \vdash C : \langle \alpha^*\rangle\phi$. Then by rule FP and since D and E are applied

with the same context as before, then (B1) $\ell : \zeta \vdash FP(B, s.\ D, g.\ E) : \psi$ and (C1) $r : \rho \vdash FP(C, s.\ D, g.\ E) : \psi$, then by rule $\langle\cup\rangle$E on (A) have $\cdot \vdash \langle \text{case } A \text{ of } \ell \Rightarrow\ FP(B, s.\ D, g.\ E) \mid r \Rightarrow\ FP(C, s.\ D, g.\ E)\rangle : \psi$.

Case caseC: From premises, have $\cdot \vdash \langle \text{case } \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \text{ of } \ell \Rightarrow\ D \mid r \Rightarrow\ E\rangle : \rho$ and by inversion (1) $\cdot \vdash \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : \phi \vee \psi$, (A) $\cdot \vdash A : \zeta_1 \vee \rho_1$, (B) $\ell : \zeta_1 \vdash B : \phi \vee \psi$, and (C) $r : \rho_1 \vdash C : \phi \vee \psi$. Then by rule $\langle\cup\rangle$E and since D and E are applied with the same context as before (that is, we allow ℓ and r to shadow here as the outer binding is never used), then (B1) $\ell : \zeta_1 \vdash \langle \text{case } B \text{ of } \ell \Rightarrow\ D \mid r \Rightarrow\ E\rangle : \rho$ and (C1) $r : \rho_1 \vdash \langle \text{case } C \text{ of } \ell \Rightarrow\ D \mid r \Rightarrow\ E\rangle : \rho$, then by rule $\langle\cup\rangle$E on (A) have $\cdot \vdash \langle \text{case } A \text{ of } \ell \Rightarrow\ \langle \text{case } B \text{ of } \ell \Rightarrow\ D \mid r \Rightarrow\ E\rangle \mid r \Rightarrow\ \langle \text{case } C \text{ of } \ell \Rightarrow\ D \mid r \Rightarrow\ E\rangle\rangle : \rho$.

Case $\langle{:}*\rangle$C: From premises, have $\cdot \vdash \text{unpack}(\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle, py.\ N) : \psi$ and by inversion (1) $\cdot \vdash \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : \langle x := *\rangle\phi$, (A) $\cdot \vdash A : \zeta \vee \rho$, (B) $\ell : \zeta \vdash B : \langle x := *\rangle\phi$, and (C) $r : \rho \vdash C : \langle x := *\rangle\phi$. Then by rule $\langle{:}*\rangle$E and by weakening N with $\zeta\frac{y}{x}$ and $\rho\frac{y}{x}$ have (B1) $\ell : \zeta \vdash \text{unpack}(B, py.\ N) : \psi$ and (C1) $r : \rho \vdash \text{unpack}(C, py.\ N) : \psi$, so rule $\langle\cup\rangle$E on (A) gives $\cdot \vdash \langle \text{case } A \text{ of } \ell \Rightarrow \text{unpack}(B, py.\ N) \mid r \Rightarrow \text{unpack}(C, py.\ N)\rangle : \psi$.

Case forC: Then have $\cdot \vdash \text{for}(p : \varphi(\mathcal{M}) = \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle; q; N)\,\{\alpha\}\,O : \langle\alpha^*\rangle(\phi)$ from the premises and by inversion (1) $\cdot \vdash \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : \varphi$, (A) $\cdot \vdash A : \zeta \vee \rho$, (B) $\ell : \zeta \vdash B : \varphi$, and (C) $r : \rho \vdash C : \varphi$. Then by rule $\langle*\rangle$I and because N is applied only in its original context, (B1) $\ell : \zeta \vdash \text{for}(p : \varphi(\mathcal{M}) = B; q; N)\,\{\alpha\}\,O : \langle\alpha^*\rangle\phi$ and (C1) $r : \rho \vdash \text{for}(p : \varphi(\mathcal{M}) = C; q; N)\,\{\alpha\}\,O : \langle\alpha^*\rangle\phi$, then by rule $\langle\cup\rangle$E on (A) have

$$\begin{aligned}\cdot \vdash \langle &\text{case } A \text{ of}\\ &\ell \Rightarrow\ \text{for}(p : \varphi(\mathcal{M}) = B; q; N)\,\{\alpha\}\,O\\ &\mid r \Rightarrow\ \text{for}(p : \varphi(\mathcal{M}) = C; q; N)\,\{\alpha\}\,O\\ &: \langle\alpha^*\rangle\phi\end{aligned}$$

Case unrollC: From premises, have $\cdot \vdash [\text{unroll } \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle] : \phi \wedge [\alpha][\alpha^*]\phi$ and by inversion (1) $\cdot \vdash \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : [\alpha^*]\phi$, (A) $\cdot \vdash A : \zeta \vee \rho$, (B) $\ell : \zeta \vdash B : [\alpha^*]\phi$, and (C) $r : \rho \vdash C : [\alpha^*]\phi$. Then by rule $[*]$E, have (B1) $\ell : \zeta \vdash [\text{unroll } B] : \phi \wedge [\alpha][\alpha^*]\phi$ and (C1) $r : \rho \vdash [\text{unroll } C] : \phi \wedge [\alpha][\alpha^*]\phi$, then by rule $\langle\cup\rangle$E on (A) have $\cdot \vdash \langle \text{case } A \text{ of } \ell \Rightarrow\ [\text{unroll } B] \mid r \Rightarrow\ [\text{unroll } C]\rangle : \phi \wedge [\alpha][\alpha^*]\phi$.

Case caseC*: Have that $\cdot \vdash \langle \text{case}_* \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \text{ of } s \Rightarrow\ D \mid g \Rightarrow E\rangle : \psi$ from the premises and by inversion (1) $\cdot \vdash \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle : \langle\alpha^*\rangle\phi$, (A) $\cdot \vdash A : \zeta \vee \rho$, (B) $\ell : \zeta \vdash B : \langle\alpha^*\rangle\phi$, and (C) $r : \rho \vdash C : \langle\alpha^*\rangle\phi$. Then by rule $\langle*\rangle$C, have (B1) $\ell : \zeta \vdash \langle \text{case}_* B \text{ of } s \Rightarrow\ D \mid g \Rightarrow E\rangle : \psi$ and (C1) $r : \rho \vdash \langle \text{case}_* C \text{ of } s \Rightarrow\ D \mid g \Rightarrow E\rangle : \psi$, then by rule $\langle\cup\rangle$E on (A) have $\cdot \vdash \langle \text{case } A \text{ of } \ell \Rightarrow\ \langle \text{case}_* B \text{ of } s \Rightarrow\ D \mid g \Rightarrow E\rangle \mid r \Rightarrow\ \langle \text{case}_* C \text{ of } s \Rightarrow\ D \mid g \Rightarrow E\rangle\rangle : \psi$. □

Appendix B

Appendices to Chapter 5

B.1 Example Proofs

We restate the definition of the 1D driving game and its reach-avoid specification here:

$$\begin{aligned}
\mathsf{ctrl} &\equiv a := *; ? -B \leq a \leq A; t := 0 \\
\mathsf{plant} &\equiv \{t' = 1, x' = v, v' = a \,\&\, t \leq T \wedge v \geq 0\}^d \\
\mathsf{pre} &\equiv T > 0 \wedge A > 0 \wedge B > 0 \wedge v = 0 \wedge x = 0 \\
\mathsf{safe} &\equiv x \leq g \\
\mathsf{reachAvoid} &\equiv \mathsf{pre} \rightarrow \langle (\mathsf{ctrl}; \mathsf{plant}; ?\mathsf{safe}; ?(t \geq T/2)^d)^* \rangle (g = x \wedge v = 0)
\end{aligned}$$

We first give an overview of our proof approach, then give the main algebraic derivations, then finally the complete natural deduction proof.

B.1.1 Proof Overview

While safety of 1D driving is a thoroughly-studied introductory example, adversarial 1D reach-avoid is more challenging due to the combination of adversarial timing and liveness.

To simplify our arithmetic, the proof uses the same acceleration magnitude $C = \min(A, B)$ to both accelerate and brake. The resulting strategy is conservative, but still satisfies reach-avoid correctness. Our proof proceeds by convergence: we establish a minimum distance Δx which is traversed in each iteration, guaranteeing that the goal is eventually reached. The minimum distance is determined by appealing to a velocity envelope which is invariant throughout the loop, given as a function of the position. Much of Section B.1.2 is devoted to identifying a velocity envelope which is invariant while also strong enough to ensure liveness. The car's velocity goes to 0 during braking, so the key is to show that velocity decreases slowly enough to ensure sufficient progress toward the desired goal g. We depict the safe driving envelope in Fig. B.1.

A major task of the controller is to detect events within an adversarial environment. We detect one event: when are we close enough to the goal that we must brake? Because our acceleration and braking rates are the same, it suffices to begin braking by the midpoint

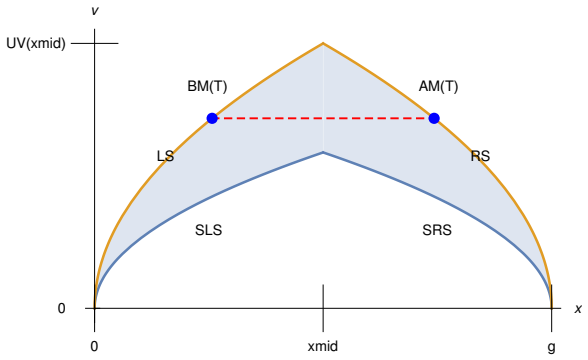


Figure B.1: Safe driving envelope.

$x = g/2$. Care is still required because the controller is *time-triggered*: we must determine whether it is possible to cross the midpoint within the coming timestep, and react right *before* we actually cross the midpoint. In Fig. B.1, the blue point $BM(T)$ is the point at which we would start to react.

The other major task of the controller is to choose an acceleration value. Until we approach the midpoint, the acceleration is at the maximum value C. Once the midpoint is detected, acceleration is computed with a predictive method: compute the state at the end of the timestep, then solve for the greatest C that maintains safety.

Recall that CdGL features Type-2 effective computations on reals. We note that in proofs which use inexact comparisons for constructive reals, only approximate, not exact "reach" properties might be provable. The reason our proof obtains an exact result is subtle: in Type-2 effectivity, term-level comparisons (i.e., extrema computations) with the functions $\min(f, g)$ and $\max(f, g)$ are allowed to be exact, in stark contrast to inexact formula-level comparisons.

We take a brief aside to explain why and how Type-2 effectivity permits exact computations of $\min(f, g)$ and $\min(f, g)$. While CdGL does not require any particular representation of computable real numbers, the standard notion of Type-2 effectivity assumes computable numbers are represented by infinite (think: lazy) streams of arbitrarily precise approximations, so we found it natural to consider the representation of computable reals by infinite streams of bits for the sake of this discussion. Specifically, to explain why a function is Type-2 effective, we will show it is effective when the input and output are represented as lazy streams of bits. There is more than one way to lazily represent f and g; we assume that the infinite stream of bits is the mantissa of their floating-point representation and that they are normalized to use the same (finite but unbounded) exponent $k \in \mathbb{Z}$.

The *exact* binary representation of $\min(f, g)$ and $\max(f, g)$ are computable *lazily* in a stream representation, which is why they are Type-2 computable. We describe how to implement $\min(f, g)$ lazily as an example. When first asked to generate a bit of the output, compare the corresponding bit in f and in g. If they agree, output their (shared) value. So long as the values agree, keep outputting the shared result. Else, exactly one of f or g generated the bit 0; that is the smaller number. Generate all (infinitely many) remaining bits of the minimum by copying the bits generated by the smaller number.

The algorithm described in the previous paragraph works even if f and g have identical representations: it lazily returns their shared representation. While there exist real numbers with multiple binary representations ($1.0_2 = 0.111\ldots_2$), the existence of multiple representations can only make the problem easier, because both 1.0_2 and $0.111\ldots_2$ are correct outputs of $\max(1.0_2, 0.111\ldots_2)$.

B.1.2 Algebraic Derivations

We now algebraically derive the main equations of the proof, e.g., the equations for invariant regions, termination metrics, and the acceleration controller. In this section, "monotonicity" does not refer to the monotonicity rule for game modalities, but to monotone functions in the arithmetic sense, e.g.,

$$x \geq y \rightarrow f(x) \geq f(y)$$

The following development employs the well-known Newtonian motion equations:

$$v(t) = v(0) + at$$
$$x(t) = x(0) + v(0)t + \frac{at^2}{2}$$

Because our initial conditions are $v(0) = x(0) = 0$, we can eliminate the first terms. We write $x(k)$ and $v(k)$ for the values of x and v at the beginning of a given iteration of the game loop, as opposed to the beginning of the game. From the Newton equations we derive the safe-braking (SB) inequality, which says braking rate $-C$ suffices to stop the car by the time x reaches g:

$$\text{SB} \equiv \left(\frac{v^2}{2C} \leq g - x\right)$$

When the equality is strict, then x reaches g exactly as the car stops.

We write x_{mid} for the midpoint, i.e., $g/2$. We write v_{mid} for the maximum velocity which may ever be attained, i.e., the maximum velocity one might have at the midpoint and still brake safely. We write $\text{UV}(x)$ for the **u**pper bound of permissible **v**elocity at a position x. The shape of UV will be a convex, curved "triangle" which is 0 at $x = 0$ and $x = g$ and attains value v_{mid} at $x = x_{mid}$. The shape of UV is curved rather than a true triangle because its edges are defined by square-root expressions defining the relationship between position and velocity.

Its **l**eft **s**ide (LS) is derived by setting the maximum acceleration $a = C$ and solving the Newton equations for v as a function of x:

$$\text{LS}(x) = \sqrt{2Cx}$$

The **r**ight **s**ide (RS) is derived by setting the braking acceleration to C, assuming SB holds as a strict equality, and solving for v as a function of x:

$$\text{RS}(x) = \sqrt{2C(g - x)}$$

The upper velocity envelope is their minimum:

$$\mathrm{UV}(x) = \min(\mathrm{LS}(x), \mathrm{RS}(x))$$

We refer to the set of points bounded by these curves as the Large Triangle (LT), with the understanding that its sides are actually convex curves.

Choosing a **l**ower **v**elocity bound $\mathrm{LV}(x)$ at each point x is more challenging, because velocity necessarily decays as x approaches g. To prove that position $x = d$ is eventually reached, our proof must rule out so-called "Zeno" behaviors, such as those where distance traversed in each timestep decreases exponentially. Ruling out Zeno behaviors requires a strong lower bound LV. Strong bounds are of course desirable in and of themselves: the higher the bound, the faster x is guaranteed to reach g.

Our control scheme predicts future motion: a control choice is safe if for every duration $t \in [0, T]$, the motion is safe. By monotonicity, it suffices to show the case $t = T$. The lower bound likewise predicts motion, we introduce several helper functions which predict motion. We write $\mathrm{BM}(T)$ (before midpoint) for "the position x from which the car will reach the midpoint in time T at maximum acceleration." Under time-triggered control, $\mathrm{BM}(T)$ is the "point of no return" by which Angel must react, and we will safety detect the approaching midpoint by comparing our position to $\mathrm{BM}(T)$. We also write $\mathrm{AM}(T)$ (after midpoint) for the conjugate point of $\mathrm{BM}(T)$ opposite the midpoint. To derive $\mathrm{BM}(T)$ we set $x(k)+\mathrm{UV}(x(k))T+\frac{CT^2}{2} = g/2$ with the simplification that $\mathrm{UV}(x(k)) = \mathrm{LS}(x(k))$ before the midpoint. By algebra, the solutions of:

$$x + \sqrt{2Cx}T + \frac{CT^2}{2} = g/2$$

are the equation

$$\mathrm{BM}(T) = \frac{1}{2}(-2T\sqrt{Cg} + CT^2 + g)$$

and its conjugate

$$\mathrm{AM}(T) = \frac{1}{2}(2T\sqrt{Cg} - CT^2 + g)$$

The need for safe braking makes it clear that control must be more conservative past $\mathrm{BM}(T)$, the only question is *how* conservative. For example, the minimum safe acceleration from $\mathrm{BM}(T/2)$ is 0, which takes us to the boundary at $\mathrm{AM}(T/2)$ if Demon chooses $t = T$. We might wonder if it is sufficient for reach-avoid correctness to exclude all states above the line connecting $\mathrm{BM}(T/2)$ and $\mathrm{AM}(T/2)$, indicated by a red dashed line in Fig. B.1. It is not, under adversarial timing. Consider blue point $\mathrm{BM}(T/2)$ again. Demon may choose $t = T/2$ so that by definition Angel regains control at $x = x_{mid}$ but $v < v_{mid}$. Demon now has a strategy to keep Angel strictly in the interior of the Large Triangle indefinitely, so that the lower bound is eventually violated, probably after $\mathrm{AM}(T/2)$.

Our envelope can only hope to be an invariant if a *strict* inequality $\mathrm{LV}(x) < \mathrm{UV}(x)$ holds for *all* $x \in (x_{mid}, g)$. We believe that the optimal lower bound is particularly nontrivial, so we do not aim to show our bound is perfectly tight. We do note that our bound is not exceptionally loose, either. For example, if we were to permit Demon to elapse physics up to

time $2T$, then any strategy more aggressive than ours becomes clearly unsafe. Regardless, we believe the controller used in this proof is tight modulo our simplifying assumption $A = B = C$, we simply use this slightly looser bound for the sake of *proving* liveness.

The starting observation is that Angel might need to decrease acceleration as early as $\mathrm{BM}(T)$. The simplest live decision would be to construct a braking rate $a_{end} > 0$ which reaches g exactly when $v = 0$, and simply brake at rate $a = -a_{end}$ until stopped. The braking curve of rate a_{end} form the small right side (SRS) of the triangle, while its mirror forms the small left side (SLS). Perhaps we could reuse LS for the left side, but we preserve symmetry in hopes of simplifying the proof.

Consider the upper-left point $\mathrm{UL} = (\mathrm{BM}(T), \mathrm{UV}(\mathrm{BM}(T)))$ and upper-right point $\mathrm{UR} = (\mathrm{AM}(T), \mathrm{UV}(\mathrm{AM}(T)))$. The conservative braking rate a_{end} is defined by setting SB as an equality and solving for a as a function of x and v.

$$
\begin{aligned}
&\left(\frac{\mathrm{UV}(\mathrm{BM}(T))^2}{2a_{end}} = g - \mathrm{BM}(T)\right) \\
\text{iff } &\left(a_{end} = \frac{\mathrm{LS}(\mathrm{BM}(T))^2}{2(g - \mathrm{BM}(T))}\right) \\
= &\frac{\mathrm{LS}(\frac{1}{2}(-2T\sqrt{Cg} + CT^2 + g))^2}{2(g - \frac{1}{2}(-2T\sqrt{Cg} + CT^2 + g))} \\
= &\frac{\mathrm{LS}(\frac{1}{2}(-2T\sqrt{Cg} + CT^2 + g))^2}{2g + 2T\sqrt{Cg} - CT^2 - g} \\
= &\frac{\mathrm{LS}(\frac{1}{2}(-2T\sqrt{Cg} + CT^2 + g))^2}{g + 2T\sqrt{Cg} - CT^2} \\
= &\frac{\mathrm{LS}(-T\sqrt{Cg} + CT^2/2 + g/2))^2}{g + 2T\sqrt{Cg} - CT^2} \\
= &\frac{\left(\sqrt{2C(-T\sqrt{Cg} + C/2T^2 + g/2)}\right)^2}{g + 2T\sqrt{Cg} - CT^2} \\
= &\frac{2C(-T\sqrt{Cg} + CT^2/2 + g/2)}{g + 2T\sqrt{Cg} - CT^2} \\
= &\frac{C(\sqrt{g} - T\sqrt{C})^2}{g + 2T\sqrt{Cg} - CT^2}
\end{aligned}
$$

Then we define the Small Triangle (ST) as the set of points bounded by:

$$
\begin{aligned}
\mathrm{SLS}(x) &= \sqrt{2a_{end}x} \\
\mathrm{SRS}(x) &= \sqrt{2a_{end}(g - x)}
\end{aligned}
$$

The lower velocity envelope is their minimum:

$$\mathrm{LV}(x) = \min(\mathrm{SLS}(x), \mathrm{SRS}(x))$$

We are finally ready to give the invariant formula φ and the progress term Δx which induces the termination metric. The invariant simply says the velocity is between the envelopes and gives the signs of state variables:

$$\varphi \leftrightarrow \mathrm{LV}(x) \leq v \leq \mathrm{UV}(x) \wedge x \geq 0 \wedge v \geq 0$$

The minimum progress is found by taking the shortest distance traversed along any path of duration $T/2$ contained within the envelope between triangles LT and ST:

$$\Delta x = \inf_{(x,v)\in \mathrm{LT}\setminus\mathrm{ST}, t\in[T/2,T]} \inf_a (vt/2 + at^2/2)$$

where a ranges over accelerations which remain within the envelope for time t. We observe that distance traversed is monotone in x, v, and t. Thus, it suffices to consider the worst case where $t = T/2, x = 0, v = 0, a = -a_{end}$. The worst-case acceleration is $a = a_{end}$ because a_{end} defines the lower bound of the velocity envelope. The worst case is then

$$\frac{a_{end}T^2}{8}$$

We construct our termination argument. Constructive termination arguments are subtle, so our termination metric has a subtle design in order to make the termination argument work. We do *not* use a termination metric whose value is always a real number. Rather than define the terminal value $\mathbf{0}$ as a specific real number, we treat $\mathbf{0}$ as a distinguished value, so that the type of $\mathcal{M}$ is a disjoint union of the reals and the singleton type containing $\mathbf{0}$, meaning the real number 0 and distinguished $\mathbf{0}$ are distinct. Value $\mathbf{0}$ represents the case where we *know* the metric is zero and the loop can be terminated; because comparisons are inexact, it is possible for the metric to reach 0 without our having a proof of it, a case represented by 0 rather than $\mathbf{0}$.

The metric is defined by:

$$\begin{array}{llll} r \succcurlyeq \mathbf{0} & r \leq 0 \to \mathbf{0} \succcurlyeq r & r \geq s \to r \succcurlyeq s & \mathbf{0} \succcurlyeq \mathbf{0} \\ r \succ \mathbf{0} & r < 0 \to \mathbf{0} \succ r & r > s + \Delta x \to r \succ s & \end{array}$$

The definition of virtual term $\mathcal{M}$ is equally subtle. Intuitively speaking, the remaining distance $g - x$ is used as a metric. However, the metric $\mathcal{M}$ should return the distinguished value $\mathbf{0} \neq 0$ when the distance is *provably* zero. The major technical challenge in defining the metric $\mathcal{M}$ is that no deterministic computable function of the state can distinguish the case where the distance is provable zero ($\mathcal{M} = \mathbf{0}$) from the case where the distance is unprovably zero ($\mathcal{M} = 0$). The metric is only *provably* zero if an inexact comparison tells us so, but inexact comparisons are resolved nondeterministically; we do not control them.

We rigorously construct $\mathcal{M}$ by appealing to choice functions and the Strong Collection axiom[1] from constructive set theory (Aczel & Rathjen, 2001, §2.3, §7):

$$\forall S\,((\forall x : S\ \exists y\, p(x,y)) \to (\exists R\,((\forall x : S\ \exists y : R\ p(x,y)) \wedge (\forall y : R\ \exists x : S\ p(x,y)))))$$

[1]Formalized as an axiom *schema* in the literature (Aczel & Rathjen, 2001, §2.3, §7). We do not use the minimality assumption on R, i.e., we do not need the *strong* axiom, but we state it in order to be faithful to the cited sources, which use the strong axiom (schema).

where S and R range over sets, i.e., the axiom requires second-order quantifiers. In short, the collection axiom takes a relation p and a proof that $p(x, y)$ is a well-moded relation with input x and output y, then constructs the minimal codomain R: an element of R (an output) is constructed for every element of S (input). Additionally, an element of S (input) is constructed for every element of R (output), thus showing the codomain is minimal.

The metric term $\mathcal{M}$ is built from a *multi-valued function* F: in the case $g - x = 0$, both values in the set $\{0, \mathbf{0}\}$ are possible, depending on the results of a comparing 0 and $\mathcal{M}$. The computational interpretation of the collection axiom is a choice function which when applied to F yields a single-valued function (call it $\mathit{choice}(F)$) which deterministically selects its result from the set of multiple return values of F. Different constructive theories differ in the strength of their collection axioms and choice axioms, but termination metrics only require choice over sets of size 2, which is provided by all common collection axioms.

Let ε be the comparison epsilon used in the termination check, then we define $F(x) = \{\mathbf{0}\}$ when $g - x \leq -\varepsilon$, $F(x) = \{0, \mathbf{0}\}$ when $-\varepsilon < g - x \leq 0$ and $F(x) = \{g - x\}$ otherwise, then let $\mathcal{M} = \mathit{choice}(F)$. That is, $\mathcal{M}$ agrees with comparison result in all states where the comparison has a determined value, but can take on either value (consistently) in states where the comparison is indeterminate[2].

For readers who may be uncomfortable with the use of multi-valued terms, it may be useful to compare our convergence argument against a dL-style argument using variants and ghost variables. In the case where $\mathcal{M}$ has multiple possible values, the value is resolved Demonically (i.e., we do not choose it). In a dL-style argument, the choice between 0 and $\mathbf{0}$ could be resolved Angelically by implementing $\mathcal{M}$ as an existential ghost variable rather than a term. In cases where $g - x \leq 0$ is known, the special value $\mathbf{0}$ would be chosen, else the value $g - x$ is chosen. Our Angel also chooses $\mathbf{0}$ in cases where $g - x \leq 0$ is known, the only difference being that our choice function makes it explicit that Demon can force Angel to choose 0 rather than $\mathbf{0}$ where the comparison is underdetermined. Our development uses a choice function rather than a ghost variable for $\mathcal{M}$ because of our stylistic preference for explicit terms, but both approaches are technically feasible and the latter is likely more familiar to many readers.

The benefit of our subtle definition of $\mathcal{M}$ is that the inductive step of the induction is allowed to prove a disjunction: either prove that the loop has made sufficient progress *or* prove that the loop has finished. This disjunctive structure is well-suited for our setting because constructive comparisons can at best establish upper bounds on remaining distance which, when sufficiently low, suffice to prove that a system terminates in the next step, yet do not suffice to decide whether the system has already reached its goal. Because our setting is also adversarial, we also cannot compute the exact duration of a loop in advance.

We have now discussed the invariant and metric, which are two of three major components of the proof. The last major component is the strategy for choosing a, which we

[2]The comparison in the proof will not actually compare the final value of $g - x$ against the bound 0 because that comparison would be indeterminate even when ε is small. Rather, the distance remaining is compared in the initial state of the loop body, from which the final value of $\mathcal{M}$ is inferred in proof. Performing a comparison in the initial state is sufficient under the modest assumption comparison respects transition, i.e., we modestly assume that for all x, y, and k the results of comparing x against y or $x + k$ against $y + k$ agree.

have only alluded to thus far. We wish to set the highest acceleration that is guaranteed to remain within the velocity envelope.

To find this acceleration, recall the motion equations:

$$x(k+t) = x(k) + v(k)t + a\frac{t^2}{2}$$
$$v(k+t) = v(k) + at$$

where $v(k)$ and $x(k)$ are the values of state variables v and x at the start of the current iteration of the game loop.

The most aggressive safe acceleration is that which satisfies SB as a strict equality after the pessimal time interval T, so we set

$$\frac{(v+aT)^2}{2C} = \left(g - (x(k) + v(k)t + a\frac{t^2}{2})\right)$$

and solve for a. By algebra, there are two conjugate solutions (assuming $T \neq 0$ and $C \neq 0$, assumptions which are true):

$$a = -\frac{\sqrt{C}\sqrt{CT^2 + 8g - 4Tv - 8x} + CT + 2v}{2T}$$
$$a = \frac{\sqrt{C}\sqrt{CT^2 + 8g - 4Tv - 8x} + CT + 2v}{2T}$$

the latter of which is positive because $T > 0, v \geq 0, C > 0$.

We take this second solution as a candidate for the acceleration:

$$a_{cand} = \frac{\sqrt{C}\sqrt{CT^2 + 8g - 4Tv - 8x} - CT - 2v}{2T}$$

Recall that accelerations are required to fall within the range $[-B, A]$ and that for simplicity we show the stronger condition $a \in [-C, C]$. The lower bound $a_{cand} \geq -C$ holds by construction: a_{cand} is the *greatest* acceleration which remains within the safe envelope. By construction of the envelope, an acceleration $-C$ always remains below the upper limit, so a_{cand} must be at least $-C$. The upper bound $a_{cand} \leq C$ does not hold in general: we computed a_{cand} as the acceleration required to reach the maximum velocity *in this timestep*, yet reaching maximum velocity usually takes multiple timesteps. Thus, the final acceleration (acc) is computed by computably bounding a_{cand} against the upper limit C:

$$\text{acc} \equiv \min\Big(C, a_{cand}\Big)$$

B.1.3 Natural Deduction Proof

We give a formal proof in the natural deduction calculus. We first give derived rules and lemmas used in the proof, and we split the deduction into small pieces for the sake of simplifying its formatting.

B.1.3.1 Derived Rules

The vacuity axiom for constant propositions (indicated $p()$), which is sound for systems, is not sound for games (Platzer, 2015a):

$$p() \not\rightarrow [\alpha]p()$$

The following rule is sound for games, however, and can be derived from Lemma 5.6 and Lemma 5.1:

$$\text{(GV)} \quad \frac{\Gamma \vdash p() \quad \Gamma \vdash \langle\alpha\rangle Q}{\Gamma \vdash \langle\alpha\rangle p()}$$

The formula Q is arbitrary: as soon as Angel has *any* winning strategy, vacuity becomes sound. Formula $Q = \mathit{true}$ is usually chosen in practice because it is the easiest to prove: by rule M, any other choice of Q would imply the case for $Q = \mathit{true}$. Rule GV is analogous to the sound axiom schema VK of dGL (Platzer, 2015a).

As discussed in Section 5.4, we do not axiomatize first-order reasoning in this paper, but assume it has been implemented in "the host logic." Thus, we label first-order steps "FO" but do not give arithmetic proofs in full axiomatic detail. To be precise, the following (non-effective!) rule is sound:

$$\text{(FO)} \quad \frac{*}{\Gamma \vdash \phi} \quad \text{where exists } M : (\Pi s : \mathfrak{S}.\, {}^{\ulcorner}\Gamma^{\urcorner}(s) \rightarrow {}^{\ulcorner}\phi^{\urcorner}\, s) \text{ and } \Gamma, \phi \text{ F.O.}$$

B.1.3.2 Lemmas

We use several arithmetic facts throughout the proof.

Lemma B.1 (Safe Upper Bound)**.** *Braking is safe when the upper velocity bound is satisfied. Formula $v \leq UV(x) \rightarrow (g - x) \geq SB(x, v)$ is provable in context $(C > 0)$.*

Proof. By first-order arithmetic. □

Lemma B.2 (Acceleration in Bounds)**.** *Our control algorithm only proposed accelerations which are feasible. Formula $SB \rightarrow -C \leq acc \leq C \rightarrow -B \leq acc \leq A$ is provable in context $(A > 0 \wedge B > 0 \wedge C = \min(A, B) \wedge (g - x) \geq SB)$.*

Proof. The lower bound holds by construction: as discussed in the last section, acc $\geq -C$ when $(g - x) \geq$ SB. The upper bound holds trivially because acc is computed by computably bounding a_{cand} to C. □

B.1.3.3 Main Proof

The main proof begins by applying the convergence rule $\langle * \rangle$I, which is parameterized by a termination metric $\mathcal{M}$ with termination value $\mathbf{0}$ and ordering relations $\succcurlyeq, \succ$. The metric is as described in Section B.1.1: the metric tracks distance $\mathcal{M} = (g - x)$ with a unique distinguished value $\mathbf{0}$ for provable termination. We write Δx for the minimum progress

required per iteration. The invariant φ says the signs of variables are preserved and that velocity remains within its envelope:

$$\varphi \leftrightarrow (\mathrm{LV}(x) \leq v \leq \mathrm{UV}(x) \wedge x \geq 0 \wedge v \geq 0)$$

Note we do not explicitly include the sign conditions on T, A, B, C in the invariant because they are constants. By rule GV, for any game α and constant proposition (indicated $p()$), we have $p() \rightarrow \langle\alpha\rangle p()$ whenever $\langle\alpha\rangle\psi$ holds for *any* postcondition ψ. Loop convergence contains such a proof, thus vacuity can always applied to constants of a convergence proof in the inductive step. We abbreviate $\mathcal{D}_{\mathrm{pre}}$ for the preconditions of the proof and $\mathcal{D}_{\mathrm{post}}$ for the postcondition.

$$\text{implyR}\,\dfrac{\dfrac{\mathcal{D}_{\mathrm{pre}}}{\mathsf{pre} \vdash \varphi} \quad \dfrac{\mathcal{D}_{\mathrm{body}}}{\varphi, \mathcal{M}_0 = \mathcal{M} \succ \mathbf{0} \vdash (\varphi \wedge \mathcal{M}_0 \succ \mathcal{M})} \quad \dfrac{\mathcal{D}_{\mathrm{post}}}{\varphi \wedge \mathbf{0} \succcurlyeq \mathcal{M} \vdash \phi}}{\dfrac{\mathsf{pre} \vdash \langle\alpha^*\rangle\mathsf{post}}{\mathsf{pre} \rightarrow \langle\alpha^*\rangle\mathsf{post}}}\,\langle*\rangle\text{I}$$

We first dispatch the precondition and postcondition steps, which are purely arithmetic.

$$\text{FO}\dfrac{*}{T > 0 \wedge A > 0 \wedge B > 0 \wedge v = 0 \wedge x = 0 \vdash (\mathrm{LV}(x) \leq v \leq \mathrm{UV}(x) \wedge x \geq 0 \wedge v \geq 0)}$$

By construction, when $x = v = 0$ then $\mathrm{LV}(x) = \mathrm{UV}(x) = 0$ and the first two conjuncts are trivially satisfied. The latter two conjuncts follow directly from $v = 0$ and $x = 0$.

$$\text{FO}\dfrac{*}{(\mathrm{LV}(x) \leq v \leq \mathrm{UV}(x) \wedge x \geq 0 \wedge v \geq 0) \wedge 0 \geq (g - x) \vdash (g = x \wedge v = 0)}$$

Since $v \leq \mathrm{UV}(x)$ then (SB) $\frac{v^2}{2C} \leq (g - x)$. The LHS is always nonnegative, so $x \leq g$. Since $0 \geq (g - x)$ then $x \geq g$ so $g = x$ as desired. Moreover $\frac{v^2}{2C} \leq (g - x) = 0$ so $v = 0$ as desired.

We proceed to the proof of the loop body. We abbreviate $\Gamma \equiv \mathrm{LV}(x) \leq v \leq \mathrm{UV}(x) \wedge x \geq 0 \wedge v \geq 0, \mathcal{M}_0 = (g - x) \geq 0$, abbreviate $\Gamma'(f) \leftrightarrow \Gamma, a = \mathrm{acc}, f \leq t \leq T, V(t) \geq 0$, and abbreviate and $\phi \equiv (\mathrm{LV}(x) \leq v \leq \mathrm{UV}(x) \wedge x \geq 0 \wedge v \geq 0) \wedge \mathcal{M}_0 > \Delta x + (g - x)$. Note that $\mathcal{M}_0$ is Γ is always a real-valued variable because the condition $\mathcal{M} \succ \mathbf{0}$ is never satisfied when $\mathcal{M}$ is the special value $\mathbf{0}$, but can be satisfied when $\mathcal{M}$ is any number, including 0.

The first $\langle?\rangle$I step applies the lemma SB $\rightarrow \mathrm{acc} \in [-B, A]$. The second $\langle?\rangle$I step applies the lemma $\mathcal{D}_{\mathrm{arith1}}$. In the dsolve step, we write $X(t)$ and $V(t)$ for the solutions of x and v, where $X(t) = x + vt + \mathrm{acc}\frac{t^2}{2}$ and $V(t) = v + \mathrm{acc}\, t$. The domain constraint assumption in

the dsolve step has been simplified monotonically.

$$
\begin{array}{rl}
 & \mathcal{D}_{\text{arith2}} \\
\hline
 & \Gamma'(T/2) \vdash [(X(t),V(t))/(x,v)]\phi \\
\hline
[?]\text{I} & \Gamma'(0) \vdash [(X(t),V(t))/(x,v)][?t \geq T/2]\phi \\
\hline
\langle\!\langle d\rangle\!\rangle\text{I} & \Gamma'(0) \vdash [(X(t),V(t))/(x,v)]\langle ?t \geq T/2^d\rangle\phi \\
\hline
\langle ?\rangle\text{I} & \Gamma'(0) \vdash [(X(t),V(t))/(x,v)]\langle ?\mathsf{safe}\rangle\langle ?t \geq T/2^d\rangle\phi \\
\hline
\langle\!\langle ;\rangle\!\rangle\text{I} & \Gamma'(0) \vdash [(X(t),V(t))/(x,v)]\langle ?\mathsf{safe}; ?t \geq T/2^d\rangle\phi \\
\hline
\text{dsolve} & \Gamma, a = \text{acc}, t = 0 \vdash [x' = v, v' = a \,\&\, t \leq T \wedge v \geq 0]\langle ?\mathsf{safe}; ?t \geq T/2^d\rangle\phi \\
\hline
\langle\!\langle d\rangle\!\rangle\text{I} & \Gamma, a = \text{acc}, t = 0 \vdash \langle \mathsf{plant}\rangle\langle ?\mathsf{safe}; t \geq T/2^d\rangle\phi \\
\hline
\langle\!\langle ;\rangle\!\rangle\text{I} & \Gamma, a = \text{acc}, t = 0 \vdash \langle \mathsf{plant}; ?\mathsf{safe}; t \geq T/2^d\rangle\phi \\
\hline
\langle\!\langle :=\rangle\!\rangle\text{I} & \Gamma, a = \text{acc} \vdash \langle t := 0\rangle\langle \mathsf{plant}; ?\mathsf{safe}; t \geq T/2^d\rangle\phi \\
\hline
\langle\!\langle ;\rangle\!\rangle\text{I} & \Gamma, a = \text{acc} \vdash \langle t := 0; \mathsf{plant}; ?\mathsf{safe}; t \geq T/2^d\rangle\phi \\
\hline
\langle ?\rangle\text{I} & \Gamma, a = \text{acc} \vdash \langle ? - B \leq a \leq A\rangle\langle t := 0; \mathsf{plant}; ?\mathsf{safe}; t \geq T/2^d\rangle\phi \\
\hline
\langle\!\langle ;\rangle\!\rangle\text{I} & \Gamma, a = \text{acc} \vdash \langle ? - B \leq a \leq A; t := 0; \mathsf{plant}; ?\mathsf{safe}; t \geq T/2^d\rangle\phi \\
\hline
\langle :*\rangle\text{I} & \Gamma \vdash \langle a := *\rangle\langle ? - B \leq a \leq A; t := 0\rangle\langle \mathsf{plant}; ?\mathsf{safe}; t \geq T/2^d\rangle\phi \\
\hline
\langle\!\langle ;\rangle\!\rangle\text{I} & \Gamma \vdash \langle \mathsf{ctrl}\rangle\langle \mathsf{plant}; ?\mathsf{safe}; t \geq T/2^d\rangle\phi \\
\hline
\langle\!\langle ;\rangle\!\rangle\text{I} & \Gamma \vdash \langle \alpha\rangle\phi
\end{array}
$$

The remainder of the proof consists of derivations $\mathcal{D}_{\text{arith1}}$ and $\mathcal{D}_{\text{arith2}}$. To construct these derivations, we first prove a lemma $\mathcal{D}_{\text{arith3}}$:

$$\Gamma'(0) \vdash LV(X(t)) \leq V(t) \leq \text{UV}(X(t))$$

To prove $\mathcal{D}_{\text{arith3}}$, prove upper bound $V(T) \leq \text{UV}(X(T))$, then lower bound $\text{LV}(X(T)) \leq V(T)$. The upper bound holds by construction since acc is specifically chosen to remain within $\text{LV}(T)$. To show the lower bound, consider two regions which partition the safe envelope. Let Region 1 be bounded by SLS, LS, and SRS, while Region 2 is bounded by SRS, RS, and LS. Note that, in our strategy, case analysis on Region 1 vs. Region 2 is implicit in the comparisons min and max. We make this case analysis explicit in our proof for the sake of presentation clarity, despite the fact that explicit case analysis is not part of the CdGL proof term since it is not decidable which region(s) the state is in. Formally, the analysis of min and max given here all occurs within a single non-syntactic arithmetic proof step. It suffices in Region 1 to show acc $\geq a_{min}$ and in Region 2 to show acc $\geq -C$.

First consider an initial state $(X(0), V(0)) \in$ Region 1. The acceleration can be determined by first determining the distance $\delta X = X(T) - X(0)$ traversed in time T, not to be confused with the *global minimum* traversed distance Δx. The greatest elapsed distance is attained at upper-left point UL = LS $\wedge$ SRS where, by definition of $\text{BM}(T)$ we have $\delta X = (x_{mid} - \text{BM}(T))$. At this point, the acceleration acc $= C$ is clearly live. The next extremum is the upper-right point UR = SLS $\wedge$ SRS. Because a_{min} defines the curve SLS, any acceleration acc $\geq a_{min}$ is live from any point on SRS. Monotonicity shows that all other points are live because δX decreases with distance from SRS. That is, acc $\leq C \rightarrow \delta X \leq (x_{mid} - \text{BM}(T)) = (\text{AM}(T) - x_{mid})$, so that $X(T) \leq \text{AM}(T)$. Then acc increases as δX decreases. Since a_{min} is defined to yield $V(T) = \text{UV}(X(T))$ only at $X(T) = \text{AM}(T)$, then by monotonicity also $V(T) \leq \text{UV}(T)$.

From the UL and UR points, correctness of the entire Region 1 follows by additional monotonicity and continuity arguments. Any point between UL and UR has acc $\in [a_{min}, C]$ because the restriction of acc to this line is monotone. As we move toward the lower-left corner (LL), then acc can only increase: decreasing $V(0)$ or $X(0)$ frees us to be *less* conservative. Thus, acc $\geq a_{min}$ everywhere in Region 1 as desired.

From every point in Region 2, consider the simplistic braking rate a_{simp} which, if followed indefinitely, achieves $x = g$ exactly when $v = 0$. The trajectory of a_{simp} is clearly within Region 2 for any initial point in Region 2. The value a_{simp} is always in $[-C, -a_{min}]$ and so is not only physically achievable but is also live. Thus, concludes the proof of $\mathcal{D}_{\text{arith3}}$.

For $\mathcal{D}_{\text{arith1}}$ we prove $\Gamma, a = \text{acc}, 0 \leq t \leq T, V(t) \geq 0 \vdash [(X(t), V(t))/(x, v)]\mathsf{safe}$, i.e., we prove $\Gamma, a = \text{acc}, 0 \leq t \leq T, V(t) \geq 0 \vdash X(t) \leq g$, assuming $\mathcal{D}_{\text{arith3}}$. Since we already have $V(T) \leq \text{UV}(X(T)) \leq \text{RS}(X(T))$ then (SB) $\frac{V(T)^2}{2C} \leq (g - X(T))$. Since the LHS is trivially nonnegative, the RHS is nonnegative, i.e. $X(T) \leq g$ as desired.

We now prove ($\mathcal{D}_{\text{arith2}}$): $\Gamma, a = \text{acc}, T/2 \leq t \leq T, V(t) \geq 0 \vdash [(X(t), V(t))/(x, v)]\phi$ where we abbreviate: $\rho_1 = \text{LV}(x) \leq v \leq \text{UV}(x), \rho_2 = x \geq 0, \rho_3 = v \geq 0, \rho_4 = (\mathcal{M}_0 = (g - x) \geq 0), \rho_5 = a = \text{acc}, \rho_4 = t \in [T/2, T], \rho_7 = V(t) \geq 0$ and $\phi_1 = \text{LV}(X(t)) \leq V(t) \leq \text{LV}(X(t)), \phi_2 = X(t) \geq 0, \phi_3 = V(t) \geq 0, \phi_4 = \mathcal{M}_0 \succ \mathcal{M}$ so that the context $(\Gamma, a = \text{acc}, 0 \leq t \leq T, V(t) \geq 0) \leftrightarrow \bigwedge_i \rho_i$ and $[(X(t), V(t))/(x, v)]\phi = \bigwedge_i \phi_i$. We prove each conjunct ϕ_i. Conjunct ϕ_1 is already proven by $\mathcal{D}_{\text{arith3}}$. We prove ϕ_3 next, from which we then prove ϕ_2 as a corollary. We prove ϕ_4 last.

In each case except ϕ_4, it suffices to consider the case $t = T$ by monotonicity.

We prove ϕ_3 by hypothesis rule: assumption ρ_7 is the desired result.

We prove ϕ_2. We can prove it by the ODE solution or even more obviously by rule DI. From the domain constraint, $v \geq 0$ is an invariant. Then ρ_2 says $X(0) \geq 0$ and since $x' = v \geq 0$ then by rule DI we have $X(T) \geq 0$.

We prove ϕ_4. To prove progress, we must test whether we are in the "final" iteration of the loop. Recall that $\mathcal{M}$ is a choice function with a piecewise definition; we perform a case analysis where each branch tells us which piece of the piecewise definition of $\mathcal{M}$ applies. We perform an inexact (formula-level) comparison of $g - x$ against $\frac{a_{min}T^2}{16}$ with tolerance $\frac{a_{min}T^2}{16}$ so that we constructively have (compare) $g - x \leq \frac{a_{min}T^2}{8} \vee g - x \geq \frac{a_{min}T^2}{16}$. Expanding the definitions of $\mathcal{M}$ and $\succ$, formula ϕ_4 it suffices to show $(\mathcal{M}_0 \geq \Delta x + g - X(0))$ (for real-valued $\mathcal{M}_0$) before the final iteration and suffices to prove $g = X(t)$ in the final iteration ($\mathcal{M} = \mathbf{0}$). In the former case since $t \geq T/2$ we derive from construction of a and the initial velocity envelope that $\frac{V(0)^2}{2a} = (g - x)$ so by definition of $X(t)$ have $X(t) = g$ which satisfies the equation $g = X(t)$.

In the second case, not only does the test yield $g - X(0) \geq \frac{a_{min}T^2}{16}$, but when combined with the test $t \geq T/2$ it implies a stronger condition:

$$X(t) - X(0) \geq \frac{a_{min}T^2}{8}$$

which combined with safe braking entails $g - X(0) \geq \frac{a_{min}T^2}{8}$. Then

$$\begin{aligned} &\mathcal{M}_0 \geq \Delta x + (g - X(T)) \\ \text{iff } &g - X(0) \geq \Delta x + (g - X(T)) \\ \text{iff } &X(T) \geq \Delta x + X(0) \end{aligned}$$

We argue by monotonicity. The elapsed distance δX is minimized (i.e., $\delta X = \Delta x$) when velocity and acceleration are minimized, that is when $a = a_{min}$ and $x = 0$. In the worst case Demon chooses $t = T/2$, then Demon is responsible for satisfying the domain constraint $V(t) \geq 0$ and test $t \geq T/2$ simultaneously. Then by the Newton equations, $(X(T) - X(0)) \geq \delta X = a_{\min}\frac{T^2}{8}$, which exactly the definition of Δx as required.

This completes the last case of the arithmetic lemma, which in turn closes the final goal of the reach-avoid proof.

Though the proof is already complete, it is worth remarking on the interaction between test $t \geq T/2$, metric $\mathcal{M}$, and how the final iteration of a loop is handled. Both Angel and Demon face challenges in the final step of a loop: Angel may not be able to guarantee the full progress Δx, while Demon may not be able to satisfy the test $t \geq T/2$ without violating the domain constraint $v \geq 0$. Angel's dilemma is resolved with a disjunctive treatment of termination metrics: Angel is allowed to move an arbitrarily small distance (including none) if she can prove the loop stops after this iteration. Demon's dilemma can cause Demon to lose the game, but only in cases that *should* be Demon losses: in the final iteration, Angel may choose a braking rate which causes the system to stop in time $t < T/2$. In any reasonable practical implementation of this system[3], Demon will choose to fail the test $t < T/2$ and lose the game. However, Demon has only lost the game because Angel is so infinitesimally close to victory that Demon cannot recover: Angel is so close to winning that she can fairly be declared the winner[4].

[3]Looking ahead to the use of game strategies in monitoring (Chapter 8), Demon would encounter a plant monitor failure if he chooses large values of T because his model will predict $v < 0$ and a real system will exhibit $v \geq 0$ so long as it does not go into the reverse gear.

[4]From a practical perspective, if Angel stops an inch before her destination, she is generally considered to have reached the destination.

B.2 Theory Proofs

We prove the stated meta-theorems of CdGL such as monotonicity, soundness, the Existence Property, and the Disjunction Property.

B.2.1 Preliminaries and Assumptions

We first state preliminaries from the literature and assumptions.

Constructive ODEs. A difference between our soundness proof and that of dGL is that we draw on results of constructive analysis rather than classical analysis. The major results on which we rely have been proven in the literature, but we restate them here because the theorem statements are otherwise difficult to locate. The main catch in applying these results is that they are proven for time-derivatives, whereas our differentials are spatial. For this reason, we will prove Lemma B.7 equating time and space differentials within the context of an ODE, which justifies our usage of these existing results.

We restate[5] the definition of uniform continuity, a concept which will be used in the following theorem statements.

Definition B.1 (Uniform Continuity)**.** Let X and Y be metric spaces and let $f : X \to Y$. For any metric space S and any $\delta : \mathbb{Q}$ and $y_1, y_2 : S$, let $ball_S(\delta, y_1, y_2)$ be a predicate which holds iff the distance between y_1 and y_2 in metric space S is at most δ. Function f is uniformly continuous if there exists a function $\mu : \mathbb{Q} \to (\mathbb{Q} \cup \{\infty\})$ such that:

- $\forall \epsilon : \mathbb{Q}\ (0 < \epsilon \to 0 < \mu(\epsilon))$
- $\forall \epsilon : \mathbb{Q}\ \forall x_1 x_2 : X\ (0 < \epsilon \to ball_X(\mu(\epsilon), x_1, x_2) \to ball_Y(\epsilon, f(x_1), f(x_2)))$

That is, for each $\epsilon > 0$, there must constructively exist $\mu(\epsilon) > 0$ such that $\mu(\epsilon)$ that the distance between outputs is at most ϵ whenever the distance between inputs is at most $\mu(\epsilon)$. The function μ is a witness which constructs the bound on input distance as a function of output distance.

The functions and theorems referenced in our proof summary are also originally taken from the CoRN repository.

Theorem B.3 (Constructive Picard-Lindelöf)**.** *Picard-Lindelöf has been formalized in Coq. We restate it from CoRN*[6]*.*

Let $x_0 \in \mathbb{Q}$ and $y_0 \in \mathbb{R}$. Let $a \in \mathbb{Q}_{\geq 0}$ and $b \in \mathbb{R}_{\geq 0}$. Define the closed intervals $(X \subset \mathbb{Q}) = [x_0 - a, x_0 + a]$ and $(Y \subset \mathbb{R}) = [y_0 - b, y_0 + b]$. Let $v : X \times Y \to \mathbb{R}$ be effectively Lipschitz continuous with Lipschitz constant $L > 0$. Because v is a Lipschitz continuous function on a compact domain, it is bounded, i.e., there exists[7] *a bound M such*

[5] From `ode/metric.v` of the CoRN repository, which may be phrased different from elsewhere in the literature. See the bibliography entry for (Cruz-Filipe et al., 2004) for the URL and commit numbers we based our statements on.

[6] The statement of Picard-Lindelöf is in file `ode/Picard.v` of the CoRN repository. See the bibliography entry for (Cruz-Filipe et al., 2004) for the URL and relevant commit numbers.

[7] Because v is Lipschitz continuous, it is also uniformly continuous. Uniform continuity implies existence of an algorithm to compute M to arbitrary precision by sampling the domain at sufficient precision.

that $|v(x,y)| \leq M$ for all $x \in X, y \in Y$. Assume Y is large enough to contain the trajectory of the resulting solution throughout time domain X, specifically assume $aM \leq b$.

Then there exists a unique function $f : X \to Y$ that solves the initial value problem:

- $f(x_0) = y_0$
- $(f)'(x) = v(x, f(x))$

Summary. The proof relies on the existence for each v of the well-known *Picard* operator $\mathsf{picard}_v : (X \Rightarrow Y) \Rightarrow (X \Rightarrow Y)$ and the fact that this operator is contractive, let $q \in [0, 1)$ be the contraction factor. When contracted iteratively, the limit is the solution f of the IVP $(f)'(x, y) = v(x, y)$ with initial condition $f(x_0) = y_0$. The proof relies on the Banach fixed-point operator fp such that $\mathsf{fp}\ g\ g_0$ is a fixed point of g, computed as the limit of the sequence $g_{i+1} = g\ g_i$ starting from the given g_0. Specifically, define $g_0(t) = y_0$ and let $g = \mathsf{picard}_v$, then the solution of the IVP is $\mathsf{fp}\ g\ g_0$.

1. By the Banach fixed-point theorem, then $\mathsf{fp}\ \mathsf{picard}_v\ g_0$ is a fixed point such that
$$\mathsf{picard}_v\ (\mathsf{fp}\ \mathsf{picard}_v\ g_0) = (\mathsf{fp}\ \mathsf{picard}_v\ g_0)$$
2. $\mathsf{fp}\ \mathsf{picard}_v\ g_0$ is a solution of the IVP.
3. $\mathsf{fp}\ \mathsf{picard}_v\ g_0$ is constructive: its exact value is arbitrarily approximated by iterating the picard operator. □

The constructive proof of the Banach fixed-point theorem shows the rate of convergence is geometric in contraction factor q. The constructive proof of Picard-Lindelöf constructs the contraction factor as a function of the Lipschitz constant and interval size. The explicit convergence bound given by the Banach fixed-point theorem informs the computational interpretation of the constructed ODE solution. Recall that a function over constructive reals is provided an arbitrarily small desired output precision by its caller and must return an approximate result within that precision. When asked to evaluate the ODE solution with some precision $\epsilon > 0$, a computational interpretation of this proof could consult the contraction factor to compute a bound k such that k iterations of the Picard operator are within precision ϵ of the true solution, then return the result of k iterations of the Picard operator. Because convergence is geometric, the bound k is roughly proportional to the logarithm of ϵ in base q.

Unfortunately, the statement of Theorem B.3 is just slightly too demanding for our purposes, specifically the requirement $aM \leq b$ is too strong for our purposes. We assume correctness of the following, minor generalization which drops the requirement $aM \leq b$ in return for strengthening Y from a compact interval to the whole real line. The generalization also relaxes the time domain to allow real numbers, since this thesis uses real-valued time domains:

Theorem B.4 (Constructive Picard-Lindelöf, Unbounded Space Domain)**.** *Let $x_0 \in \mathbb{R}$ and $y_0 \in \mathbb{R}$. Let $a \in \mathbb{R}_{\geq 0}$. Define the sets $(X \subset \mathbb{R}) = [x_0 - a, x_0 + a]$ and $Y = \mathbb{R}$. Let $v : X \times Y \to \mathbb{R}$ be effectively Lipschitz continuous with respect to the second argument, with Lipschitz constant $L > 0$. Then there exists a unique function $f : X \to Y$ that solves the initial value problem:*

- $f(x_0) = y_0$
- $(f)'(x) = v(x, f(x))$

The classical counterpart to Theorem B.4 is well-known (Walter, 1998, §10.VII).

Theorem B.5 (Constructive Differential Induction (DI) Lemmas)**.** *The following statements are from CoRN*[8] *and are corollaries of the constructive Mean Value Theorem.*

Let $a, b : \mathbb{R}$ with $a < b$, let $I = [a, b]$. Let $\varepsilon : \mathbb{R} > 0$. Let $F, (F)', G, (G)' : I \Rightarrow \mathbb{R}$.

- *Lemma* (Feq-criterium) *supports equational DI. If $(F)' = (G)'$ on I and there exists $x \in I$ such that $F\ x = G\ x$, then $F = G$ on I.*
- *Lemma* (Derivative-imp-resp-less) *supports strict inequational DI. If $0 < (F)'$ on $[a, b]$ then $F\ a < F\ b$.*
- *Lemma* (Derivative-imp-resp-leEq) *supports nonstrict inequational DI. If $0 \leq (F)'$ on I, then $F\ a \leq F\ b$.*

The cases for $>$ and $\geq$, which are also proven by CoRN, are symmetric.

Theorem B.6 (Constructive DV Lemma)**.** *If $(G)' \geq d$ on $[a, b]$ for some constant $d > 0$, then $G(b) - G(a) \geq d(b - a)$.*

Proof. CoRN features a lemma *(Law-of-the-Mean-Abs-ineq)* which is almost our desired lemma for DV: If $(F)' \leq c$ on $[a, b]$ for some constant c, then $F(b) - F(a) \leq c(b - a)$. Assume (0) $(G)' \geq d$ on $[a, b]$ and (1) $d > 0$. Because (the full statement of) *(Law-of-the-Mean-Abs-ineq)* supports $c < 0$ and $b < a$ as well, it suffices to let $F = -G$ and $c = -d$, then from (0) and (1) have (2) $(F)' \leq c$ on $[a, b]$ so by *(Law-of-the-Mean-Abs-ineq)* have $F(b) - F(a) \leq c(b - a)$, thus $G(b) - G(a) \geq d(b - a)$ by definition of F and c as desired. □

B.2.2 Notations and Proof Style

We review and introduce notations for the proofs. We use renaming and substitution on states: the renamed state $s\frac{y}{x}$ is defined as $\mathsf{set}\ (\mathsf{set}\ s\ x\ (s\ y))\ y\ (s\ x)$. The substituted state s^f_x is defined as $\mathsf{set}\ s\ x\ (f\ s)$. State substitutions will be generalized to adjoint states in Lemma 5.7 and Lemma 5.8. For proof terms M, it is also useful to speak of the renaming $M\frac{y}{x}$ and substitution M^f_x. Rather than explicitly define these functions, we note that the constructive proofs of Lemma 5.2 and Lemma 5.7 constitute algorithms for computing $M\frac{y}{x}$ and M^f_x, respectively.

Because renaming and substitution are functions on proof terms, it is worth understanding the nature of computations over proof terms M, a nature which is explained by metatheoretic lemmas about CIC such as canonical forms. That is, canonical forms implies that computations over proof terms M are permitted to convert a proof to its canonical form and then inspect the contents of that canonical form. For our purposes, however, we did not need to make explicit use of metatheoretic lemmas about CIC such as canonical forms, rather the only computations we needed over M were those corresponding to application of standard inference rules and the packing and unpacking of definitions. Discussion of normal proof terms in Chapter 4 applies to CdGL proof terms (as opposed to their corresponding CIC proof terms) as well.

[8] See files `ftc/CalculusTheorems.v` and `ftc/Rolle.v` of CoRN (Cruz-Filipe et al., 2004).

Semantic proofs make heavy use of the simple function type $\tau_1 \Rightarrow \tau_2$, which models constructive semantic implication. We use if-and-only-if notation $\tau_1 \Leftrightarrow \tau_2$ for the product of $\tau_1 \Rightarrow \tau_2$ and $\tau_2 \Rightarrow \tau_1$. For the sake of readability and in order to more closely follow the style of classical semantics proofs, we *intentionally abuse notation* and just write τ when we mean to say "τ is inhabited," likewise writing $\tau_1 \Leftrightarrow \tau_2$ when we mean to say "τ_1 is inhabited iff τ_2 is inhabited". We do this because our proofs rely heavily on chain-of-equivalence reasoning which would become unreadably verbose if the judgment "is inhabited" were written on every line. For the same reason of readability, we do not explicitly write down the CIC term which inhabits each type on each line of a proof.

We write $\top$ for the unit type in CIC, which is used to interpret the formula *true*.

It is sometimes useful to talk of ODE solutions as yielding an entire state rather than the value for one variable. Thus, when *sol* is a scalar-valued solution function (of time) for some ODE, we abbreviate

$$Sol(t) = \mathsf{set}\ s\ (x, x')\ (sol\ t, f\ (\mathsf{set}\ s\ x\ (sol\ t)))$$

for a state-valued solution.

B.2.3 Static Semantics.

The proof calculus and soundness proofs rely on standard syntactic notions of free variables $\mathrm{FV}(e)$, bound variables $\mathrm{BV}(\alpha)$, and must-bound variables $\mathrm{MBV}(\alpha)$. First we discuss the syntactic treatment of static semantics, which we use for formulas and games. In the syntactic treatment, explicit recursive algorithms are given for free, bound, and must-bound variables. We then discuss the semantic versions of static semantics for formulas, games, and terms. In the semantic treatment, free variables, bound variables, and must-bound variables are defined as (least or greatest) sets that satisfy *coincidence* and *bound effect* properties: free variables influence meaning of expressions and non-bound variables do not change during execution. The semantic versions of formula and game static semantics are purely for the sake of comparison, while for terms we use the semantic treatment rather than syntactic. The use of a semantic treatment is necessary in the case of terms because CdGL terms are treated semantically in general. The section concludes by enumerating language assumptions about terms, both related to static semantics and otherwise.

Variable Set Computations. When restricted to the hybrid systems fragment, our syntactic calculations for game and formula static semantics agree exactly with the syntactic characterization used for dL (Platzer, 2017a). Semantic characterizations have also been given in the literature for dGL (Platzer, 2018b, 2019b) and dL (Platzer, 2017a). The syntactic definitions of the hybrid systems cases are presented in Fig. A.11 of Appendix A.4. The new cases are the cases for dual games, which all map through homomorphically:

$$\mathrm{FV}(\alpha^d) = \mathrm{FV}(\alpha)$$
$$\mathrm{BV}(\alpha^d) = \mathrm{BV}(\alpha)$$
$$\mathrm{MBV}(\alpha^d) = \mathrm{MBV}(\alpha)$$

In proving soundness, we will prove standard lemmas for these functions: *coincidence* says that only free variables influence the value of an expression, while *bound effect* says only bound variables can change value during execution. While the basic ideas of coincidence and bound effect are unchanged from Chapter 4, the new semantics demand new proofs and lemma statements.

We use the syntactic characterization for games and formulas in this appendix because it emphasizes effectivity, i.e., the definitions give an explicit algorithm for computing free and bound variables, which is useful because free and bound variable computations are used in checking applications of proof rules. Because we treat terms entirely semantically, their static semantics will be treated semantically as well. When we care about effectivity of term static semantics, we could additionally assume the existence of an algorithm which computes (a non-strict superset of) the static semantics.

Semantic Variable Computations. While we use syntactic computations for static semantics of games and formulas, we give the semantic versions for the sake of comparison. At the same time, we will introduce our free variable definition for terms, which is necessarily semantic because CdGL terms are treated semantically. We briefly describe the semantic versions of each variable set (note the different font) $\mathsf{FV}(e), \mathsf{BV}(\alpha)$, and $\mathsf{MBV}(\alpha)$ based directly on the coincidence and bound effect properties. In these definitions, we write $s \overset{V}{=} t$ to abbreviate for $\star_{x \in V}(s\ x = t\ x)$ where V is a finite variable set. We write $S^{\complement}$ for the complement of set S and we let $s, t, \hat{s}, \hat{t} : \mathfrak{S}$ and $P : (\mathfrak{S} \Rightarrow \mathbb{T})$.

For an expression e (term f, formula ϕ, or game α), the semantic free variables $\mathsf{FV}(e)$ are those which can influence the meaning of e, equivalently the smallest variable set V where states agreeing on V give equal values to f. The semantic bound variables $\mathsf{BV}(\alpha)$ of games are the complement of the set of preserved variables, which are provably equal to their initial values regardless of the postcondition. Variable computations for games must include both Angelic and Demonic plays. Consider the games $x := w \cup y := z$ and $x := w \cap y := z$: in each case the bound variables are $\{x, y\}$ and free variables are $\{w, z\}$, but in each case a different player has control over the choice, meaning that their must-bound variables, which can also influence the free variables of a surrounding game modality, differ.

In each of our semantic definitions, the high-level idea is to capture the desired lemmas (coincidence and bound effect) as types.

$$\mathsf{FV}(f) = \bigcap \{V \mid \text{for all } s, t : s \overset{V}{=} t.\ f\ s = f\ t\}$$

$$\mathsf{FV}(\phi) = \bigcap \{V \mid \text{for all } s, t : s \overset{V}{=} t.\ \ulcorner\phi\urcorner\ s = \ulcorner\phi\urcorner\ t\}$$

$$\mathsf{FV}(\alpha) = \left(\bigcap \{V \mid \text{for all } P, s, t : s \overset{V}{=} t.\ \langle\!\langle\alpha\rangle\!\rangle\ P\ s \Leftrightarrow \langle\!\langle\alpha\rangle\!\rangle\ P\ t\}\right) \cup \left(\bigcap \{V \mid \text{for all } P, s, t : s \overset{V}{=} t.\ [[\alpha]]\ P\ s \Leftrightarrow [[\alpha]]\ P\ t\}\right)$$

$$\mathsf{BV}(\alpha) = \left(\left(\bigcup \{V \mid \text{for all } P, s : \langle\!\langle\alpha\rangle\!\rangle\ P\ s.\ \langle\!\langle\alpha\rangle\!\rangle\ (\lambda t.\ P\ t\star(s \overset{V}{=} t))\ s\}\right) \cap \left(\bigcup \{V \mid \text{for all } P, s : [[\alpha]]\ P\ s.\ [[\alpha]]\ (\lambda t.\ P\ t\star(s \overset{V}{=} t))\ s\}\right)\right)^{\complement}$$

The semantic free variables are never more than the syntactic free variables, but sometimes are a strict subset.

A semantic treatment does not strictly need a definition of must-bound variables, because they are mostly used to provide a less conservative definition of *syntactic* free variables. We give a definition for the sake of comparison. To define must-bound variables, we run the game twice from states that agree on the free variables. A set V is semantically must-bound if its variables always agree in the final states, even for differing initial states. While it is common for semantic and syntactic characterizations of the same high-level concept to disagree (e.g., one may be more conservative than the other) this definition of must-bound variables both over-approximates and under-approximates syntactic must-bound variables: free variables that are never written are counted as must-bound, while variables which are always written with no-ops are counted as not must-bound. That being said, our definition captures the conditions needed for a formula coincidence guarantee.

$$\begin{aligned}\mathsf{MBV}(\alpha) = &\bigcup \Big(\{V \mid \text{for all } s,t : s \overset{\mathsf{FV}(\alpha)}{=} t.\ \langle\!\langle\alpha\rangle\!\rangle\ (\lambda \hat{s}.\ \langle\!\langle\alpha\rangle\!\rangle\ (\lambda \hat{t}.\ \hat{s} \overset{V}{=} \hat{t})\ t)\ s\}\Big) \\ &\cap \bigcup \Big(\{V \mid \text{for all } s,t : s \overset{\mathsf{FV}(\alpha)}{=} t.\ [[\alpha]]\ (\lambda \hat{s}.\ [[\alpha]]\ (\lambda \hat{t}.\ \hat{s} \overset{V}{=} \hat{t})\ t)\ s\}\Big)\end{aligned}$$

Term Language Assumptions. We assume there exist free variable, renaming, and substitution functions for semantic terms which satisfy standard lemmas. In any reasonable type theory, these functions should be expected to exist: they are modest extensions of the type theory's own free variable, renaming, and substitution functions. The only difference is that rather than considering a single variable $s : \mathfrak{S}$, one considers every use of $s\ x$ and every use of $\mathsf{set}\ s\ x\ v$.

Assumption 1 (Term coincidence). There exists an effective function $\mathrm{FV}(\cdot)$ such that for all s, t, f, if $s \overset{\mathrm{FV}(f)}{=} t$ then $f\ s = f\ t$.

Assumption 2 (Term renaming). There exists an effective function which implements uniform renaming by transposition on terms ($f\frac{y}{x}$). That is, for all x, y there exists a function $\cdot\frac{y}{x}$ on terms such that for all s, f, $(f\frac{y}{x})\ s = f(s\frac{y}{x})$.

Assumption 3 (Term substitution). There exists an effective function which implements program variable substitutions on terms ($\sigma(f)$). That is, if σ is admissible for f then $(\sigma(f))\ s = f\ (\sigma^*_s(s))$ where $\sigma(\cdot)$ is the effective substitution function for substitution σ. Recall that in the special case of singleton substitutions, we write f^g_x for the replacement of x by g in f. In the singleton case and notation, the substitution assumption on terms specializes to $(f^g_x)\ s = f(\mathsf{set}\ s\ x\ (g\ s))$.

B.2.4 Proofs of Stated Results

For semantic proofs about the inhabitance of types, we do not explicitly write out the proof terms which inhabit each type, since the proof terms are obvious from our proofs-by-type-rewriting.

We restate the definition of differential terms for reference:

Definition 5.3 (Differential term semantics)**.** We virtually define the differential term $(f)'$ by defining when it equals a given term g in a given state s:

$$\begin{aligned}((f)'\ s = g\ s) \equiv\ & \Sigma\nabla : \mathbb{R}^{|s'|}.\ (g\ s = \nabla{\cdot}s')\star\Pi\varepsilon : \mathbb{R}_{>0}.\ \Sigma\delta : \mathbb{R}_{>0}.\ \Pi r : \mathfrak{S}.\\ & (\|r - s\| < \delta) \Rightarrow |f\ r - f\ s - \nabla{\cdot}(r - s)| \leq \varepsilon\|r - s\|\end{aligned}$$

Lemma B.7 (Differential Lemma)**.** *Assume (A1)* $(sol, s, d \vDash x' = f)$ *and (A2)* $t \in [0, d]$ *and (A3)* $\mathrm{FV}(g) \subseteq \{x\}$*. Assume (A4) that* g *is differentiable. Recall that we define* $Sol(t) =$ *set* $s\ (x, x')\ (sol\ t, f\ (\textsf{set}\ s\ x\ (sol\ t)))$*. Then* $(g)'\ (Sol(t)) = \frac{\mathrm{d}}{\mathrm{d}r}(g\ (Sol(r)))\ (t)$*.*

Proof. We write $Sol'(t)$ for $\textsf{set}\ (s')\ x'\ (f\ (\textsf{set}\ s\ x\ (sol\ t)))$, i.e., the differential part of $Sol(t)$. From expanding the definition of a solution in (A1) we have (SolDer) $((sol)'\ t = f\ (\textsf{set}\ s\ x\ (sol\ t)))$ for all times $t \in [0, d]$.

Note for all $t \in [0, d]$ that (SolX)

$$\begin{aligned}& Sol'(t)(x) && \\ &= f\ (\textsf{set}\ s\ x\ (sol\ t)) && \text{Definition of Sol'} \\ &= \frac{\mathrm{d}}{\mathrm{d}t}\big(sol(t)\big)\ (t) && \text{(SolDer)} \\ &= (Sol(\cdot)(x))'\ (t) && \text{Definition of Sol}\end{aligned}$$

where notation $(Sol(\cdot)(x))'(t)$ indicates the derivative of the x component with respect to the time argument, at time t. Thus, $Sol'(t) = (Sol(t))'$ on $\{x\}$ where $\{x\} \supseteq \mathrm{FV}(g)$ by (A3). Then by Assumption 1 have (GSol) $g(Sol'(t)) = g((Sol(t))')$.

By assumption (A4), term g is differentiable, so by the semantics of $(g)'$ we have (Eq1)

$$(g)'\ (Sol(t)) = \nabla_{(g,t)}{\cdot}(Sol'(t))$$

for some $\nabla_{(g,t)} : \mathbb{R}^{|s'|}$ such that (GradDef) $\Pi\varepsilon : \mathbb{R}_{>0}.\ \Sigma\delta : \mathbb{R}_{>0}.\ \Pi r : \mathfrak{S}.\ (\|r - Sol(t)\| < \delta) \Rightarrow |f\ r - f\ (Sol(t)) - \nabla_{(g,t)}{\cdot}(r - Sol(t))| \leq \varepsilon\|r - Sol(t)\|$ is inhabited. By definition of gradient, fact (GradDef) says $\nabla_{(g,t)}$ is the gradient of g at $Sol(t)$ (GradAt). Then by the chain rule have (Eq2) $\nabla_{(g,t)}{\cdot}Sol'(t) =_{(\mathrm{GSol})} (g{\circ}Sol)'(t)$ and lastly (Eq3) $(g{\circ}Sol)'(t) =_{(\mathrm{GradAt})} \frac{\mathrm{d}}{\mathrm{d}t}\big(g(Sol(t))\big)(t)$ by expanding $(g \circ Sol)(t) = g(Sol(t))$. The result follows by transitivity on (Eq1), (Eq2), and (Eq3). □

The basic structure of the proof of Lemma B.7 is the same as the differential lemma in dL (Platzer, 2017a, Lem. 35). The first main difference is that an explicit differentiability assumption (A4) is necessary in CdGL because our term language admits non-differentiable terms, while the dL counterpart in the literature (Platzer, 2017a, Lem. 35) assumes a smooth term language. The second difference is that the dL uniform substitution calculus defines the semantics of differentials using a sum of partial derivatives which is then transformed to a gradient in the proof, while CdGL defines the semantics using a gradient and can thus skip the transformation step in the proof. Lastly, the dL semantics of ODEs use a solution function which computes the entire state, while the CdGL semantics use a function *sol* which only computes the bound base variable(s) of the ODE, thus the state-level solution $Sol(t)$ has to be treated as a defined construct in our proof.

In practice, assumption (A3) is not a limitation. Rather, before applying a rule which relies on this differential lemma, one would apply a step that locally transforms any additional program variables y into constants, which ensures their derivatives are 0 as intended, also fulfilling the requirement of this lemma.

Lemma 5.1 (Monotonicity)**.** *Let $P, Q : \mathfrak{S} \to \mathbb{T}$ and $s : \mathfrak{S}$. Note that in this lemma, P and Q are* any *inhabitants of $\mathfrak{S} \to \mathbb{T}$, as opposed to only those of form $\ulcorner\phi\urcorner$. If $\Gamma, P\ s \vdash Q\ s$ is inhabited, then $\Gamma, \langle\!\langle\alpha\rangle\!\rangle\ P\ s \vdash \langle\!\langle\alpha\rangle\!\rangle\ Q\ s$ and $\Gamma, [[\alpha]]\ P\ s \vdash [[\alpha]]\ Q\ s$ are inhabited.*

Proof. In each case, assume (0) $\Gamma, P\ s \vdash Q\ s$. Then fix context (1) ($\Gamma, \langle\!\langle\alpha\rangle\!\rangle\ P\ s$) or ($\Gamma, [[\alpha]]\ P\ s$) to show $\langle\!\langle\alpha\rangle\!\rangle\ Q\ s$ or $[[\alpha]]\ Q\ s$ accordingly. We annotate a step with subscript 0 when its justification is fact (0), likewise for other facts.

The Angel and Demon cases are proven by simultaneous induction, of which we list the Angel cases first.

Angel cases:

Case $x := f$: Have $\langle\!\langle x := f\rangle\!\rangle\ P\ s \Leftrightarrow P\ (\mathsf{set}\ s\ x\ (f\ s)) \Rightarrow_0 Q\ (\mathsf{set}\ s\ x\ (f\ s)) \Leftrightarrow \langle\!\langle x := f\rangle\!\rangle\ Q\ s$ where step (0) instantiates $\mathfrak{s}$ to $\mathsf{set}\ \mathfrak{s}\ x\ (f\ s)$.

Case $x := *$: Have $\langle\!\langle x := *\rangle\!\rangle\ P\ s \Leftrightarrow \Sigma v : \mathbb{R}.\, (P\ (\mathsf{set}\ s\ x\ v))$. Let v such that $(P\ (\mathsf{set}\ s\ x\ v))$, which exists by (1). Then by instantiating $\mathfrak{s}$ to $\mathsf{set}\ s\ x\ (f\ s)$ in (0), have $Q\ (\mathsf{set}\ s\ x\ v)$, and picking the same v, have $\Sigma v : \mathbb{R}.\, (Q\ (\mathsf{set}\ s\ x\ v)) \Leftrightarrow \langle\!\langle x := *\rangle\!\rangle\ Q\ s$.

Case $?\phi$: Have $\langle\!\langle ?\phi\rangle\!\rangle\ P\ s \Leftrightarrow \ulcorner\phi\urcorner\ s * P\ s \Rightarrow_0 \ulcorner\phi\urcorner\ s * Q\ s \Leftrightarrow \langle\!\langle ?\phi\rangle\!\rangle\ Q\ s$.

Case $x' = f\,\&\,\psi$: Have

$$
\begin{aligned}
& \langle\!\langle x' = f\,\&\,\psi\rangle\!\rangle\ P\ s \\
\Leftrightarrow\ & \Sigma d : \mathbb{R}_{\geq 0}.\, \Sigma sol : [0,d] \Rightarrow \mathbb{R}. \\
& \quad (sol, s, d \vDash x' = f) \\
& \quad *(\Pi t : [0,d].\, \ulcorner\psi\urcorner\ (\mathsf{set}\ s\ x\ (sol\ t))) \\
& \quad * P\ (\mathsf{set}\ s\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d))))
\end{aligned}
$$

Then unpack d and sol such that $(sol, s, d \vDash x' = f)$ and $(\Pi t : [0,d].\, P\ (\mathsf{set}\ s\ x\ (sol\ t)))$ so that $P\ (\mathsf{set}\ s\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d))))$.

Instantiate $\mathfrak{s}$ to $(\mathsf{set}\ s\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d))))$ in fact (0) in order to get $Q\ (\mathsf{set}\ s\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d))))$ so

$$
\begin{aligned}
& \Sigma d : \mathbb{R}_{\geq 0}.\, \Sigma sol : [0,d] \Rightarrow \mathbb{R}. \\
& \quad (sol, s, d \vDash x' = f) \\
& \quad *(\Pi t : [0,d].\, \ulcorner\psi\urcorner\ (\mathsf{set}\ s\ x\ (sol\ t))) \\
& \quad * Q\ (\mathsf{set}\ s\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d)))) \\
\Leftrightarrow\ & \langle\!\langle x' = f\,\&\,\psi\rangle\!\rangle\ Q\ s
\end{aligned}
$$

Case $\alpha;\beta$: Have $\langle\!\langle\alpha;\beta\rangle\!\rangle\ P\ s \Leftrightarrow \langle\!\langle\alpha\rangle\!\rangle\ (\langle\!\langle\beta\rangle\!\rangle\ P)\ s$ (2). Note by IH on β that (3) $\Gamma, \langle\!\langle\beta\rangle\!\rangle\ P\ s \vdash \langle\!\langle\beta\rangle\!\rangle\ Q\ s$. Then (2) and (3) suffice to apply the IH on α, giving $\langle\!\langle\alpha\rangle\!\rangle\ (\langle\!\langle\beta\rangle\!\rangle\ Q)\ s \Leftrightarrow \langle\!\langle\alpha;\beta\rangle\!\rangle\ Q\ s$.

Case $\alpha \cup \beta$: Have $\langle\!\langle\alpha \cup \beta\rangle\!\rangle\ P\ s \Leftrightarrow (\langle\!\langle\alpha\rangle\!\rangle\ P\ s + \langle\!\langle\beta\rangle\!\rangle\ P\ s) \underset{\text{IH}}{\Rightarrow} (\langle\!\langle\alpha\rangle\!\rangle\ Q\ s + \langle\!\langle\beta\rangle\!\rangle\ Q\ s) \Leftrightarrow \langle\!\langle\alpha \cup \beta\rangle\!\rangle\ Q\ s$.

Case α^*: Have $\langle\!\langle \alpha^* \rangle\!\rangle\ P\ s \Leftrightarrow (\mu\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\, \lambda t : \mathfrak{S}(P\ t \Rightarrow \tau'\ t) + (\langle\!\langle \alpha \rangle\!\rangle\ \tau'\ t \Rightarrow \tau'\ t))\ s$. Note that (0) holds when instantiating $\mathfrak{s}$ to any t so that

$$\begin{aligned}
&(\mu\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\, \lambda t : \mathfrak{S}\,(P\ t \Rightarrow \tau'\ t) + (\langle\!\langle \alpha \rangle\!\rangle\ \tau'\ t \Rightarrow \tau'\ t))\ s \\
\Rightarrow &(\mu\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\, \lambda t : \mathfrak{S}\,(Q\ t \Rightarrow \tau'\ t) + (\langle\!\langle \alpha \rangle\!\rangle\ \tau'\ t \Rightarrow \tau'\ t))\ s \\
\Leftrightarrow &\langle\!\langle \alpha^* \rangle\!\rangle\ Q\ s
\end{aligned}$$

Case α^d: Have $\langle\!\langle \alpha^d \rangle\!\rangle\ P\ s \Leftrightarrow [[\alpha]]\ P\ s \underset{\text{IH}}{\Rightarrow} [[\alpha]]\ Q\ s \Leftrightarrow \langle\!\langle \alpha^d \rangle\!\rangle\ Q\ s$ where the step marked IH employs the IH from the simultaneous IH on Demonic games, which applies because α is structurally smaller than α^d.

Demon cases:

Case $x := f$: Have $[[x := f]]\ P\ s \Leftrightarrow P\ (\mathsf{set}\ s\ x\ (f\ s)) \Rightarrow_0 Q\ (\mathsf{set}\ s\ x\ (f\ s)) \Leftrightarrow [[x := f]]\ Q\ s$ where step 0 instantiates $\mathfrak{s}$ to $\mathsf{set}\ s\ x\ (f\ x)$.

Case $x := *$: Have $[[x := *]]\ P\ s \Leftrightarrow \Pi v : \mathbb{R}.\, (P\ (\mathsf{set}\ s\ x\ v))$, so (2) $(\Gamma, [[x := *]]\ P\ s, v : \mathbb{R} \vdash P\ (\mathsf{set}\ s\ x\ v))$ and, instantiating $\mathfrak{s}$ to $\mathsf{set}\ s\ x\ v$ in (0), have (2) $\Gamma, [[x := *]]\ P\ s, v : \mathbb{R} \vdash (Q\ (\mathsf{set}\ s\ x\ v))$, thus $\Gamma, [[x := *]]\ P\ s \vdash \Pi v : \mathbb{R}.\, (Q\ (\mathsf{set}\ s\ x\ v)) \Leftrightarrow \Gamma, [[x := *]]\ P\ s \vdash [[x := *]]\ Q\ s$.

Case $?\psi$: Have $[[?\psi]]\ P\ s \Leftrightarrow (\ulcorner \psi \urcorner\ s \Rightarrow P\ s)$ (2) so it suffices to show $\Gamma, (\ulcorner \psi \urcorner\ s \Rightarrow P) \vdash (\ulcorner \psi \urcorner\ s \Rightarrow Q\ s) \Leftarrow \Gamma, (\ulcorner \psi \urcorner\ s \Rightarrow P\ s), \ulcorner \psi \urcorner\ s \vdash Q\ s \Leftarrow \Gamma, P\ s \vdash Q\ s$ where the first step is by implication introduction, the second is by implication elimination, and the last step is assumed true by (0).

Case $x' = f\,\&\,\psi$: Have

$$\begin{aligned}
&[[x' = f\,\&\,\psi]]\ P\ s \\
\Leftrightarrow &\Pi d : \mathbb{R}_{\geq 0}.\, \Pi sol : [0, d] \Rightarrow \mathbb{R}. \\
&\quad (sol, s, d \vDash x' = f) \\
&\Rightarrow (\Pi t : [0, d].\, \ulcorner \psi \urcorner\ (\mathsf{set}\ s\ x\ (sol\ t))) \\
&\Rightarrow P\ (\mathsf{set}\ s\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d))))
\end{aligned}$$

Then for arbitrary d and sol assume $(sol, s, d \vDash x' = f)$ and $(\Pi t : [0, d].\, P\ (\mathsf{set}\ s\ x\ (sol\ t)))$, so that $P\ (\mathsf{set}\ s\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d))))$. We instantiate the state variable $\mathfrak{s}$ to $(\mathsf{set}\ s\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d))))$ in (0), and so $Q\ (\mathsf{set}\ s\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d))))$, so

$$\begin{aligned}
&(\Pi d : \mathbb{R}_{\geq 0}.\, \Pi sol : [0, d] \Rightarrow \mathbb{R}. \\
&\quad (sol, s, d \vDash x' = f) \\
&\Rightarrow (\Pi t : [0, d].\, \ulcorner \psi \urcorner\ (\mathsf{set}\ s\ x\ (sol\ t))) \\
&\Rightarrow Q\ (\mathsf{set}\ s\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d))))) \\
\Leftrightarrow &[[x' = f\,\&\,\psi]]\ Q\ s
\end{aligned}$$

Case $\alpha; \beta$: Have $[[\alpha; \beta]]\ \ulcorner P\ s \urcorner \Leftrightarrow [[\alpha]]\ ([[\beta]]\ P)\ s$, call this fact (2). Note by IH on β that (3) $\Gamma, [[\beta]]\ P\ s \vdash [[\beta]]\ Q\ s$. Then (2) and (3) suffice to apply the IH on α, giving $[[\alpha]]\ ([[\beta]]\ Q)\ s \Leftrightarrow [[\alpha; \beta]]\ Q\ s$.

Case $\alpha \cup \beta$: Have $[[\alpha \cup \beta]]\ P\ s \Leftrightarrow ([[\alpha]]\ P\ s \mathbin{\star} [[\beta]]\ P\ s) \underset{\text{IH}}{\Rightarrow} ([[\alpha]]\ Q\ s \mathbin{\star} [[\beta]]\ Q\ s) \Leftrightarrow [[\alpha \cup \beta]]\ Q\ s$.

Case α^*: Have $[[\alpha^*]]\ P\ s \Leftrightarrow (\rho\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}\,(\tau'\ t \Rightarrow [[\alpha]]\ \tau'\ t) \mathbin{\star} (\tau'\ t \Rightarrow P\ t))\ s$. Since (0) holds when instantiating $\mathfrak{s}$ to any t, then have

$$\begin{aligned}&(\rho\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda s' : \mathfrak{S}(\tau'\ t \Rightarrow [[\alpha]]\ \tau'\ t) \mathbin{\star} (\tau'\ t \Rightarrow P\ t))\ s\\ \Rightarrow&(\rho\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}(\tau'\ t \Rightarrow [[\alpha]]\ \tau'\ t) \mathbin{\star} (\tau'\ t \Rightarrow Q\ t))\ s\\ \Leftrightarrow&[[\alpha^*]]\ Q\ s\end{aligned}$$

Case α^d: Have $[[\alpha^d]]\ P\ s \Leftrightarrow \langle\langle\alpha\rangle\rangle\ P\ s \underset{\text{IH}}{\Rightarrow} \langle\langle\alpha\rangle\rangle\ Q\ s \Leftrightarrow [[\alpha^d]]\ Q\ s$ where the IH is from the simultaneous induction on Angelic games. □

The static semantics results are stated without proof in the main text for the sake of brevity. We give full proofs here. The coincidence lemmas for formulas, Angelic games, and Demonic games are proven by simultaneous induction. We also include results for contexts, which are simply finite conjunctions of formulas, and write $\ulcorner\Gamma\urcorner(\mathfrak{s})$ to mean the product of $\ulcorner\phi\urcorner\ s$ for $\phi \in \Gamma$ with typing assumption $\mathfrak{s} : \mathfrak{S}$ for a distinguished state. The same holds of the renaming and substitution lemmas.

Note that we state coincidence and bound effect for games differently from prior work (Platzer, 2018b) to avoid introducing extra notations used in prior work.

Specifically, prior work (Platzer, 2018b) uses a notation $X{\downarrow}\omega(V)$ for the (dGL) subregion of X containing states ν that agree with ω on V. That bound effect lemma from that work (Platzer, 2018b) shows that (any) state ω belongs to winning region of (any) game α with (any) goal region X iff it belongs to the initial region of the same game when the goal region is $X{\downarrow}\omega(\mathrm{BV}(\alpha)^C)$. In short, the bound effect lemma in both dGL and CdGL says that when playing a game α, final states agree with initial states on variables in $\mathrm{BV}(\alpha)^C$, meaning variables which are not bound. We just happen to choose to give a direct description of the final region of the bound effect property as opposed to introducing the auxiliary notation used in dGL.

Lemma 5.4 (Formula Coincidence)**.** *Let $s,t : \mathfrak{S}$ and let $V \supseteq \mathrm{FV}(\phi)$ be a set of variables. Assume $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash s \overset{V}{=} t$, meaning s and t assign equal values to each variable in V. Assume $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash (\ulcorner\phi\urcorner\ s)$, then $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash (\ulcorner\phi\urcorner\ t)$.*

Coincidence for contexts also holds. Note that the states s and t at which we consider the conclusion context Γ_2 need not be the distinguished state $\mathfrak{s}$ of the context Γ_1. The claim can also be generalized to allow an arbitrary state argument for Γ_1 as a consequence by applying it again with $\Gamma_2 = \Gamma_1$.

If $\ulcorner\Gamma_1\urcorner(\mathfrak{s}) \vdash s \overset{V}{=} t$ for $V \supseteq \mathrm{FV}(\Gamma_2)$ then $\ulcorner\Gamma_1\urcorner(\mathfrak{s}) \vdash \ulcorner\Gamma_2\urcorner(s)$ is inhabited iff $\ulcorner\Gamma_1\urcorner(\mathfrak{s}) \vdash \ulcorner\Gamma_2\urcorner(t)$ is inhabited.

Coincidence for the construct $(sol, s, d \vDash x' = f)$ also holds: If $s \overset{\mathrm{FV}(f)\cup\{x\}}{=} t$ then $(sol, s, d \vDash x' = f) \Leftrightarrow (sol, t, d \vDash x' = f)$.

Proof. The formula and context cases are proven by simultaneous induction with one another and with Lemma 5.5.

Context cases:
Case $\Gamma_2 = \cdot$ where $\cdot$ is the empty context: This case holds trivially because $\ulcorner\cdot\urcorner(s)$ and $\ulcorner\cdot\urcorner(t)$ are both inhabited for all t, witnessed by the unit tuple.

Case $\Gamma_2 = (\Gamma, \psi)$: Then (A0) $\ulcorner\Gamma, \psi\urcorner(s) = (\ulcorner\Gamma\urcorner(s), \ulcorner\psi\urcorner\ s)$. Note $\mathrm{FV}(\Gamma, \psi) = \mathrm{FV}(\Gamma) \cup \mathrm{FV}(\psi)$ as required to apply the IHs. Apply the IH on smaller context Γ to get $\ulcorner\Gamma\urcorner(s) \Leftrightarrow \ulcorner\Gamma\urcorner(t)$ and apply the formula IH to get (1) $\ulcorner\psi\urcorner\ s \Leftrightarrow \ulcorner\psi\urcorner\ t$. Then from (1) and the right conjunct of (A0) have $\ulcorner\psi\urcorner\ t$, then with the left conjunct of (A0) have $(\ulcorner\Gamma\urcorner(t), \ulcorner\psi\urcorner\ t) = \ulcorner(\Gamma, \psi)\urcorner(t)$ as desired.

Formula cases:
Every formula case begins with the same steps, which we present once here to avoid repetition: Each case assumes for some formula ϕ that (A1) $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash \ulcorner\phi\urcorner\ s$ holds to prove $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash \ulcorner\phi\urcorner\ t$, which it proves by assuming (A2) $\ulcorner\Gamma\urcorner(\mathfrak{s})$ to prove $\ulcorner\phi\urcorner\ s$. In each case, apply modus ponens on (A1) and (A2) to get (A3) $\ulcorner\phi\urcorner\ s$. In reading the following cases, assume assumptions (A1) and (A2) have already been introduced and modus ponens already applied so that (A1), (A2), and (A3) are available as assumptions.

Case $\langle\alpha\rangle\phi$: The case proves by transitivity: $\ulcorner\langle\alpha\rangle\phi\urcorner\ s \Leftrightarrow \langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\phi\urcorner\ s \underset{\text{IH}}{\Leftrightarrow} \langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\phi\urcorner\ t \Leftrightarrow \ulcorner\langle\alpha\rangle\phi\urcorner\ t$ where the IH applies because $\mathrm{FV}(\langle\alpha\rangle\phi) = \mathrm{FV}(\alpha) \cup (\mathrm{FV}(\phi) \setminus \mathrm{MBV}(\alpha))$ and the assumption for the game IH (Lemma 5.5) is that $s \overset{V}{=} t$ for $V \supseteq \mathrm{FV}(\alpha) \cup (\mathrm{FV}(\phi) \setminus \mathrm{MBV}(\alpha))$ where we already have $s \overset{V}{=} t$ for $V \supseteq \mathrm{FV}(\langle\alpha\rangle\phi)$ as an assumption of this case.

Case $[\alpha]\phi$: The case proves by transitivity: $\ulcorner[\alpha]\phi\urcorner\ s \Leftrightarrow [[\alpha]]\ \ulcorner\phi\urcorner\ s \underset{\text{IH}}{\Leftrightarrow} [[\alpha]]\ \ulcorner\phi\urcorner\ t \Leftrightarrow \ulcorner[\alpha]\phi\urcorner\ t$ where the IH applies because $\mathrm{FV}([\alpha]\phi) = \mathrm{FV}(\alpha) \cup (\mathrm{FV}(\phi) \setminus \mathrm{MBV}(\alpha))$ and the assumption for the game IH (Lemma 5.5) is that $s \overset{V}{=} t$ for $V \supseteq \mathrm{FV}(\alpha) \cup (\mathrm{FV}(\phi) \setminus \mathrm{MBV}(\alpha))$ where we already have $s \overset{V}{=} t$ for $V \supseteq \mathrm{FV}([\alpha]\phi)$ as an assumption of this case.

Case $f \sim g$: Because $\mathrm{FV}(f \sim g) = \mathrm{FV}(f) \cup \mathrm{FV}(g)$ then by Assumption 1 (term coincidence) have $f\ s = f\ t$ and $g\ s = g\ t$ from which we derive $\ulcorner f \sim g\urcorner\ s \Leftrightarrow (f\ s \sim g\ s) \Leftrightarrow (f\ t \sim g\ t) \Leftrightarrow \ulcorner f \sim g\urcorner\ t$.

Case $(sol, s, d \vDash x' = f)$: Note (0) $(s\ x = sol\ 0)$ iff $(t\ x = sol\ 0)$ since $s \overset{\{x\}}{=} t$. Also, $s \overset{\mathrm{FV}(f)\setminus\{x\}}{=} t$ so by Assumption 1 (term coincidence) have (1) $((y)'\ r = f\ (\mathsf{set}\ s\ x\ (sol\ r)))$ iff $((sol)'\ r = f\ (\mathsf{set}\ t\ x\ (sol\ r)))$ for all $r \in [0, d]$. Next, by Assumption 1 (term coincidence) have (2) $f\ (\mathsf{set}\ s\ x\ (sol\ r)) = f\ (\mathsf{set}\ t\ x\ (sol\ r))$ since $s \overset{\mathrm{FV}(f)}{=} t$.

Then by (0), (1), (2) have $(sol, s, d \vDash x' = f) \Leftrightarrow ((s\ x = sol\ 0) \star (\Pi r : [0, d].\ ((sol)'\ r = f\ (\mathsf{set}\ s\ x\ (sol\ r))))) \Leftrightarrow ((t\ x = sol\ 0) \star (\Pi r : [0, d].\ ((sol)'\ r = f\ (\mathsf{set}\ t\ x\ (sol\ r))))) \Leftrightarrow (sol, t, d \vDash x' = f)$. □

Lemma 5.5 (Game coincidence)**.** *If* $s \overset{V}{=} t$ *for* $V \supseteq \mathrm{FV}(\alpha) \cup (\mathrm{FV}(\phi) \setminus \mathrm{MBV}(\alpha))$ *then:*

- *if* $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash (\langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\phi\urcorner\ s)$ *is inhabited then* $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash (\langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\phi\urcorner\ t)$ *is.*
- *if* $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash ([[\alpha]]\ \ulcorner\phi\urcorner\ s)$ *is inhabited then* $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash ([[\alpha]]\ \ulcorner\phi\urcorner\ t)$ *is.*

Because the proof is constructive, it amounts to an algorithm which computes an inhabitant of the conclusion.

Proof. Proven by induction simultaneously with Lemma 5.4. In the α^* case, we rely on a generalization of the inductive claim for formulas: the goal region of the game semantics

need not be the semantics $\ulcorner\phi\urcorner$ of a formula ϕ but is allowed to be any CIC region $P : (\mathfrak{S} \Rightarrow \mathbb{T})$ which satisfies the (semantic) coincidence property.

In each case, assume (A0) $s \overset{V}{=} t$ for $V \supseteq \mathrm{FV}(\alpha) \cup (\mathrm{FV}(\phi) \setminus \mathrm{MBV}(\alpha))$ and (A1) $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash M : (\langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\phi\urcorner\ s)$ or $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash M : ([[\alpha]]\ \ulcorner\phi\urcorner\ s)$ as appropriate. We note a simplification: In each case the proof of the conclusion begins by assuming (A2) $\ulcorner\Gamma\urcorner(\mathfrak{s})$ is inhabited, which by modus ponens on (A1) gives (A3) $\langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\phi\urcorner\ s$ or $[[\alpha]]\ \ulcorner\phi\urcorner\ s$ in each case. In short, in every case we are entitled to assume (A3).

Angel cases:

Case $x := f$: Since $\mathrm{FV}(x := f) = \mathrm{FV}(f)$, note by Assumption 1 (term coincidence) that (0) $f\ s = f\ t$. Note that $\mathrm{MBV}(x := f) = \{x\}$ so (1) $s \overset{V}{=} t$ for $V \supseteq \mathrm{FV}(\phi) \setminus \{x\}$. Then $\langle\!\langle x := f\rangle\!\rangle\ \ulcorner\phi\urcorner\ s \Leftrightarrow \ulcorner\phi\urcorner(\mathsf{set}\ s\ x\ (f\ s)) \Leftrightarrow_{(0)} \ulcorner\phi\urcorner(\mathsf{set}\ s\ x\ (f\ t)) \Leftrightarrow_{(1)} \ulcorner\phi\urcorner(\mathsf{set}\ t\ x\ (f\ t)) \Leftrightarrow \langle\!\langle x := f\rangle\!\rangle\ \ulcorner\phi\urcorner\ t$.

Case $x := *$: Note that $\mathrm{MBV}(x := *) = \{x\}$ so (0) $s \overset{V}{=} t$ for $V \supseteq \mathrm{FV}(\phi) \setminus \{x\}$. Then $\langle\!\langle x := *\rangle\!\rangle\ \ulcorner\phi\urcorner\ s \Leftrightarrow (\Sigma v : \mathbb{R}.\ \ulcorner\phi\urcorner\ (\mathsf{set}\ s\ x\ v)) \Leftrightarrow_{(0)} (\Sigma v : \mathbb{R}.\ \ulcorner\phi\urcorner\ (\mathsf{set}\ t\ x\ v)) \Leftrightarrow \langle\!\langle x := *\rangle\!\rangle\ \ulcorner\phi\urcorner\ t$.

Case $?\psi$: Have $\langle\!\langle ?\psi\rangle\!\rangle\ \ulcorner\phi\urcorner\ s \Leftrightarrow \ulcorner\psi\urcorner\ s \star \ulcorner\phi\urcorner\ s \underset{\text{IH}}{\Leftrightarrow} \ulcorner\psi\urcorner\ t \star \ulcorner\phi\urcorner\ t \Leftrightarrow \langle\!\langle ?\psi\rangle\!\rangle\ \ulcorner\phi\urcorner\ t$.

Case $x' = f\,\&\,\psi$: By the case for $(sol, s, d \vDash x' = f)$ have (0) $(sol, s, d \vDash x' = f) = (sol, t, d \vDash x' = f)$. Note $s \overset{\mathrm{FV}(\psi)\setminus\{x\}}{=} t$ since $\mathrm{FV}(\psi) \subseteq \mathrm{FV}(x' = f\,\&\,\psi)$ and $\{x, x'\} = \mathrm{MBV}(x' = f\,\&\,\psi)$[9] and $x' \notin \mathrm{FV}(\psi)$ by syntactic constraints, so the IH on ψ gives (1) $(\Pi r : [0, d].\ \ulcorner\psi\urcorner\ (\mathsf{set}\ s\ x\ (sol\ r))) = (\Pi r : [0, d].\ \ulcorner\psi\urcorner\ (\mathsf{set}\ t\ x\ (sol\ r)))$. Likewise $s \overset{\mathrm{FV}(\phi)\setminus\{x,x'\}}{=} t$ so the IH on ϕ gives (2):

$$\ulcorner\phi\urcorner\ (\mathsf{set}\ s\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d)))) \Leftrightarrow \ulcorner\phi\urcorner\ (\mathsf{set}\ t\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ t\ x\ (sol\ d))))$$

Then applying (1), (2), and (3) we have

$$\begin{aligned}
&\langle\!\langle x' = f\,\&\,\psi\rangle\!\rangle\ \ulcorner\phi\urcorner\ s \\
\Leftrightarrow &\Sigma d : \mathbb{R}_{\geq 0}.\ \Sigma sol : [0, d] \Rightarrow \mathbb{R}. \\
&\quad (sol, s, d \vDash x' = f) \\
&\quad \star (\Pi r : [0, d].\ \ulcorner\psi\urcorner\ (\mathsf{set}\ s\ x\ (sol\ r))) \\
&\quad \star (\ulcorner\phi\urcorner\ (\mathsf{set}\ s\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d))))) \\
\underset{\text{IH}}{\Leftrightarrow} &\Sigma d : \mathbb{R}_{\geq 0}.\ \Sigma sol : [0, d] \Rightarrow \mathbb{R}. \\
&\quad (sol, t, d \vDash x' = f) \\
&\quad \star (\Pi r : [0, d].\ \ulcorner\psi\urcorner\ (\mathsf{set}\ t\ x\ (sol\ r))) \\
&\quad \star (\ulcorner\phi\urcorner\ (\mathsf{set}\ t\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d))))) \\
= &\langle\!\langle x' = f\,\&\,\psi\rangle\!\rangle\ \ulcorner\phi\urcorner\ t
\end{aligned}$$

as desired.

Case $\alpha; \beta$: Note that $\mathrm{FV}(\alpha; \beta) \cup (\mathrm{FV}(\phi) \setminus \mathrm{MBV}(\alpha; \beta)) = \mathrm{FV}(\alpha) \cup (\mathrm{FV}(\beta) \setminus \mathrm{MBV}(\alpha)) \cup (\mathrm{FV}(\phi) \setminus \mathrm{MBV}(\alpha; \beta)) = \mathrm{FV}(\alpha) \cup ((\mathrm{FV}(\beta) \cup (\mathrm{FV}(\phi) \setminus \mathrm{MBV}(\beta))) \setminus \mathrm{MBV}(\alpha)) = \mathrm{FV}(\alpha) \cup (\mathrm{FV}(\langle\beta\rangle\phi) \setminus$

[9] Note that if we were to use a semantic characterization of must-bound variables instead, only the weaker condition $\{x, x'\} \supseteq \mathrm{MBV}(x' = f\,\&\,\psi)$ would hold.

$\mathrm{MBV}(\alpha)$) as required for the IH. Then $\langle\!\langle \alpha;\beta \rangle\!\rangle \ulcorner\phi\urcorner\, s \Leftrightarrow \langle\!\langle \alpha \rangle\!\rangle (\langle\!\langle \beta \rangle\!\rangle \ulcorner\phi\urcorner)\, s \Leftrightarrow \langle\!\langle \alpha \rangle\!\rangle \ulcorner\langle\beta\rangle\phi\urcorner\, s \underset{\text{IH}}{\Leftrightarrow}$ $\langle\!\langle \alpha \rangle\!\rangle \ulcorner\langle\beta\rangle\phi\urcorner\, t \Leftrightarrow \langle\!\langle \alpha \rangle\!\rangle (\langle\!\langle \beta \rangle\!\rangle \ulcorner\phi\urcorner)\, t \Leftrightarrow \langle\!\langle \alpha;\beta \rangle\!\rangle \ulcorner\phi\urcorner\, t$.

Case $\alpha \cup \beta$: Have $\langle\!\langle \alpha \cup \beta \rangle\!\rangle \ulcorner\phi\urcorner\, s \Leftrightarrow \langle\!\langle \alpha \rangle\!\rangle \ulcorner\phi\urcorner\, s + \langle\!\langle \beta \rangle\!\rangle \ulcorner\phi\urcorner\, s \underset{\text{IH}}{\Leftrightarrow} \langle\!\langle \alpha \rangle\!\rangle \ulcorner\phi\urcorner\, t + \langle\!\langle \beta \rangle\!\rangle \ulcorner\phi\urcorner\, t \Leftrightarrow$ $\langle\!\langle \alpha \cup \beta \rangle\!\rangle \ulcorner\phi\urcorner\, t$.

Case α^*: Let $FP = (\mu\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\, \lambda r : \mathfrak{S}\, (\ulcorner\phi\urcorner\, r \Rightarrow \tau'\, r) * (\langle\!\langle \alpha \rangle\!\rangle\, \tau'\, r \Rightarrow \tau'\, r))$, i.e., the fixed-point type family that defines the semantics of α^*. Define $FPV = \mathrm{FV}(\alpha) \cup \mathrm{FV}(\phi)$. Since $\mathrm{MBV}(\alpha^*) = \emptyset$ and $\mathrm{FV}(\alpha^*) = \mathrm{FV}(\alpha)$ by definition, note $FPV = \mathrm{FV}(\alpha) \cup (\mathrm{FV}(\phi) \setminus \mathrm{MBV}(\alpha^*) \subseteq V$.

The basic structure of this proof case is $\langle\!\langle \alpha^* \rangle\!\rangle \ulcorner\phi\urcorner\, s \Leftrightarrow FP\, s \Leftrightarrow FP\, t \Leftrightarrow \langle\!\langle \alpha^* \rangle\!\rangle \ulcorner\phi\urcorner\, t$ of which the key step to prove is $FP\, s = FP\, t$ as the other steps are immediate by the definition of FP.

We show $FP\, s \Leftrightarrow FP\, t$ for all $s \overset{FPV}{=} t$, that is, family FP has the coincidence property with semantic free variable set $FPV \subseteq V$. By symmetry of $s \overset{FPV}{=} t$, it suffices to show $FP\, s \Rightarrow FP\, t$ for all s, t such that $s \overset{FPV}{=} t$. We show $FP\, s \Rightarrow FP\, t$ by inner induction on the (fact that s belongs to) fixed-point construction.

In the base case, the case assumption is $\ulcorner\phi\urcorner\, s$ from which the desired conclusion $\ulcorner\phi\urcorner\, t$ follows immediately by the formula IH on ϕ and because $s \overset{FPV}{=} t$ for $FPV \supseteq \mathrm{FV}(\phi)$ (by construction of FPV).

In the inner inductive case, assume some family $\tau' : (\mathfrak{S} \Rightarrow \mathbb{T})$ and assume as the inner inductive hypothesis that τ' satisfies the coincidence property for FPV. Assume $\langle\!\langle \alpha \rangle\!\rangle\, \tau'\, s$ to prove $\langle\!\langle \alpha \rangle\!\rangle\, \tau'\, t$. In a minor handwave, the case holds by invoking a generalization of the outer game IH. The claim for games considers the case where the goal region in $\langle\!\langle \alpha \rangle\!\rangle$ is the semantics of some formula, rather we appeal to the generalization where any CIC region τ can be used as the goal region in $\langle\!\langle \alpha \rangle\!\rangle$ so long as it satisfies a strong enough coincidence property. Specifically, the stated claim requires $V \supseteq \mathrm{FV}(\alpha) \cup (\mathrm{FV}(\phi) \setminus \mathrm{MBV}(\alpha))$ where $\ulcorner\phi\urcorner$ is the goal region, instead we allow $V \supseteq \mathrm{FV}(\alpha) \cup (\hat{V} \setminus \mathrm{MBV}(\alpha))$ for any $\hat{V}$ that satisfies the coincidence property for goal region τ.

To apply the IH, we choose postcondition τ' and $\hat{V} = FPV$, then $s \overset{V}{=} t$ was already assumed for $V \supseteq FPV \supseteq \mathrm{FV}(\alpha) \cup (\hat{V} \setminus \mathrm{MBV}(\alpha))$ and the inner IH is that τ' satisfies the coincidence property for FPV, thus the preconditions of the outer IH hold and its application yields $\langle\!\langle \alpha \rangle\!\rangle\, \tau'\, t$ which completes the inner case and the inner induction.

The case is complete by observing $\langle\!\langle \alpha^* \rangle\!\rangle \ulcorner\phi\urcorner\, s \Leftrightarrow FP\, s \Leftrightarrow FP\, t \Leftrightarrow \langle\!\langle \alpha^* \rangle\!\rangle \ulcorner\phi\urcorner\, t$.

Case α^d: Have $\langle\!\langle \alpha^d \rangle\!\rangle \ulcorner\phi\urcorner\, s \Leftrightarrow [[\alpha]] \ulcorner\phi\urcorner\, s \Leftrightarrow [[\alpha]] \ulcorner\phi\urcorner\, t \Leftrightarrow \langle\!\langle \alpha^d \rangle\!\rangle \ulcorner\phi\urcorner\, t$.

Demon cases:

Case $x := f$: Since $\mathrm{FV}(x := f) = \mathrm{FV}(f)$, note by Assumption 1 (term coincidence) that (0) $f\ s = f\ t$. Note that $\mathrm{MBV}(x := f) = \{x\}$ so (1) $s \overset{V}{=} t$ for $V \supseteq \mathrm{FV}(\phi) \setminus \{x\}$. Then $[[x := f]] \ulcorner\phi\urcorner\, s \Leftrightarrow \ulcorner\phi\urcorner(\mathsf{set}\ s\ x\ (f\ s)) \Leftrightarrow_{(0)} \ulcorner\phi\urcorner(\mathsf{set}\ s\ x\ (f\ t)) \Leftrightarrow_{(1)} \ulcorner\phi\urcorner(\mathsf{set}\ t\ x\ (f\ t)) \Leftrightarrow$ $[[x := f]] \ulcorner\phi\urcorner\, t$.

Case $x := *$: Note that $\mathrm{MBV}(x := *) = \{x\}$ so (0) $s \overset{V}{=} t$ for $V \supseteq \mathrm{FV}(\phi) \setminus \{x\}$. Then $[[x := *]] \ulcorner\phi\urcorner\, s \Leftrightarrow (\Pi v : \mathbb{R}.\, \ulcorner\phi\urcorner\, (\mathsf{set}\ s\ x\ v)) \Leftrightarrow_{(0)} (\Pi v : \mathbb{R}.\, \ulcorner\phi\urcorner\, (\mathsf{set}\ t\ x\ v)) \Leftrightarrow [[x := *]] \ulcorner\phi\urcorner\, t$.

Case $?\phi$: Have $[[?\psi]] \ulcorner\phi\urcorner\, s \Leftrightarrow (\ulcorner\psi\urcorner\, s \Rightarrow \ulcorner\phi\urcorner\, s) \underset{\text{IH}}{\Leftrightarrow} (\ulcorner\psi\urcorner\, t \Rightarrow \ulcorner\phi\urcorner\, t) \Leftrightarrow [[?\psi]] \ulcorner\phi\urcorner\, t$.

Case $x' = f \,\&\, \psi$: By the case for $(sol, s, d \vDash x' = f)$ have (0) $(sol, s, d \vDash x' = f) = (sol, t, d \vDash x' = f)$. Note $s \overset{\mathrm{FV}(\psi)\setminus\{x\}}{=} t$ since $\mathrm{FV}(\psi) \subseteq \mathrm{FV}(x' = f \,\&\, \psi)$ and $\{x, x'\} = \mathrm{MBV}(x' = f \,\&\, \psi)$ and $x' \notin \mathrm{FV}(\psi)$ by syntactic constraints, thus the IH on ψ gives (1)

$$(\Pi r : [0, d].\ulcorner\psi\urcorner\ (\mathsf{set}\ s\ x\ (sol\ r))) = (\Pi r : [0, d].\ulcorner\psi\urcorner\ (\mathsf{set}\ t\ x\ (sol\ r)))$$

Likewise $s \overset{\mathrm{FV}(\phi)\setminus\{x,x'\}}{=} t$ so the IH on ϕ gives (2):

$$\ulcorner\phi\urcorner\ (\mathsf{set}\ s\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d)))) \Leftrightarrow \ulcorner\phi\urcorner\ (\mathsf{set}\ t\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ t\ x\ (sol\ d))))$$

Then applying (1), (2), and (3) we have

$$\begin{aligned}
&[[x' = f \,\&\, \psi]]\ \ulcorner\phi\urcorner\ s\\
\Leftrightarrow &\Pi d : \mathbb{R}_{\geq 0}.\ \Sigma sol : [0, d] \Rightarrow \mathbb{R}.\\
&\quad (sol, s, d \vDash x' = f)\\
&\quad \Rightarrow (\Pi r : [0, d].\ulcorner\psi\urcorner\ (\mathsf{set}\ s\ x\ (sol\ r)))\\
&\quad \Rightarrow \ulcorner\phi\urcorner\ (\mathsf{set}\ s\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d))))\\
\Leftrightarrow &\Pi d : \mathbb{R}_{\geq 0}.\ \Sigma sol : [0, d] \Rightarrow \mathbb{R}.\\
&\quad (sol, t, d \vDash x' = f)\\
&\quad \Rightarrow (\Pi sol : [0, d].\ulcorner\psi\urcorner\ (\mathsf{set}\ t\ x\ (sol\ r)))\\
&\quad \Rightarrow (\ulcorner\phi\urcorner\ (\mathsf{set}\ t\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d)))))\\
=&\ulcorner\phi\urcorner\ (\mathsf{set}\ t\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ t\ x\ (sol\ d))))
\end{aligned}$$

Case $\alpha; \beta$: Note that $\mathrm{FV}(\alpha; \beta) \cup (\mathrm{FV}(\phi) \setminus \mathrm{MBV}(\alpha; \beta)) = \mathrm{FV}(\alpha) \cup (\mathrm{FV}(\beta) \setminus \mathrm{MBV}(\alpha)) \cup (\mathrm{FV}(\phi) \setminus \mathrm{MBV}(\alpha; \beta)) = \mathrm{FV}(\alpha) \cup ((\mathrm{FV}(\beta) \cup (\mathrm{FV}(\phi) \setminus \mathrm{MBV}(\beta))) \setminus \mathrm{MBV}(\alpha)) = \mathrm{FV}(\alpha) \cup (\mathrm{FV}([\beta]\phi) \setminus \mathrm{MBV}(\alpha))$ as required for the IH. Then $[[\alpha; \beta]]\ \ulcorner\phi\urcorner\ s \Leftrightarrow [[\alpha]]\ ([[\beta]]\ \ulcorner\phi\urcorner)\ s \Leftrightarrow [[\alpha]]\ \ulcorner[\beta]\phi\urcorner\ s \underset{\text{IH}}{\Leftrightarrow} [[\alpha]]\ \ulcorner[\beta]\phi\urcorner\ t \Leftrightarrow [[\alpha]]\ ([[\beta]]\ \ulcorner\phi\urcorner)\ t \Leftrightarrow [[\alpha; \beta]]\ \ulcorner\phi\urcorner\ t$.

Case $\alpha \cup \beta$: Have $[[\alpha \cup \beta]]\ \ulcorner\phi\urcorner\ s \Leftrightarrow [[\alpha]]\ \ulcorner\phi\urcorner\ s * [[\beta]]\ \ulcorner\phi\urcorner\ s \Leftrightarrow [[\alpha]]\ \ulcorner\phi\urcorner\ t * [[\beta]]\ \ulcorner\phi\urcorner\ t \Leftrightarrow [[\alpha \cup \beta]]\ \ulcorner\phi\urcorner\ t$.

Case α^*:

Let $FP = (\rho\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}\ (\tau'\ t \Rightarrow [[\alpha]]\ \tau'\ t) * (\tau'\ t \Rightarrow \ulcorner\phi\urcorner\ t))$, i.e., the fixed-point semantics of α^*. Define $FPV = \mathrm{FV}(\alpha) \cup \mathrm{FV}(\phi)$. Since $\mathrm{MBV}(\alpha^*) = \emptyset$ and $\mathrm{FV}(\alpha^*) = \mathrm{FV}(\alpha)$ by definition, note $FPV = \mathrm{FV}(\alpha) \cup (\mathrm{FV}(\phi) \setminus \mathrm{MBV}(\alpha^*) \subseteq V$.

The basic structure of this proof case is $[[\alpha^*]]\ \ulcorner\phi\urcorner\ s \Leftrightarrow FP\ s \Leftrightarrow FP\ t \Leftrightarrow [[\alpha^*]]\ \ulcorner\phi\urcorner\ t$ of which the key step to prove is $FP\ s = FP\ t$ as the other steps are immediate by the definition of FP.

We show $FP\ s \Leftrightarrow FP\ t$ for all $s \overset{FPV}{=} t$, that is, family FP has the coincidence property with semantic free variable set $FPV \subseteq V$. We show $FP\ s \Leftrightarrow FP\ t$ by inner coinduction on the (fact that s belongs to) fixed-point construction. By symmetry of $s \overset{FPV}{=} t$, it suffices to show $FP\ s \Rightarrow FP\ t$ for all s, t such that $s \overset{FPV}{=} t$. We apply the symmetry argument *inside* the coinductive step, i.e., we assume both directions in the inner co-IH.

There is a single case in the coinductive proof. Assume some family $\tau' : (\mathfrak{S} \Rightarrow \mathbb{T})$ from the coinductive construction, i.e., such that for all states r have (τDef) $\tau'\ r \Rightarrow ((\ulcorner\phi\urcorner\ r)\star([[\alpha]]\ \tau'\ r))$. Assume as the inner coinductive hypothesis that (IIH) τ' satisfies the coincidence property for FPV, then want to show the coincidence property for the expansion, i.e., show

- (C1) $\ulcorner\phi\urcorner\ s \Rightarrow \ulcorner\phi\urcorner\ t$
- (C2) $[[\alpha]]\ \tau'\ s \Rightarrow [[\alpha]]\ \tau'\ t$

Claim (C1) is immediate by the outer IH on formula ϕ since $s \overset{V}{=} t$ for $V \supseteq FPV \supseteq \mathrm{FV}(\phi)$. Claim (C2) follows by generalizing the outer game IH in a minor handwave to allow any goal region which satisfies the coincidence property. Goal region τ' satisfies the coincidence property by the inner co-IH. We now discuss the use of outer game IH in further detail.

Specifically, the stated claim for games requires $V \supseteq \mathrm{FV}(\alpha) \cup (\mathrm{FV}(\phi) \setminus \mathrm{MBV}(\alpha))$ where $\ulcorner\phi\urcorner$ is the goal region, instead we allow $V \supseteq \mathrm{FV}(\alpha) \cup (\hat{V} \setminus \mathrm{MBV}(\alpha))$ for any $\hat{V}$ that satisfies the coincidence property for goal region τ'.

To apply the outer IH, we choose goal region τ' and $\hat{V} = FPV$, then $s \overset{V}{=} t$ (symmetrically, $t \overset{V}{=} s$) was already assumed for $V \supseteq FPV \supseteq \mathrm{FV}(\alpha) \cup (\hat{V} \setminus \mathrm{MBV}(\alpha))$ and the inner co-IH is that τ' satisfies the coincidence property for FPV, thus the preconditions of the outer IH hold and its application yields $[[\alpha]]\ \tau'\ s$ which proves the inner case and completes the inner coinduction.

The case is complete by observing $[[\alpha^*]]\ \ulcorner\phi\urcorner\ s \Leftrightarrow FP\ s \Leftrightarrow FP\ t \Leftrightarrow [[\alpha^*]]\ \ulcorner\phi\urcorner\ t$.

Case α^d: Have $[[\alpha^d]]\ \ulcorner\phi\urcorner\ s \Leftrightarrow \langle\langle\alpha\rangle\rangle\ \ulcorner\phi\urcorner\ s \Leftrightarrow \langle\langle\alpha\rangle\rangle\ \ulcorner\phi\urcorner\ t \Leftrightarrow [[\alpha^d]]\ \ulcorner\phi\urcorner\ t$.

□

Lemma 5.6 (Bound effect)**.** *Let $P : \mathfrak{S} \Rightarrow \mathbb{T}$ and $V \subseteq \mathrm{BV}(\alpha)^\complement$, a subset of the complement of bound variables of α. For simplicity, we state the case where truth of all formulas in Γ is evaluated at designated state $\mathfrak{s}$. To apply bound effect at a context which is evaluated at some other state, additionally apply the context coincidence claim from Lemma 5.4.*

- *There exists M such that $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash M : \langle\langle\alpha\rangle\rangle\ P\ s$ iff there exists N such that $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash N : \langle\langle\alpha\rangle\rangle\ (\lambda t.\ P\ t\star(s \overset{V}{=} t))\ s$.*
- *There exists M such that $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash M : [[\alpha]]\ P\ s$ iff there exists N such that $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash N : [[\alpha]]\ (\lambda t.\ P\ t\star(s \overset{V}{=} t))\ s$.*

Because the proof of each claim is constructive, each proof amounts to an algorithm for computing a proof term N satisfying the respective claim.

Proof. By the same argument as in the coincidence lemma, sequent-style bound effect follows trivially from formula-style bound effect. We first give a uniform argument for the converse direction, then prove the forward direction. The forward direction performs an outer induction on the size of V, then an inner simultaneous induction on Angelic and Demonic games. In some cases, we will write conjunction $P\ t\star(s \overset{V}{=} t)$ from the lemma statement in its equivalent commuted form $(s \overset{V}{=} t)\star P\ t$ for readability

Converse direction: By left projection, $P\ t\star(s \overset{V}{=} t) \Rightarrow P\ t$ for all t. Then by Lemma 5.1 have that $\langle\langle\alpha\rangle\rangle\ (\lambda t.\ P\ t\star(s \overset{V}{=} t))\ s$ implies $\langle\langle\alpha\rangle\rangle\ P\ s$ and likewise for $[[\alpha]]$.

Forward direction: By induction on $|V|$, allowing goal region P to vary. The case $|V| = 0$ is trivial since $\langle\!\langle\alpha\rangle\!\rangle\ (\lambda t.\ P\ t\star(s \overset{V}{=} t))\ s \Leftrightarrow \langle\!\langle\alpha\rangle\!\rangle\ (\lambda t.\ P\ t\star\ulcorner true\urcorner\ t)\ s \Leftrightarrow \langle\!\langle\alpha\rangle\!\rangle\ (\lambda t.\ P\ t\star\top)\ s \Leftrightarrow \langle\!\langle\alpha\rangle\!\rangle\ P\ s$. In the case $|V \cup \{x\}| = k + 1$ then apply the IH to $(\lambda t.\ P\ t\star(s\ x = t\ x))$ and set V to get that $\langle\!\langle\alpha\rangle\!\rangle\ (\lambda t.\ P\ t\star(s\ x = t\ x))\ s$ iff $\langle\!\langle\alpha\rangle\!\rangle\ (\lambda t.\ P\ t\star(s\ x = t\ x)\star(s \overset{V}{=} t))\ s$. Since we also have that $(\lambda t.\ P\ t\star(s\ x = t\ x)\star(s \overset{V}{=} t)) \Leftrightarrow (\lambda t.\ P\ t\star(s \overset{V\cup\{x\}}{=} t))$, then by transitivity it suffices to show that $\langle\!\langle\alpha\rangle\!\rangle\ (\lambda t.\ P\ t\star(s\ x = t\ x))\ s$ follows from $\langle\!\langle\alpha\rangle\!\rangle\ P\ s$. We proceed by inner induction on games, simultaneously for Angel and Demon. In each case we assume (A) $\langle\!\langle\alpha\rangle\!\rangle\ P\ s$ and show $\langle\!\langle\alpha\rangle\!\rangle\ (\lambda t.\ P\ t\star(s\ x = t\ x))\ s$ or likewise for $[[\alpha]]$. We do so by inner induction on games, simultaneously for Angel and Demon.

Angel cases:
Case $y := f$: Since $x \notin \mathrm{BV}(y := f)$ then (0) $x \neq y$.

$$\begin{aligned} & \langle\!\langle y := f\rangle\!\rangle\ (\lambda t.\ P\ t\star(s\ x = t\ x))\ s \\ \Leftrightarrow\quad & (P\ (\mathsf{set}\ s\ x\ (f\ s))\star((\mathsf{set}\ s\ y\ (f\ s))\ x = s\ x)) \\ \Leftrightarrow_{(0)}\ & (P\ (\mathsf{set}\ s\ x\ (f\ s))\star(s\ x = s\ x)) \end{aligned}$$

where the left holds by (A) and the right holds reflexively.

Case $y := *$: From (A) have some v such that (A1) $P\ (\mathsf{set}\ s\ y\ v)$. Since $x \notin \mathrm{BV}(y := *)$ then (0) $x \neq y$.

$$\begin{aligned} & \langle\!\langle y := *\rangle\!\rangle\ (\lambda t.\ P\ t\star(s\ x = t\ x))\ s \\ \Leftrightarrow\ & (\Sigma v : \mathbb{R}.\ P\ (\mathsf{set}\ s\ y\ v)\star((\mathsf{set}\ s\ y\ v)\ x = s\ x)) \\ \Leftrightarrow\ & (\Sigma v : \mathbb{R}.\ P\ (\mathsf{set}\ s\ y\ v)\star(s\ x = s\ x)) \end{aligned}$$

where the left holds from (A1) and the right holds for *any* v since $(s\ x = s\ x)$ reflexively.

Case $?\psi$: Have

$$\langle\!\langle ?\psi\rangle\!\rangle\ (\lambda t.\ P\ t\star(s\ x = t\ x))\ s \quad\Leftrightarrow\quad \ulcorner\psi\urcorner\ s\star P\ s\star(s\ x = s\ x) \quad\Leftrightarrow_A\quad (s\ x = s\ x)$$

which holds reflexively.

Case $y' = f\,\&\,\psi$: Note (0) $x \notin \{y, y'\} = \mathrm{BV}(y' = f\,\&\,\psi)$. From (A) unpack d and sol such that (1) $(sol, s, d \vDash y' = f)$ and (2) $(\Pi t : [0, d].\ P\ (\mathsf{set}\ s\ y\ (sol\ t)))$ and (3) $\ulcorner\psi\urcorner\ (\mathsf{set}\ s\ y\ (sol\ d))$. Then by (0) have (4) $(\mathsf{set}\ s\ y\ (sol\ d))\ x = s\ x$. Then using the same d and sol then using (1) (2) (3) and (4) have $\langle\!\langle y' = f\,\&\,\psi\rangle\!\rangle\ (\lambda t.\ P\ t\star(s\ x = t\ x))\ s$ as desired.

Case $\alpha;\beta$: Note (1) that the truth value of $(\lambda z.\ s\ x = t\ x)$ is constant with respect to z: the step that applies (1) uses it to rewrite $s\ x = t\ x$ in the postcondition of β. We write (A) when applying the IH on α and (B) when applying the IH on β. That is, (A), written commuted for sake of readability, gives $\langle\!\langle\alpha\rangle\!\rangle\ (\lambda t.\ (s\ x = t\ x)\star P\ t)\ s$ for all s and (B) gives

$\langle\!\langle\beta\rangle\!\rangle\ (\lambda z.\ P\ z\star(t\ x = z\ x))\ t$ for all t.

$$\begin{aligned}
&\langle\!\langle\alpha;\beta\rangle\!\rangle\ P\ s\\
\Leftrightarrow\ &\langle\!\langle\alpha\rangle\!\rangle\ (\langle\!\langle\beta\rangle\!\rangle\ P)\ s\\
\Leftrightarrow_{(A)}\ &\langle\!\langle\alpha\rangle\!\rangle\ (\lambda t.\ (s\ x = t\ x)\star\langle\!\langle\beta\rangle\!\rangle\ P\ t)\ s\\
\Leftrightarrow_{(B)}\ &\langle\!\langle\alpha\rangle\!\rangle\ (\lambda t.\ (s\ x = t\ x)\star\langle\!\langle\beta\rangle\!\rangle\ (\lambda z.\ P\ z\star(t\ x = z\ x))\ t)\ s\\
\Leftrightarrow_{(1)}\ &\langle\!\langle\alpha\rangle\!\rangle\ (\lambda t.\ (s\ x = t\ x)\star\langle\!\langle\beta\rangle\!\rangle\ (\lambda z.\ P\ z\star(s\ x = z\ x))\ t)\ s\\
\Rightarrow\ &\langle\!\langle\alpha\rangle\!\rangle\ (\langle\!\langle\beta\rangle\!\rangle(\lambda z.\ P\ z\star(s\ x = z\ x))\ s\\
\Rightarrow\ &\langle\!\langle\alpha;\beta\rangle\!\rangle\ (\lambda z.\ P\ z\star(s\ x = z\ x))\ s
\end{aligned}$$

which proves the case.

Case $\alpha \cup \beta$: Have either $\langle\!\langle\alpha\rangle\!\rangle\ P\ s$ or $\langle\!\langle\beta\rangle\!\rangle\ P\ s$. In each case, the IH applies by $\mathrm{BV}(\alpha \cup \beta)^{\complement} = \mathrm{BV}(\alpha)^{\complement} \cup \mathrm{BV}(\beta)^{\complement}$. In the first case, $\langle\!\langle\alpha\rangle\!\rangle\ (\lambda t.\ P\ t\star(s\ x = t\ x))\ s \Rightarrow \langle\!\langle\alpha \cup \beta\rangle\!\rangle\ (\lambda t.\ P\ t\star(s\ x = t\ x))\ s$. In the second case,

$$\langle\!\langle\beta\rangle\!\rangle\ (\lambda t.\ P\ t\star(s\ x = t\ x))\ s \Rightarrow \langle\!\langle\alpha \cup \beta\rangle\!\rangle\ (\lambda t.\ P\ t\star(s\ x = t\ x))\ s$$

Case α^*: Proceed by induction on membership of s in the fixed point in $\langle\!\langle\alpha^*\rangle\!\rangle\ P\ s$.

In the base case, $\langle\!\langle\alpha^*\rangle\!\rangle\ (\lambda t.\ P\ t\star(s\ x = t\ x))\ s$ trivially since $s\ x = s\ x$. In the inductive case wish to show $P\ t\star(s\ x = t\ x) \Rightarrow \langle\!\langle\alpha\rangle\!\rangle\ (\lambda z.\ P\ z\star(s\ x = t\ x))\ t$ which follows from the inner IH on membership and outer IH on α by transitivity and monotonicity over α. The IH on α applies because $\mathrm{BV}(\alpha^*) = \mathrm{BV}(\alpha)$.

Case α^d: From (A) have $[[\alpha]]\ P\ s$ so by IH have $[[\alpha]]\ (\lambda t.\ P\ t\star(s\ x = t\ x))\ s$ which gives $\langle\!\langle\phi^d\rangle\!\rangle\ (\lambda t.\ P\ t\star(s\ x = t\ x))\ s$.

Demon cases:

Case $y := f$: Since $x \notin \mathrm{BV}(y := f)$ then $(0)\ x \neq y$. Then

$$\begin{aligned}
&[[y := f]]\ (\lambda t.\ P\ t\star(s\ x = t\ x))\ s\\
\Leftrightarrow\ &P\ (\mathsf{set}\ s\ y\ (f\ s))\star((\mathsf{set}\ s\ y\ (f\ s))\ x = s\ x)\\
\Leftrightarrow_A\ &((\mathsf{set}\ s\ y\ (f\ s))\ x = s\ x)\\
\Leftrightarrow_{(0)}\ &(s\ x = s\ x)
\end{aligned}$$

which holds reflexively.

Case $y := *$: Since $x \notin \mathrm{BV}(y := *)$ then $(0)\ x \neq y$. Then

$$\begin{aligned}
&[[y := *]]\ (\lambda t.\ P\ t\star(s\ x = t\ x))\ s\\
\Leftrightarrow\ &(\Pi v : \mathbb{R}.\ (P\ (\mathsf{set}\ s\ y\ v)\star(\mathsf{set}\ s\ y\ v)\ x = s\ x))\\
\Leftrightarrow_A\ &(\Pi v : \mathbb{R}.\ ((\mathsf{set}\ s\ y\ v)\ x = s\ x))\\
\Leftrightarrow\ &(\Pi v : \mathbb{R}.\ (s\ x = s\ x))
\end{aligned}$$

which holds for all v since $(s\ x = s\ x)$ reflexively.

Case $?\psi$: Have $[[?\psi]]\ (\lambda t.\ P\ t\star(s\ x = t\ x))\ s \Leftrightarrow (\ulcorner\psi\urcorner\ s \Rightarrow P\ s\star(s\ x = s\ x)\ s) \Leftrightarrow_A (\ulcorner\psi\urcorner\ s \Rightarrow (s\ x = s\ x)) \Leftrightarrow (\ulcorner\psi\urcorner\ s \Rightarrow \top)$ since $(s\ x = s\ x)$ holds reflexively. Any proposition trivially implies $\top$, which completes the case.

Case $y' = f \,\&\, \psi$: Note (0) $x \notin \{y, y'\} = \mathrm{BV}(y' = f \,\&\, \psi)$. Assume arbitrary d and sol such that (1)$(sol, s, d \vDash y' = f)$ and (2) $(\Pi t : [0, d].\, P\ (\mathsf{set}\ s\ y\ (sol\ t)))$ and (3) $\ulcorner\psi\urcorner\ (\mathsf{set}\ s\ y\ (sol\ d))$. Then by (0) have (4) $(\mathsf{set}\ s\ y\ (sol\ d))\ x = s\ x$. Then by using (1) (2) (3) and (3) we have that $[[y' = f \,\&\, \psi]]\ (\lambda t.\ P\ t\star(s\ x = t\ x))\ s$ as desired.

Case $\alpha; \beta$: Note (0) $[[\beta]]\ (\lambda z.\ P\ z\star(t\ x = z\ x))\ t$ for all t by IH on β. Note (1) that the truth value of $(\lambda z.\ s\ x = t\ x)$ is constant as a function of z.

$$
\begin{aligned}
& [[\alpha;\beta]]\ (\lambda z.\ P\ z\star(s\ x = z\ x))\ s \\
\Leftrightarrow & [[\alpha]]\ ([[\beta]]\ (\lambda z.\ P\ z\star(s\ x = z\ x)))\ s \\
\Leftrightarrow & [[\alpha]]\ (\lambda t.\ (s\ x = t\ x)\star([[\beta]]\ (\lambda z.\ P\ z\star(s\ x = z\ x))\ t))\ s \\
\Leftrightarrow & [[\alpha]]\ (\lambda t.\ (s\ x = t\ x)\star([[\beta]]\ (\lambda z.\ P\ z\star(s\ x = z\ x))\ t))\ s \\
\Leftarrow_{(0)} & [[\alpha]]\ (\lambda t.\ (s\ x = t\ x)\star([[\beta]]\ (\lambda z.\ P\ z\star(s\ x = t\ x))\ t))\ s \\
\Leftarrow_{(1)} & [[\alpha]]\ (\lambda t.\ (s\ x = t\ x)\star([[\beta]]\ P\ t))\ s \\
\Leftarrow_{(1)} & [[\alpha]]\ ([[\beta]]\ P)\ s \\
\Leftarrow_{(1)} & [[\alpha;\beta]]\ P\ s
\end{aligned}
$$

which is (A).

Case $\alpha \cup \beta$: Have both $[[\alpha]]\ P\ s$ and $[[\beta]]\ P\ s$. Each IH applies by $\mathrm{BV}(\alpha \cup \beta)^{\complement} = \mathrm{BV}(\alpha)^{\complement} \cup \mathrm{BV}(\beta)^{\complement}$. The first IH gives $[[\alpha]]\ (\lambda z.\ P\ z\star(s\ x = z\ x))$ and the second IH gives $[[\beta]](\lambda z.\ P\ z\star(s\ x = z\ x))$, then have $[[\alpha \cup \beta]]\ (\lambda z.\ P\ z\star(s\ x = z\ x))\ s$ by the conjunction introduction rule of type theory.

Case α^*: By inversion on the proof (A) of $[[\alpha^*]]\ P\ s$, we have some J such that (1) $J\ s$, (2) $J\ t \Rightarrow [[\alpha]]\ J\ t$ for all t, and (3) $J\ t \Rightarrow P\ t$ for all t. We show that invariant region $(\lambda t.\ J\ t\star(s\ x = t\ x))$ is sufficient for an invariant proof of postcondition $s\ x = t\ x$.

Fact (1a) $J\ s\star(s\ x = s\ x)\ s$ follows from (1) and reflexivity. Fact (2a) $J\ t\star(s\ x = t\ x) \Rightarrow [[\alpha]]\ (\lambda z.\ J\ z\star(s\ x = z\ x))\ t$ because (2) gives a proof of $[[\alpha]]\ J\ t$ from which the IH on α gives $[[\alpha]]\ (\lambda z.\ J\ z\star(s\ x = z\ x))\ t$ as desired. Fact (3a) $J\ t\star(s\ x = t\ x) \Rightarrow (P\ t\star(s\ x = t\ x))\ t$ from (3) and hypothesis rule.

Then the coinductive term generated by (1a), (2a), and (3a) is a proof term for the type $[[\alpha^*]]\ (\lambda t.\ P\ t\star(s\ x = t\ x))\ s$ as desired.

Case α^d: From (A) have $\langle\!\langle\alpha\rangle\!\rangle\ P\ s$ so by IH have $\langle\!\langle\alpha\rangle\!\rangle\ (\lambda t.\ P\ t\star(s\ x = t\ x))\ s$ which gives $[[\phi^d]]\ (\lambda t.\ P\ t\star(s\ x = t\ x))\ s$. □

Lemma B.8 (Transposition)**.** *The equality* $e\frac{y}{x}\frac{y}{x} = e$ *holds for any* CdGL *or CIC term, formula, game, or type* e.

Proof. Trivial induction because we define $e\frac{y}{x}$ to be *transposition renaming* which renames x to y but also renames y to x. □

Note that because the proofs of uniform renaming are constructive, they implicitly define an algorithm for computing the renaming $M\frac{y}{x}$ of a proof term M.

Lemma 5.2 (Formula uniform renaming)**.** *If* $\ulcorner\Gamma\urcorner(s) \vdash M : \ulcorner\phi\urcorner\ s$ *then there exists some proof term (call it* $M\frac{y}{x}$*) such that* $\ulcorner\Gamma\urcorner(s\frac{y}{x}) \vdash M\frac{y}{x} : \ulcorner\phi\frac{y}{x}\urcorner\ s\frac{y}{x}$*. Also,* $(sol, s, d \vDash z' = f) \Leftrightarrow (sol, s\frac{y}{x}, d \vDash (z\frac{y}{x})' = f\frac{y}{x})$*. The function sol requires no renaming as it is just a univariate*

function from time to the value of a single scalar variable, as opposed to a function which accepts or returns a state.

Lemma 5.3 (Game uniform renaming)**.** *If* $\ulcorner\Gamma\urcorner(s) \vdash M : \langle\!\langle\alpha\rangle\!\rangle \ulcorner\phi\urcorner\ s$ *then there exists* $M\frac{y}{x}$ *such that* $\ulcorner\Gamma\urcorner(s\frac{y}{x}) \vdash M\frac{y}{x} : \langle\!\langle\alpha\frac{y}{x}\rangle\!\rangle \ulcorner\phi\frac{y}{x}\urcorner\ s\frac{y}{x}$*. If* $\ulcorner\Gamma\urcorner(s) \vdash M : [[\alpha]] \ulcorner\phi\urcorner\ s$ *then there exists* $M\frac{y}{x}$ *such that* $\ulcorner\Gamma\urcorner(s\frac{y}{x}) \vdash M\frac{y}{x} : [[\alpha\frac{y}{x}]] \ulcorner\phi\frac{y}{x}\urcorner\ s\frac{y}{x}$*.*

Proof. The cases for formulas, contexts, solutions, and games are all proven by simultaneous induction. In the case for sequential composition $\alpha;\beta$, we relax the IH to allow semantic renaming in an arbitrary goal region, not only syntactic renaming in a goal formula.

Context cases:

Case $\cdot$ for the empty context: Have $\cdot\frac{y}{x} = \cdot$ and $\ulcorner\cdot\urcorner$ holds trivially.

Case (Γ,ψ): Assume $\ulcorner(\Gamma,\psi)\urcorner(s) \Leftrightarrow \ulcorner\Gamma\urcorner(s) * \ulcorner\psi\urcorner\ s$, so that by IH on Γ have (0) $(\pi_0 M)\frac{y}{x} : \ulcorner\Gamma\frac{y}{x}\urcorner(s\frac{y}{x})$ and by the IH on ψ have (1) $(\pi_1 M)\frac{y}{x} : \ulcorner\psi\frac{y}{x}\urcorner\ (s\frac{y}{x})$. Then by conjunction of (0) and (1) have $M\frac{y}{x} : \ulcorner(\Gamma,\psi)\frac{y}{x}\urcorner(s\frac{y}{x})$ as desired.

The formula and game cases employ the following simplification using the context case. Each case first assumes (A1) a sequent of shape $\ulcorner\Gamma\urcorner(s) \vdash \ulcorner\phi\urcorner\ s$ then exhibits a sequent of shape $\ulcorner\Gamma\frac{y}{x}\urcorner(s\frac{y}{x}) \vdash \ulcorner\phi\urcorner\ (s\frac{y}{x})$, the first step of which is to assume (A2) $\ulcorner\Gamma\frac{y}{x}\urcorner(s\frac{y}{x})$. From (A2), uniform renaming on contexts yields (by Lemma B.8) $\ulcorner\Gamma\urcorner(s)$, then by modus ponens on (A1) have $\ulcorner\phi\urcorner\ s$. That is, the remaining cases are free to not explicitly discuss the context Γ because it is handled uniformly in each case.

Also, we write $z\frac{y}{x}$ as shorthand for a variable which is z in the case $z \notin \{x,y\}$, or y when $z = x$, or x when $z = y$.

Formula cases:

Case $\langle\alpha\rangle\phi$: Assume $\ulcorner\langle\alpha\rangle\phi\urcorner\ s \Leftrightarrow \langle\!\langle\alpha\rangle\!\rangle \ulcorner\phi\urcorner\ s \Leftrightarrow \langle\!\langle\alpha\frac{y}{x}\rangle\!\rangle \ulcorner\phi\frac{y}{x}\urcorner\ s\frac{y}{x} \Leftrightarrow \ulcorner\langle\alpha\rangle\phi\frac{y}{x}\urcorner\ s\frac{y}{x}$.

Case $[\alpha]\phi$: Have $\ulcorner[\alpha]\phi\urcorner\ s \Leftrightarrow [[\alpha]] \ulcorner\phi\urcorner\ s \Leftrightarrow [[\alpha\frac{y}{x}]] \ulcorner\phi\frac{y}{x}\urcorner\ s\frac{y}{x} \Leftrightarrow \ulcorner[\alpha]\phi\frac{y}{x}\urcorner\ s\frac{y}{x}$.

Case $f \sim g$: The case follows from Assumption 2 (term renaming): $\ulcorner f \sim g\urcorner\ s \Leftrightarrow f\ s \sim g\ s \Leftrightarrow (f\frac{y}{x})\ (s\frac{y}{x}) \sim (g\frac{y}{x})\ (s\frac{y}{x}) \Leftrightarrow \ulcorner(f \sim g)\frac{y}{x}\urcorner\ (s\frac{y}{x})$.

Angel cases:

Case $z := f$: Have: $\langle\!\langle z := f\rangle\!\rangle \ulcorner\phi\urcorner\ s \Leftrightarrow \ulcorner\phi\urcorner(\mathsf{set}\ s\ z\ (f\ s)) \Leftrightarrow \ulcorner\phi\frac{y}{x}\urcorner\ (\mathsf{set}\ s\ z\ (f\ s))\frac{y}{x} \Leftrightarrow \ulcorner\phi\frac{y}{x}\urcorner\ (\mathsf{set}\ (s\frac{y}{x})\ (z\frac{y}{x})\ (f\ s)) \Leftrightarrow \ulcorner\phi\frac{y}{x}\urcorner\ (\mathsf{set}\ (s\frac{y}{x})\ (z\frac{y}{x})\ (f\frac{y}{x}\ s\frac{y}{x})) \Leftrightarrow \langle\!\langle(z\frac{y}{x}) := (f\frac{y}{x})\rangle\!\rangle \ulcorner\phi\frac{y}{x}\urcorner\ s\frac{y}{x} \Leftrightarrow \langle\!\langle\{z := f\}\frac{y}{x}\rangle\!\rangle \ulcorner\phi\frac{y}{x}\urcorner\ s\frac{y}{x}$.

Case $z := *$: Have $\langle\!\langle z := *\rangle\!\rangle \ulcorner\phi\urcorner\ s \Leftrightarrow \Sigma v : \mathbb{R}.\ulcorner\phi\urcorner(\mathsf{set}\ s\ z\ v) \Leftrightarrow \Sigma v : \mathbb{R}.\ulcorner\phi\frac{y}{x}\urcorner\ (\mathsf{set}\ s\ z\ v)\frac{y}{x} \Leftrightarrow \Sigma v : \mathbb{R}.\ulcorner\phi\frac{y}{x}\urcorner\ (\mathsf{set}\ (s\frac{y}{x})\ (z\frac{y}{x})\ v) \Leftrightarrow \langle\!\langle(z\frac{y}{x}) := *\rangle\!\rangle \ulcorner\phi\frac{y}{x}\urcorner\ s\frac{y}{x} \Leftrightarrow \langle\!\langle\{z := *\}\frac{y}{x}\rangle\!\rangle \ulcorner\phi\frac{y}{x}\urcorner\ s\frac{y}{x}$.

Case $?\psi$: Have $\langle\!\langle?\psi\rangle\!\rangle \ulcorner\phi\urcorner\ s \Leftrightarrow \ulcorner\psi\urcorner\ s * \ulcorner\phi\urcorner\ s \Leftrightarrow \ulcorner\psi\frac{y}{x}\urcorner\ s\frac{y}{x} * \ulcorner\phi\frac{y}{x}\urcorner\ s\frac{y}{x} \Leftrightarrow \langle\!\langle?\psi\frac{y}{x}\rangle\!\rangle \ulcorner\phi\frac{y}{x}\urcorner\ s\frac{y}{x}$.

Case $sol, s, d \vDash z' = f$: Have

$$
\begin{aligned}
&(sol, s, d \vDash z' = f)\\
\Leftrightarrow&(s\ z = sol\ 0) * \Pi r : [0,d].\ ((sol)'\ r = f\ (\mathsf{set}\ s\ z\ (sol\ r)))\\
\Leftrightarrow&(s\frac{y}{x}\ z\frac{y}{x} = sol\ 0) * \Pi r : [0,d].\ ((sol)'\ r = f\frac{y}{x}\ ((\mathsf{set}\ s\ z\ (sol\ r))\frac{y}{x}))\\
\Leftrightarrow&(s\frac{y}{x}\ z\frac{y}{x} = sol\ 0) * \Pi r : [0,d].\ ((sol)'\ r = f\frac{y}{x}\ (\mathsf{set}\ s\ (z\frac{y}{x})\ (sol\ r)))\\
\Leftrightarrow&(sol, s\frac{y}{x}, d \vDash z\frac{y}{x}' = f\frac{y}{x})
\end{aligned}
$$

Case $z' = f \,\&\, \psi$: Have

$$\begin{aligned}
&\langle\!\langle z' = f \,\&\, \psi \rangle\!\rangle \ulcorner \phi \urcorner \ s \\
\Leftrightarrow &\Sigma d : \mathbb{R}_{\geq 0}. \Sigma sol : [0,d] \Rightarrow \mathbb{R}. (sol, s, d \vDash z' = f) \\
&\quad \star (\Pi t : [0,d]. \ulcorner \psi \urcorner \ (\mathsf{set}\ s\ z\ (sol\ t))) \\
&\quad \star \ulcorner \phi \urcorner \ (\mathsf{set}\ s\ (x,x')\ (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d)))) \\
\Leftrightarrow &\Sigma d : \mathbb{R}_{\geq 0}. \Sigma sol : [0,d] \Rightarrow \mathbb{R}. (sol, s\tfrac{y}{x}, d \vDash z\tfrac{y}{x}' = f\tfrac{y}{x}) \\
&\quad \star (\Pi t : [0,d]. \ulcorner \psi\tfrac{y}{x} \urcorner \ (\mathsf{set}\ s\ z\ (sol\ t))\tfrac{y}{x}) \\
&\quad \star \ulcorner \phi\tfrac{y}{x} \urcorner \ ((\mathsf{set}\ s\ (x,x')\ (sol\ d, f\ (\mathsf{set}\ s\ x\ (sol\ d))))\tfrac{y}{x}) \\
\Leftrightarrow &\Sigma d : \mathbb{R}_{\geq 0}. \Sigma sol : [0,d] \Rightarrow \mathbb{R}. (sol, s\tfrac{y}{x}, d \vDash z\tfrac{y}{x}' = f\tfrac{y}{x}) \\
&\quad \star (\Pi t : [0,d]. \ulcorner \psi\tfrac{y}{x} \urcorner \ (\mathsf{set}\ s\tfrac{y}{x}\ z\tfrac{y}{x}\ (sol\ t))) \\
&\quad \star \ulcorner \phi\tfrac{y}{x} \urcorner \ (\mathsf{set}\ s\tfrac{y}{x}\ (z,z')\ (sol\ d, f\ (\mathsf{set}\ s\ z\ (sol\ d)))) \\
\Leftrightarrow &\langle\!\langle \{z' = f \,\&\, \psi\}\tfrac{y}{x} \rangle\!\rangle \ulcorner \phi\tfrac{y}{x} \urcorner \ s\tfrac{y}{x}
\end{aligned}$$

Case $\alpha;\beta$: In this case, $\langle\!\langle \alpha;\beta \rangle\!\rangle \ulcorner \phi \urcorner \ s \Leftrightarrow \langle\!\langle \alpha \rangle\!\rangle \ (\langle\!\langle \beta \rangle\!\rangle \ulcorner \phi \urcorner) \ s \Leftrightarrow \langle\!\langle \alpha \rangle\!\rangle \ (\lambda t.\ \langle\!\langle \beta \rangle\!\rangle \ulcorner \phi \urcorner \ t) \ s \Leftrightarrow \langle\!\langle \alpha\tfrac{y}{x} \rangle\!\rangle \ (\lambda t.\ \langle\!\langle \beta\tfrac{y}{x} \rangle\!\rangle \ulcorner \phi\tfrac{y}{x} \urcorner \ t) \ s\tfrac{y}{x} \Leftrightarrow \langle\!\langle \alpha\tfrac{y}{x} \rangle\!\rangle \ ((\langle\!\langle \beta \rangle\!\rangle \ulcorner \phi \urcorner)\tfrac{y}{x}) \ s\tfrac{y}{x} \Leftrightarrow \langle\!\langle \{\alpha;\beta\}\tfrac{y}{x} \rangle\!\rangle \ulcorner \phi\tfrac{y}{x} \urcorner \ s\tfrac{y}{x}$.

Case $\alpha \cup \beta$: Have $\langle\!\langle \alpha \cup \beta \rangle\!\rangle \ulcorner \phi \urcorner \ s \Leftrightarrow \langle\!\langle \alpha \rangle\!\rangle \ulcorner \phi \urcorner \ s + \langle\!\langle \beta \rangle\!\rangle \ulcorner \phi \urcorner \ s \Leftrightarrow \langle\!\langle \alpha\tfrac{y}{x} \rangle\!\rangle \ulcorner \phi\tfrac{y}{x} \urcorner \ s\tfrac{y}{x} + \langle\!\langle \beta\tfrac{y}{x} \rangle\!\rangle \ulcorner \phi\tfrac{y}{x} \urcorner \ s\tfrac{y}{x} \Leftrightarrow \langle\!\langle \{\alpha \cup \beta\}\tfrac{y}{x} \rangle\!\rangle \ulcorner \phi\tfrac{y}{x} \urcorner \ s\tfrac{y}{x}$.

Case α^*: Note (0)

$$\begin{aligned}
&(\mu\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}\, (\phi\ t \Rightarrow \tau'\ t) \star (\langle\!\langle \alpha \rangle\!\rangle \ \tau'\ t \Rightarrow \tau'\ t))\tfrac{y}{x} \\
\Leftrightarrow &\ (\mu\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}\, (\phi\tfrac{y}{x}\ t \Rightarrow \tau'\ t\tfrac{y}{x}) \star (\langle\!\langle \alpha\tfrac{y}{x} \rangle\!\rangle \ \tau'\ t \Rightarrow \tau'\ t))
\end{aligned}$$

by an inner induction. Likewise, an induction on the fixed-point membership of s gives

$$\begin{aligned}
&\langle\!\langle \alpha^* \rangle\!\rangle \ulcorner \phi \urcorner \ s \\
\Leftrightarrow &\ (\mu\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}). \lambda t : \mathfrak{S}\, (\phi\ t \Rightarrow \tau'\ t) \star (\langle\!\langle \alpha \rangle\!\rangle \ \tau'\ t \Rightarrow \tau'\ t))\ s \\
\Leftrightarrow &\ (\mu\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}). \lambda t : \mathfrak{S}\, (\phi\tfrac{y}{x}\ t\tfrac{y}{x} \Rightarrow \tau'\ t\tfrac{y}{x}) \star (\langle\!\langle \alpha\tfrac{y}{x} \rangle\!\rangle \ \tau'\ t \Rightarrow \tau'\ t))\tfrac{y}{x}\ s\tfrac{y}{x}
\end{aligned}$$

This simplifies to $\langle\!\langle \{\alpha^*\}\tfrac{y}{x} \rangle\!\rangle \ulcorner \phi\tfrac{y}{x} \urcorner \ s\tfrac{y}{x}$ as desired.

Case α^d: Have $\langle\!\langle \alpha^d \rangle\!\rangle \ulcorner \phi \urcorner \ s \Leftrightarrow [[\alpha]] \ulcorner \phi \urcorner \ s \Leftrightarrow [[\alpha\tfrac{y}{x}]] \ulcorner \phi\tfrac{y}{x} \urcorner \ s\tfrac{y}{x} \Leftrightarrow \langle\!\langle \{\alpha^d\}\tfrac{y}{x} \rangle\!\rangle \ulcorner \phi\tfrac{y}{x} \urcorner \ s\tfrac{y}{x}$.

Demon cases:

Case $z := f$: Have $[[z := f]] \ulcorner \phi \urcorner \ s \Leftrightarrow \ulcorner \phi \urcorner (\mathsf{set}\ s\ z\ (f\ s)) \Leftrightarrow \ulcorner \phi\tfrac{y}{x} \urcorner \ (\mathsf{set}\ s\ z\ (f\ s))\tfrac{y}{x} \Leftrightarrow \ulcorner \phi\tfrac{y}{x} \urcorner \ (\mathsf{set}\ (s\tfrac{y}{x})\ (z\tfrac{y}{x})\ (f\ s)) \Leftrightarrow \ulcorner \phi\tfrac{y}{x} \urcorner \ (\mathsf{set}\ (s\tfrac{y}{x})\ (z\tfrac{y}{x})\ (f\tfrac{y}{x}\ s\tfrac{y}{x})) \Leftrightarrow [[(z\tfrac{y}{x}) := (f\tfrac{y}{x})]] \ulcorner \phi\tfrac{y}{x} \urcorner \ s\tfrac{y}{x} \Leftrightarrow [[\{z := f\}\tfrac{y}{x}]] \ulcorner \phi\tfrac{y}{x} \urcorner \ s\tfrac{y}{x}$.

Case $z := *$: By a chain of equivalences, have $[[z := *]] \ulcorner \phi \urcorner \ s \Leftrightarrow \Pi v : \mathbb{R}. \ulcorner \phi \urcorner (\mathsf{set}\ s\ z\ v) \Leftrightarrow \Pi v : \mathbb{R}. \ulcorner \phi\tfrac{y}{x} \urcorner \ (\mathsf{set}\ s\ z\ v)\tfrac{y}{x} \Leftrightarrow \Pi v : \mathbb{R}. \ulcorner \phi\tfrac{y}{x} \urcorner \ (\mathsf{set}\ (s\tfrac{y}{x})\ (z\tfrac{y}{x})\ v) \Leftrightarrow [[z\tfrac{y}{x} := *]] \ulcorner \phi\tfrac{y}{x} \urcorner \ s\tfrac{y}{x} \Leftrightarrow [[\{z := *\}\tfrac{y}{x}]] \ulcorner \phi\tfrac{y}{x} \urcorner \ s\tfrac{y}{x}$.

Case $?\phi$: Have a chain of equivalences: $[[?\psi]] \ulcorner \phi \urcorner \ s \Leftrightarrow (\ulcorner \psi \urcorner \ s \Rightarrow \ulcorner \phi \urcorner \ s) \Leftrightarrow (\ulcorner \psi\tfrac{y}{x} \urcorner \ s\tfrac{y}{x} \Rightarrow \ulcorner \phi\tfrac{y}{x} \urcorner \ s\tfrac{y}{x}) \Leftrightarrow [[\{?\psi\}\tfrac{y}{x}]] \ulcorner \phi\tfrac{y}{x} \urcorner \ s\tfrac{y}{x}$.

Case $y' = g \,\&\, \psi$: From (A2) have (L1) $x \neq y$ and (L2) $y \notin \mathrm{FV}(g)$. Note that this is a stronger admissibility condition than those for $y := g$ and $y := *$. Unlike the former

constructs, y is always a free variable of $y' = g \,\&\, \psi$, thus if we attempted to define a sufficient admissibility condition to support the case $x = y$, we would find it unsatisfiable. Thus, we simply say x cannot be substituted into f in any ODE which binds x. So (L1) (L2), then have (S) $\{y := g\}^f_x = x := (g^f_x)$ also, from (L2) have (*) $(f\ s) = (f\ (\mathsf{set}\ s\ y\ (g^f_x\ s)))$ by Assumption 1 (term coincidence) since $s = (\mathsf{set}\ s\ y\ (g^f_x\ s))$ on $\{y\}^{\complement} \supseteq \mathrm{FV}(f)^{\complement}$. then

Have (0) $(sol, s^f_x, d \vDash y' = g) \Leftrightarrow (sol, s, d \vDash (y' = g)^f_x)$ by the "solves" IH.

Have (1) for all $t \in [0, d]$, $\ulcorner\psi\urcorner\ (\mathsf{set}\ (s^f_x)\ y\ (sol\ t)) \Leftrightarrow \ulcorner\psi^f_x\urcorner\ (\mathsf{set}\ s\ y\ (sol\ t))$ by IH on ψ and because $\mathsf{set}\ (s^f_x)\ y\ (sol\ t) \Leftrightarrow \mathsf{set}\ (\mathsf{set}\ s\ x\ (f\ s))\ y\ (sol\ t) \Leftrightarrow \mathsf{set}\ (\mathsf{set}\ s\ y\ (sol\ t))\ x\ (f\ s) \Leftrightarrow \mathsf{set}\ (\mathsf{set}\ s\ y\ (sol\ t))\ x\ (f\ (\mathsf{set}\ s\ y\ (sol\ t)))$ since by (L2) have $y \notin \mathrm{FV}(f)$ thus $s = (\mathsf{set}\ s\ y\ (sol\ t))$ on $\mathrm{FV}(f)^{\complement}$ and Assumption 1 (term coincidence) applies. Have (2)

$$\begin{aligned}&\ulcorner\phi\urcorner\ (\mathsf{set}\ (s^f_x)\ (y, y')\ (sol\ d, f\ (\mathsf{set}\ (s^f_x)\ y\ (sol\ d))))\\ \Leftrightarrow\ &\ulcorner\phi^f_x\urcorner\ (\mathsf{set}\ s\ (y, y')\ (sol\ d, f\ (\mathsf{set}\ (s^f_x)\ y\ (sol\ d))))\end{aligned}$$

by the IH on ϕ and because

$$\begin{aligned}&\mathsf{set}\ (s^f_x)\ (y, y')\ (sol\ d, g\ (\mathsf{set}\ (s^f_x)\ y\ (sol\ d)))\\ \Leftrightarrow\ &\mathsf{set}\ (\mathsf{set}\ s\ (y, y')\ (sol\ d, g\ (\mathsf{set}\ (s^f_x)\ y\ (sol\ d))))\ x\ (f\ (\mathsf{set}\ s\ y\ (sol\ d)))\end{aligned}$$

by the same argument as above and because $g\ (\mathsf{set}\ (s^f_x)\ y\ (sol\ d)) \Leftrightarrow (g^f_x)\ (\mathsf{set}\ s\ y\ (sol\ d))$ by term IH.

$$\begin{aligned}&[[y' = g \,\&\, \psi]]\ \ulcorner\phi\urcorner s^f_x\\ \Leftrightarrow &\Pi d : \mathbb{R}_{\geq 0}.\ \Sigma sol : [0, d] \Rightarrow \mathbb{R}.\\ &(sol, s^f_x, d \vDash y' = g)\\ \Rightarrow &(\Pi t : [0, d].\ \ulcorner\psi\urcorner\ (\mathsf{set}\ s^f_x\ y\ (sol\ t)))\\ \Rightarrow &\ulcorner\phi\urcorner\ (\mathsf{set}\ (s^f_x)\ (y, y')\ (sol\ d, g\ (\mathsf{set}\ (s^f_x)\ y\ (sol\ d))))\\ \Leftrightarrow_{(0,1,2)} &\Pi d : \mathbb{R}_{\geq 0}.\ \Sigma sol : [0, d] \Rightarrow \mathbb{R}.\\ &(sol, s, d \vDash (y' = g)^f_x)\\ \Rightarrow &(\Pi t : [0, d].\ \ulcorner\psi^f_x\urcorner\ (\mathsf{set}\ s\ y\ (sol\ t)))\\ \Rightarrow &\ulcorner\phi^f_x\urcorner\ (\mathsf{set}\ s\ (y, y')\ (sol\ d, (g^f_x)\ (\mathsf{set}\ s\ y\ (sol\ d))))\\ \Leftrightarrow &[[(y' = g \,\&\, \psi)^f_x]]\ \ulcorner\phi^f_x\urcorner\ s\end{aligned}$$

Case $\alpha; \beta$: We reason by a chain of equalities: $[[\alpha; \beta]]\ \ulcorner\phi\urcorner\ s \Leftrightarrow [[\alpha]]\ (\langle\!\langle\beta\rangle\!\rangle\ \ulcorner\phi\urcorner)\ s \Leftrightarrow [[\alpha]]\ (\lambda t.\ [[\beta]]\ \ulcorner\phi\urcorner\ t)\ s \Leftrightarrow [[\alpha\frac{y}{x}]]\ (\lambda t.\ [[\beta\frac{y}{x}]]\ \ulcorner\phi\frac{y}{x}\urcorner\ t)\ s\frac{y}{x} \Leftrightarrow [[\alpha\frac{y}{x}]]\ ((([[\beta]]\ \ulcorner\phi\urcorner)\frac{y}{x})\ s\frac{y}{x} \Leftrightarrow [[\{\alpha; \beta\}\frac{y}{x}]]\ \ulcorner\phi\frac{y}{x}\urcorner\ s\frac{y}{x}$.

Case $\alpha \cup \beta$: By a chain of equalities, have: $[[\alpha \cup \beta]]\ \ulcorner\phi\urcorner\ s \Leftrightarrow [[\alpha]]\ \ulcorner\phi\urcorner\ s \star [[\beta]]\ \ulcorner\phi\urcorner\ s \Leftrightarrow [[\alpha\frac{y}{x}]]\ \ulcorner\phi\frac{y}{x}\urcorner\ s\frac{y}{x} \star [[\beta\frac{y}{x}]]\ \ulcorner\phi\frac{y}{x}\urcorner\ s\frac{y}{x} \Leftrightarrow [[\{\alpha \cup \beta\}\frac{y}{x}]]\ \ulcorner\phi\frac{y}{x}\urcorner\ s\frac{y}{x}$.

Case α^*: Note (0) that the following equivalence can be proven by an inner coinduction:

$$\begin{aligned}&(\rho\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}\ (\tau'\ t \Rightarrow [[\alpha]]\ \tau'\ t) \star (\tau'\ t \Rightarrow \ulcorner\phi\urcorner\ t))\frac{y}{x}\\ \Leftrightarrow\ &(\rho\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}\ (\tau'\ t \Rightarrow [[\alpha\frac{y}{x}]]\ \tau'\ t) \star (\tau'\ t \Rightarrow \ulcorner\phi\frac{y}{x}\urcorner\ t))\end{aligned}$$

Likewise

$$\begin{aligned}&[[\alpha^*]]\ \ulcorner\phi\urcorner\ s\\ \Leftrightarrow\ &(\rho\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}\,(\tau'\ t \Rightarrow [[\alpha]]\ \tau'\ t) \star (\tau'\ t \Rightarrow \ulcorner\phi\urcorner\ t))\ s\\ \Leftrightarrow\ &(\rho\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}\,(\tau'\ t \Rightarrow [[\alpha\tfrac{y}{x}]]\ \tau'\ t) \star (\tau'\ t \Rightarrow \ulcorner\phi\tfrac{y}{x}\urcorner\ t))\ s\tfrac{y}{x}\end{aligned}$$

by coinduction on the membership of s in the fixed point. This simplifies to $[[\alpha^*\frac{y}{x}]]\ \ulcorner\phi\frac{y}{x}\urcorner\ s\frac{y}{x}$ as desired.

Case α^d: Have $[[\alpha^d]]\ \ulcorner\phi\urcorner\ s \Leftrightarrow \langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\phi\urcorner\ s \Leftrightarrow \langle\!\langle\alpha\frac{y}{x}\rangle\!\rangle\ \ulcorner\phi\frac{y}{x}\urcorner\ s\frac{y}{x} \Leftrightarrow [[\{\alpha^d\}\frac{y}{x}]]\ \ulcorner\phi\frac{y}{x}\urcorner\ s\frac{y}{x}$. □

Before proving formula and game substitution, we must introduce a notion of adjoint states, the same notion used in the substitution proof for CGL. We will use a general notation σ for substitutions, but in practice most substitutions will be of form $\cdot^f_x$ and replace a single variable x with a term f. Recall that $Dom(\sigma)$ is the set of variables substituted in σ.

Definition B.2 (Adjoints)**.** We write $\sigma^*_t(s)$ for the adjoint state of s where the substituted variables of substitution σ are replaced by their values at state t. To define adjoint states, let n be the number of substitution pairs in σ, writing x_i and $f_i = \sigma\, x_i$ respectively for the ith substituted variable and its replacement, then define:

$$\sigma^*_t(s) = \mathsf{set} \cdots (\mathsf{set}\ s\ x_1\ (f_1\ t))\ x_n\ (f_n\ t)$$

We abbreviate $\sigma^*(\omega)$ for the common case $\sigma^*_s(s)$ which specializes to $\mathsf{set}\ s\ x\ (f\ x)$ in the special case where $\sigma = \cdot^f_x$. For any region P, the notation $\sigma^*_s(P) = (\lambda t.\ P\ (\sigma^*_s(t)))$ denotes a region of states whose adjoints with respect to s belong to P. Function $\mathrm{FV}(\sigma)$ denotes the union of free variables of all term replacements (or right-hand sides) defined by σ, i.e.,

$$\mathrm{FV}(\sigma) \equiv \bigcup_{x \in \mathrm{FV}(\sigma\, x)}$$

We say that σ is U-admissible for expression e iff $\left(\bigcup_{x\in\mathrm{FV}(e)\cap Dom(\sigma)} \mathrm{FV}(\sigma\, x)\right) \cap U = \emptyset$, i.e., if no replacement performed in $\sigma(e)$ mentions any element of U as a free variable. Substitution σ is admissible for expression e if no variable replacement performed by $\sigma(e)$ introduces a free dependency on a variable which is bound in the surrounding context.

Having defined adjoints and admissibility, we give a lemma about adjoints.

Lemma B.9 (Adjoints)**.**

- *If* $s \overset{\mathrm{FV}(\sigma)}{=} t$*, then* $\sigma^*_s(u) = \sigma^*_t(u)$ *for all states* u*.*
- *If* $s \overset{U^{\complement}}{=} t$ *and* σ *is* U*-admissible for an expression* f, α*, or* ϕ*, then for all states* u *have:*
  - $f\ \sigma^*_s(u) = f\ \sigma^*_t(u)$
  - $\ulcorner\phi\urcorner\ \sigma^*_s(u)$ *iff* $\ulcorner\phi\urcorner\ \sigma^*_t(u)$
  - $\langle\!\langle\alpha\rangle\!\rangle\ (\sigma^*_s(P))\ \sigma^*_s(u)$ *iff* $\langle\!\langle\alpha\rangle\!\rangle(\sigma^*_t(P))\ \sigma^*_t(u)$
  - $[[\alpha]]\ (\sigma^*_s(P))\ \sigma^*_s(u)$ *iff* $[[\alpha]](\sigma^*_t(P))\ \sigma^*_t(u)$

Proof. The first claim holds by definition of adjoints and applying Assumption 1 (term coincidence) on each replacement in σ. The remaining claims hold by induction or coinduction on the expressions; they reduce to the case for program variables. If x is not

replaced in σ, the adjoints agree by definition. If it is, then $\mathrm{FV}(\sigma(x)) \subseteq U^{\complement}$ by definition of admissibility, then $s \overset{\mathrm{FV}(\sigma(x))}{=} t$ by assumption, so the semantics agree by Lemma 5.4 or by Lemma 5.5 for formulas and games respectively. □

Note that because the proofs of substitution are constructive, they implicitly define an algorithm for computing the proof term substitution result $\sigma(M)$, including the case M^f_x.

Lemma 5.7 (Formula substitution)**.** *If σ is admissible for Γ, then $\ulcorner\sigma(\Gamma)\urcorner(s) \Leftrightarrow \ulcorner\Gamma\urcorner(\sigma^*_s(s))$. If σ is additionally admissible for ϕ then $\ulcorner\Gamma\urcorner(\sigma^*_s(s)) \vdash M : \ulcorner\phi\urcorner\ \sigma^*_s(s)$ iff there exists a CIC proof term (call it $\sigma(M)$) such that $\ulcorner\sigma(\Gamma)\urcorner(s) \vdash \sigma(M) : \ulcorner\sigma(\phi)\urcorner\ s$. Likewise for predicate $(sol, s, d \vDash y' = g)$. In the converse direction, the witness is not necessarily M.*

Because the proof (in Appendix B.2) is constructive, it amounts to an algorithm for computing the substitution result $\sigma(M)$ for CIC proof terms M of the formula and context semantics. From here onward when M is a CIC proof term for the formula or context semantics, notation $\sigma(M)$ refers specifically to the proof term constructed in accordance to the proof.

Lemma 5.8 (Game substitution)**.** *If σ is admissible for $\langle\alpha\rangle\phi$ or respectively $[\alpha]\phi$, then:*

- *$\ulcorner\Gamma\urcorner(\sigma^*_s(s)) \vdash M : \langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\phi\urcorner\ \sigma^*_s(s)$ iff there exists a CIC proof term (call it $\sigma(M)$) such that $\ulcorner\sigma(\Gamma)\urcorner(s) \vdash \sigma(M) : \langle\!\langle\sigma(\alpha)\rangle\!\rangle\ \ulcorner\sigma(\phi)\urcorner\ s$.*
- *$\ulcorner\Gamma\urcorner(\sigma^*_s(s)) \vdash M : [[\alpha]]\ \ulcorner\phi\urcorner\ \sigma^*_s(s)$ iff there exists a CIC proof term (call it $\sigma(M)$) such that $\ulcorner\sigma(\Gamma)\urcorner(s) \vdash \sigma(M) : [[\sigma(\alpha)]]\ \ulcorner\sigma(\phi)\urcorner\ s$.*

In the converse direction, the witness is not necessarily M.

Because the proof (in Appendix B.2) is constructive, it amounts to an algorithm for computing the substitution result $\sigma(M)$ for CIC proof terms M of the semantics of games. From here onward when M is a CIC proof term for the semantics of games, notation $\sigma(M)$ refers specifically to the proof term constructed in accordance to the proof.

Proof. As usual, we do not explicitly construct the substituted proof term $\sigma(M)$, we just show that it exists by showing implications between (types-as-)propositions. Note that in the cases for games, several applications of the IH assume a stronger claim that the one stated. Specifically, we state the claim where the postcondition is a formula ϕ because formulas have an easily-understood syntactic notion of substitution. The theorem works equally well when the goal region is some region not of the form $\ulcorner\phi\urcorner$, in which case substitution is interpreted semantically. That is, the stated claim uses $\ulcorner\phi\urcorner$ in the assumption and $\ulcorner\sigma(\phi)\urcorner$ in the conclusion, but it is equally permissible to use P in the assumption and $(\lambda t.\ P\ \sigma^*_s(t))$ in the conclusion. The two presentations are made the same by applying the Lemma 5.7 IH on ϕ. We work primarily with the oversimplified theorem lemma statement solely to avoid excessive invocation of the IH.

In the formula cases, we assume (A0) $\ulcorner\Gamma\urcorner(\sigma^*_s(s)) \vdash M : \ulcorner\phi\urcorner\ \sigma^*_s(s)$, (A1) admissibility of $\sigma(\Gamma)$ and (A2) admissibility of $\sigma(\phi)$. Likewise for contexts, games, and predicate $(sol, s, d \vDash x' = f)$. We note that in each case, the admissibility conditions of the IH hold following (A1) and (A2) and unpacking the inductive definition of admissibility. As usual, we also do not explicitly discuss the contexts in the formula and game cases since the cases with contexts follow easily from those without, combined with IHs on the contexts.

Context cases:
Case $\Gamma = \cdot$ where $\cdot$ is the empty context: Then trivially $\ulcorner\cdot\urcorner(\sigma_s^*(s)) \Leftrightarrow \top \Leftrightarrow \ulcorner\sigma(\cdot)\urcorner(s)$.

Case (Γ, ψ): From (A0) have $\ulcorner\Gamma\urcorner(\sigma_s^*(s))$ and $\ulcorner\psi\urcorner(\sigma_s^*(s))$, then by the IHs have $\ulcorner\sigma(\Gamma)\urcorner(s)$ and $\ulcorner\sigma(\psi)\urcorner(\sigma_s^*(s))$ giving $\ulcorner\sigma(\Gamma, \psi)\urcorner(s)$ as desired.

Formula cases:
Case $g \sim h$: From (A0) by Assumption 3 (term substitution) have $\ulcorner g \sim h\urcorner\ \sigma_s^*(s) \Leftrightarrow (g\ \sigma_s^*(s) \sim h\ \sigma_s^*(s)) \Leftrightarrow (\sigma(g)\ s \sim \sigma(h)\ s) \Leftrightarrow (\sigma(g \sim h))\ s$.

Case $[\alpha]\phi$: From (A0) by the inductive hypothesis on game α we have $\ulcorner[\alpha]\phi\urcorner\ \sigma_s^*(s) \Leftrightarrow [[\alpha]]\ \ulcorner\phi\urcorner\ \sigma_s^*(s) \Leftrightarrow [[\sigma(\alpha)]]\ \ulcorner\sigma(\phi)\urcorner\ s \Leftrightarrow \ulcorner\sigma([\alpha]\phi)\urcorner\ s$.

Case $\langle\alpha\rangle\phi$: From (A0) have by the IH on α that $\ulcorner\langle\alpha\rangle\phi\urcorner\ \sigma_s^*(s) \Leftrightarrow \langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\phi\urcorner\ \sigma_s^*(s) \Leftrightarrow \langle\!\langle\sigma(\alpha)\rangle\!\rangle\ \ulcorner\sigma(\phi)\urcorner\ s \Leftrightarrow \ulcorner\sigma(\langle\alpha\rangle\phi)\urcorner\ s$.

Case $(sol, s, d \vDash y' = g)$: From (A2) have (L1) that $y \notin Dom(\sigma)$ and (L2) $y \notin \mathrm{FV}(\sigma)$. Have (0) $(\sigma_s^*(s))\ y = s\ y$ by (L1). Then for all $r \in [0, d]$ have (1) $g\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ r)) = (\sigma(g))\ (\mathsf{set}\ s\ y\ (sol\ r))$ by Assumption 1 (term coincidence). This step relies on the equation $(\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ r)) = \sigma^*_{\mathsf{set}\ s\ y\ (sol\ r)}(\mathsf{set}\ s\ y\ (sol\ r))$ which holds by (L2) $s \overset{\{y\}^{\complement}}{=} \mathsf{set}\ s\ y\ (sol\ r)$ for $\{y\}^{\complement} \supseteq \mathrm{FV}(f)^{\complement}$ thus Assumption 1 (term coincidence) applies. Have

$$
\begin{aligned}
&(sol, \sigma_s^*(s), d \vDash y' = g)\\
\Leftrightarrow\ &((\sigma_s^*(s))\ y = sol\ 0 \star \Pi r : [0, d].\ ((sol)'\ r = g\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ r))))\\
\Leftrightarrow_{(0,1)}\ &(s\ y = sol\ 0 \star \Pi r : [0, d].\ ((sol)'\ r = (\sigma(g))\ (\mathsf{set}\ s\ y\ (sol\ r))))\\
\Leftrightarrow\ &(sol, s, d \vDash (y' = \sigma(g))^f_x)
\end{aligned}
$$

Angel cases:
Case $y := g$: From (A2) have (L1) $y \notin Dom(\sigma)$ and (L2) σ is $\{y\}$-admissible for ϕ. Note (*) by (L2) and Lemma B.9 that $\ulcorner\phi\urcorner\ \sigma^*_{\mathsf{set}\ s\ y\ (g\ \sigma_s^*(s))}(\mathsf{set}\ s\ y\ (g\ \sigma_s^*(s))) \Leftrightarrow \ulcorner\phi\urcorner\ \sigma_s^*(\mathsf{set}\ s\ y\ (g\ \sigma_s^*(s)))$ since $s \overset{\{y\}^{\complement}}{=} \mathsf{set}\ s\ y\ (g\ \sigma_s^*(s))$. Step (1) follows from assumption (L1) that $y \notin Dom(\sigma)$. Then we have

$$
\begin{aligned}
&\langle\!\langle y := g\rangle\!\rangle\ \ulcorner\phi\urcorner\ \sigma_s^*(s)\\
\Leftrightarrow\ &\ulcorner\phi\urcorner\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (g\ (\sigma_s^*(s))))\\
\Leftrightarrow_{(1)}\ &\ulcorner\phi\urcorner\ (\sigma_s^*(\mathsf{set}\ s\ y\ (g\ \sigma_s^*(s))))\\
\Leftrightarrow_{*}\ &\ulcorner\phi\urcorner\ (\sigma^*_{\mathsf{set}\ s\ y\ (g\ \sigma_s^*(s))}(\mathsf{set}\ s\ y\ (g\ \sigma_s^*(s))))\\
\underset{\text{IH}}{\Leftrightarrow}\ &\ulcorner\sigma(\phi)\urcorner\ (\mathsf{set}\ s\ y\ (g\ \sigma_s^*(s)))\\
\Leftrightarrow\ &\ulcorner\sigma(\phi)\urcorner\ (\mathsf{set}\ s\ y\ (\sigma(g)\ s))\\
\Leftrightarrow\ &\langle\!\langle y := (\sigma(g))\rangle\!\rangle\ \ulcorner\sigma(\phi)\urcorner\ s\\
\Leftrightarrow\ &\langle\!\langle \sigma(\{y := g\})\rangle\!\rangle\ \ulcorner\sigma(\phi)\urcorner\ s
\end{aligned}
$$

Case $y := *$: From (A2) have (L1) $x \notin Dom(\sigma)$ and (L2) σ is $\{y\}$-admissible for ϕ. Note (*) by (L2) and Lemma B.9 that $\ulcorner\phi\urcorner\ \sigma^*_{\mathsf{set}\ s\ y\ v}(\mathsf{set}\ s\ y\ v) \Leftrightarrow \ulcorner\phi\urcorner\ \sigma_s^*(\mathsf{set}\ s\ y\ v)$ for all v

since $s \overset{\{y\}^{\complement}}{=} \mathsf{set}\ s\ y\ v$. Step (1) follows from assumption (L1) that $y \notin Dom(\sigma)$. Then

$$\begin{aligned}
&\langle\!\langle y := * \rangle\!\rangle\ \ulcorner\phi\urcorner\ \sigma_s^*(s)\\
\Leftrightarrow\ &\Sigma v : \mathbb{R}.\ulcorner\phi\urcorner\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ v)\\
\Leftrightarrow_{(1)}\ &\Sigma v : \mathbb{R}.\ulcorner\phi\urcorner\ (\sigma_s^*(\mathsf{set}\ s\ y\ v))\\
\Leftrightarrow_{*}\ &\Sigma v : \mathbb{R}.\ulcorner\phi\urcorner\ (\sigma^*_{\mathsf{set}\ s\ y\ v}(\mathsf{set}\ s\ y\ v))\\
\underset{\text{IH}}{\Leftrightarrow}\ &\Sigma v : \mathbb{R}.\ulcorner\sigma(\phi)\urcorner\ (\mathsf{set}\ s\ y\ v)\\
\Leftrightarrow\ &\langle\!\langle y := * \rangle\!\rangle\ \ulcorner\sigma(\phi)\urcorner\ s\\
\Leftrightarrow\ &\langle\!\langle \sigma(\{y := *\}) \rangle\!\rangle\ \ulcorner\sigma(\phi)\urcorner\ s
\end{aligned}$$

Case $?\psi$: In this case, $\langle\!\langle ?\psi \rangle\!\rangle\ \ulcorner\phi\urcorner\ \sigma_s^*(s) \Leftrightarrow \ulcorner\psi\urcorner\ \sigma_s^*(s) \star \ulcorner\phi\urcorner\sigma_s^*(s) \underset{\text{IH}}{\Leftrightarrow} \ulcorner\sigma(\psi)\urcorner\ s \star \ulcorner\sigma(\phi)\urcorner\ s \Leftrightarrow \langle\!\langle \sigma(\{?\psi\}) \rangle\!\rangle\ \ulcorner\sigma(\phi)\urcorner\ s$.

Case $y' = g\,\&\,\psi$: From (A2) have (L1) $y \notin Dom(\sigma)$ and (L2) σ is $\{y, y'\}$-admissible for ϕ and ψ. We will use Lemma B.9 (the adjoint lemma) several times. Since $\{y, y'\}^{\complement} \subset \{y\}^{\complement}$, note (Agree) $s \overset{\{y,y'\}^{\complement}}{=} (\mathsf{set}\ s\ y\ (sol\ t))$. In combination with (L2), which says σ is $\{y, y'\}$-admissible for ϕ, we can apply Lemma B.9 to get (*1)

$$\ulcorner\psi\urcorner\ (\sigma_s^*((\mathsf{set}\ s\ y\ (sol\ t)))) \Leftrightarrow \ulcorner\psi\urcorner\ (\sigma^*_{(\mathsf{set}\ s\ y\ (sol\ t))}((\mathsf{set}\ s\ y\ (sol\ t))))$$

Note (*2) by (L2) and Lemma B.9 that

$$\begin{aligned}
&\ulcorner\phi\urcorner\ (\sigma_s^*((\mathsf{set}\ s\ (y,y')\ (sol\ d, g\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ d))))))\\
&\ulcorner\phi\urcorner\ (\sigma^*_{\mathsf{set}\ s\ (y,y')\ (sol\ d, g\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ d)))}((\mathsf{set}\ s\ (y,y')\ (sol\ d, g\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ d))))))
\end{aligned}$$

since $s \overset{\{y,y'\}^{\complement}}{=} \mathsf{set}\ s\ (y,y')\ (sol\ d, g\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ d)))$. Note (*3) by (L2) and Lemma B.9 that $g\ (\sigma_s^*((\mathsf{set}\ s\ y\ (sol\ d)))) = g\ (\sigma^*_{\mathsf{set}\ s\ y\ (sol\ d)}((\mathsf{set}\ s\ y\ (sol\ d))))$ since $s \overset{\{y\}^{\complement}}{=} \mathsf{set}\ s\ y\ (sol\ d)$.

Have (0) $(sol, \sigma_s^*(s), d \vDash y' = g) = (sol, s, d \vDash \sigma((y' = g)))$ by the IH for ODE solutions.

Have (1) for all $t \in [0, d]$, $\ulcorner\psi\urcorner\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ t)) \Leftrightarrow_{(\text{L1})} \ulcorner\psi\urcorner\ (\sigma_s^*((\mathsf{set}\ s\ y\ (sol\ t)))) \Leftrightarrow_{(*1)} \ulcorner\psi\urcorner\ (\sigma^*_{(\mathsf{set}\ s\ y\ (sol\ t))}((\mathsf{set}\ s\ y\ (sol\ t)))) \underset{\text{IH}}{\Leftrightarrow} \ulcorner\sigma(\psi)\urcorner\ (\mathsf{set}\ s\ y\ (sol\ t))$.

Abbreviate $ss = (\mathsf{set}\ s\ (y,y')\ (sol\ d, g\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ d))))$. Have (2)

$$\begin{aligned}
&\ulcorner\phi\urcorner\ (\mathsf{set}\ (\sigma_s^*(s))\ (y,y')\ (sol\ d, g\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ d))))\\
\Leftrightarrow_{(\text{L1})}\ &\ulcorner\phi\urcorner\ (\sigma_s^*((\mathsf{set}\ s\ (y,y')\ (sol\ d, g\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ d))))))\\
\Leftrightarrow_{(*2)}\ &\ulcorner\phi\urcorner\ (\sigma^*_{\mathsf{set}\ s\ (y,y')\ (sol\ d, g\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ d)))}(ss))\\
\Leftrightarrow\ &\ulcorner\sigma(\phi)\urcorner\ (\mathsf{set}\ s\ (y,y')\ (sol\ d, g\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ d))))\\
\Leftrightarrow_{(\text{L1})}\ &\ulcorner\sigma(\phi)\urcorner\ (\mathsf{set}\ s\ (y,y')\ (sol\ d, g\ (\sigma_s^*((\mathsf{set}\ s\ y\ (sol\ d))))))\\
\Leftrightarrow_{(*3)}\ &\ulcorner\sigma(\phi)\urcorner\ (\mathsf{set}\ s\ (y,y')\ (sol\ d, g\ (\sigma^*_{\mathsf{set}\ s\ y\ (sol\ d)}((\mathsf{set}\ s\ y\ (sol\ d))))))\\
\underset{\text{IH}}{\Leftrightarrow}\ &\ulcorner\sigma(\phi)\urcorner\ (\mathsf{set}\ s\ (y,y')\ (sol\ d, \sigma(g)\ (\mathsf{set}\ s\ y\ (sol\ d))))
\end{aligned}$$

Now the main claim follows from (1), (2), and (3):

$$
\begin{aligned}
&\langle\!\langle y' = g \,\&\, \psi \rangle\!\rangle \ {}^{\ulcorner}\phi^{\urcorner}\, \sigma_s^*(s) \\
\Leftrightarrow\ & \Sigma d : \mathbb{R}_{\geq 0}.\, \Sigma sol : [0,d] \Rightarrow \mathbb{R}. \\
& \quad (sol, \sigma_s^*(s), d \vDash y' = g) \\
& \quad \star(\Pi t : [0,d].\, {}^{\ulcorner}\psi^{\urcorner}\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ t))) \\
& \quad \star\, {}^{\ulcorner}\phi^{\urcorner}\ (\mathsf{set}\ (\sigma_s^*(s))\ (y, y')\ (sol\ d, g\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ d)))) \\
\Leftrightarrow_{(0,1,2)}\ & \Sigma d : \mathbb{R}_{\geq 0}.\, \Sigma sol : [0,d] \Rightarrow \mathbb{R}. \\
& \quad (sol, s, d \vDash \sigma((y' = g))) \\
& \quad \star(\Pi t : [0,d].\, {}^{\ulcorner}\sigma(\psi)^{\urcorner}\ (\mathsf{set}\ s\ y\ (sol\ t))) \\
& \quad \star\, {}^{\ulcorner}\sigma(\phi)^{\urcorner}\ (\mathsf{set}\ s\ (y, y')\ (sol\ d, \sigma(g)\ (\mathsf{set}\ s\ y\ (sol\ d)))) \\
\Leftrightarrow\ & \langle\!\langle \sigma(\{y' = g \,\&\, \psi\}) \rangle\!\rangle \ {}^{\ulcorner}\sigma(\phi)^{\urcorner}\ s
\end{aligned}
$$

Case $\alpha;\beta$: Admissibility condition (A2) gives that (L1) $Dom(\sigma) \cap \mathrm{BV}(\alpha;\beta) = \emptyset$, (L2) σ is $\mathrm{BV}(\sigma(\alpha))$-admissible for β, and (L3) σ is $\mathrm{BV}(\sigma(\alpha;\beta))$-admissible for ϕ. Facts (1a) and (1b) say that in the relevant steps, the postconditions of α and β respectively are only applied to states of shape $\sigma_s^*(t)$ and $\sigma_t^*(u)$ for some t, u. Formally, the notation $(\lambda\sigma_s^*(t).\ P\ \sigma_s^*(t))$ is definable as $(\lambda u.P\ u \star \Sigma t : \mathfrak{S}.\, u = \sigma_s^*(t))$, i.e., region $(\lambda\sigma_s^*(t).\ P\ \sigma_s^*(t))$ is a total type-level function, but $(\lambda\sigma_s^*(t).\ P\ \sigma_s^*(t))\ u$ is only inhabited when u is provably of form $\sigma_s^*(t)$. The steps (1a) and (1b) hold by Lemma 5.6 and (L1): the final state agrees with initial state $\sigma_s^*(s)$ on the substituted variables of σ, which by definition of adjoint suffices to show the final state is of form $\sigma_s^*(t)$ for some state t.

Facts (IHA), (IHB), (IHP) are the IHs on α, β, and ϕ. Step (*1) says that $\sigma_s^*(t) = \sigma_t^*(t)$ because $s \overset{\mathrm{BV}(\sigma(\alpha))^{\complement}}{=} t$ by Lemma 5.6. Step (*2) says that $\sigma_t^*(u) = \sigma_u^*(u)$ because $t \overset{\mathrm{BV}(\sigma(\beta))^{\complement}}{=} u$ by Lemma 5.6.

Next, we have

$$
\begin{aligned}
&\langle\!\langle \alpha;\beta \rangle\!\rangle \ {}^{\ulcorner}\phi^{\urcorner}\ \sigma_s^*(s) \\
\Leftrightarrow\ & \langle\!\langle \alpha \rangle\!\rangle\ (\langle\!\langle \beta \rangle\!\rangle \ {}^{\ulcorner}\phi^{\urcorner})\ \sigma_s^*(s) \\
\Leftrightarrow_{(1a)}\ & \langle\!\langle \alpha \rangle\!\rangle\ (\lambda\sigma_s^*(t).\ \langle\!\langle \beta \rangle\!\rangle \ {}^{\ulcorner}\phi^{\urcorner}\ \sigma_s^*(t))\ \sigma_s^*(s) \\
\Leftrightarrow_{(1b)}\ & \langle\!\langle \alpha \rangle\!\rangle\ (\lambda\sigma_s^*(t).\ \langle\!\langle \beta \rangle\!\rangle\ (\lambda\sigma_t^*(u).\ {}^{\ulcorner}\phi^{\urcorner}\ \sigma_t^*(u))\ \sigma_s^*(t))\ \sigma_s^*(s) \\
\Leftrightarrow_{(\mathrm{IHA})}\ & \langle\!\langle \sigma(\alpha) \rangle\!\rangle\ (\lambda t.\ \langle\!\langle \beta \rangle\!\rangle\ (\lambda\sigma_t^*(u).\ {}^{\ulcorner}\phi^{\urcorner}\ \sigma_t^*(u))\ \sigma_s^*(t))\ s \\
\Leftrightarrow_{(*1)}\ & \langle\!\langle \sigma(\alpha) \rangle\!\rangle\ (\lambda t.\ \langle\!\langle \beta \rangle\!\rangle\ (\lambda\sigma_t^*(u).\ {}^{\ulcorner}\phi^{\urcorner}\ \sigma_t^*(u))\ \sigma_t^*(t))\ s \\
\Leftrightarrow_{(\mathrm{IHB})}\ & \langle\!\langle \sigma(\alpha) \rangle\!\rangle\ (\lambda t.\ \langle\!\langle \sigma(\beta) \rangle\!\rangle\ (\lambda u.\ {}^{\ulcorner}\phi^{\urcorner}\ \sigma_t^*(u))\ t)\ s \\
\Leftrightarrow_{(*2)}\ & \langle\!\langle \sigma(\alpha) \rangle\!\rangle\ (\lambda t.\ \langle\!\langle \sigma(\beta) \rangle\!\rangle\ (\lambda u.\ {}^{\ulcorner}\phi^{\urcorner}\ \sigma_u^*(u))\ t)\ s \\
\Leftrightarrow_{(\mathrm{IHP})}\ & \langle\!\langle \sigma(\alpha) \rangle\!\rangle\ (\lambda t.\ \langle\!\langle \sigma(\beta) \rangle\!\rangle\ (\lambda u.\ {}^{\ulcorner}\sigma(\phi)^{\urcorner}\ u)\ t)\ s \\
\Leftrightarrow\ & \langle\!\langle \sigma(\alpha) \rangle\!\rangle\ (\langle\!\langle \sigma(\beta) \rangle\!\rangle \ {}^{\ulcorner}\sigma(\phi)^{\urcorner})\ s \\
\Leftrightarrow\ & \langle\!\langle \sigma(\{\alpha;\beta\}) \rangle\!\rangle\ \sigma(\phi)\ s
\end{aligned}
$$

Case $\alpha \cup \beta$: In this case, have: $\langle\!\langle \alpha \cup \beta \rangle\!\rangle \ulcorner \phi \urcorner\ \sigma_s^*(s) \Leftrightarrow \langle\!\langle \alpha \rangle\!\rangle \ulcorner \phi \urcorner\ \sigma_s^*(s) + \langle\!\langle \beta \rangle\!\rangle \ulcorner \phi \urcorner\ \sigma_s^*(s) \Leftrightarrow \langle\!\langle \sigma(\alpha) \rangle\!\rangle \ulcorner \sigma(\phi) \urcorner\ s + \langle\!\langle \sigma(\beta) \rangle\!\rangle \ulcorner \sigma(\phi) \urcorner\ s \Leftrightarrow \langle\!\langle \sigma(\alpha \cup \beta) \rangle\!\rangle \ulcorner \sigma(\phi) \urcorner\ s$.

Case α^*:

Consider fixed points: $FP_1 \equiv (\mu\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}\ (\ulcorner \psi \urcorner\ t \Rightarrow \tau'\ t) \star (\langle\!\langle \alpha \rangle\!\rangle\ \tau'\ t \Rightarrow \tau'\ t))$ and $FP_2 \equiv (\mu\tau'' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}\ (\ulcorner \sigma(\psi) \urcorner\ t \Rightarrow \tau''\ t) \star (\langle\!\langle \sigma(\alpha) \rangle\!\rangle\ \tau''\ t \Rightarrow \tau''\ t))$.

The key fact for this case (*) is $FP_1\ \sigma_s^*(s)$ iff $FP_2\ s$ (for all $s : \mathfrak{S}$). We prove fact (*) inductively. Technically, each direction is an induction on the respective fixed-point definition, but we present a single proof because both inductions rely on primarily the same bidirectional reasoning. Let τ' and τ'' refer to the type family variables from the fixed-point constructions, for which we are allowed to assume an inductive hypothesis.

In the base case, we show that the base cases of each fixed point agree. We rely on the IH on ψ: $\ulcorner \psi \urcorner\ \sigma_s^*(s)) \Leftrightarrow \ulcorner \sigma(\psi) \urcorner\ s \Rightarrow \tau''\ s$.

In the inductive case, (1) refers to the fact $\sigma_s^*(t) = \sigma_t^*(t)$ which holds by Lemma B.9 because we assumed that σ was $\mathrm{BV}(\sigma(\alpha))$-admissible in α and because by Lemma 5.6 we have $s \overset{\mathrm{BV}(\sigma(\alpha))^{\complement}}{=} t$. Fact (OIH) refers to the outer IH on α, which we generalize here to allow an arbitrary goal region, not just a CdGL formula. Fact (IIH) refers to the inner IH, which allows us to rewrite $\tau'\ \sigma_s^*(s)$ iff $\tau''\ s$, but only after executing α in order to ensure well-foundedness. Fact (adj) refers to the fact that in the relevant proof step τ' is only ever invoked on states of shape $\sigma_s^*(t)$, which is the case by Lemma 5.6 and the admissibility assumption $\mathrm{BV}(\alpha) \cap Dom(\sigma) = \emptyset$.

We prove the inductive case: $(\langle\!\langle \alpha \rangle\!\rangle\ \tau'\ \sigma_s^*(s)) \Leftrightarrow_{(\mathrm{adj})} (\langle\!\langle \alpha \rangle\!\rangle\ (\lambda\sigma_s^*(t).\ \tau'\ \sigma_s^*(t))\ \sigma_s^*(s)) \Leftrightarrow_{(\mathrm{OIH})} (\langle\!\langle \sigma(\alpha) \rangle\!\rangle\ (\lambda t.\ \tau'\ \sigma_s^*(t))\ s) \Leftrightarrow_{(1)} (\langle\!\langle \sigma(\alpha) \rangle\!\rangle\ (\lambda t.\ \tau'\ \sigma_t^*(t))\ s) \Leftrightarrow_{(\mathrm{IIH})} (\langle\!\langle \sigma(\alpha) \rangle\!\rangle\ \tau'\ s) \Rightarrow \tau'\ s$ which completes the induction. Then the proof of the case completes as follows:

$$
\begin{aligned}
&\langle\!\langle \alpha^* \rangle\!\rangle \ulcorner \phi \urcorner\ \sigma_s^*(s) \\
\Leftrightarrow\ & (\mu\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}\ (\ulcorner \psi \urcorner\ t \Rightarrow \tau'\ t) \star (\langle\!\langle \alpha \rangle\!\rangle\ \tau'\ t \Rightarrow \tau'\ t))\ (\sigma_s^*(s)) \\
\Leftrightarrow_*\ & (\mu\tau'' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}\ (\ulcorner \sigma(\psi) \urcorner\ t \Rightarrow \tau''\ t) \star (\langle\!\langle \sigma(\alpha) \rangle\!\rangle\ \tau''\ t \Rightarrow \tau''\ t))\ s \\
\Leftrightarrow\ & \langle\!\langle \sigma(\{\alpha^*\}) \rangle\!\rangle \ulcorner \sigma(\phi) \urcorner\ s
\end{aligned}
$$

Case α^d: Reasoning by equality, $\langle\!\langle \alpha^d \rangle\!\rangle \ulcorner \phi \urcorner\ \sigma_s^*(s) \Leftrightarrow [[\alpha]] \ulcorner \phi \urcorner\ \sigma_s^*(s) \underset{\mathrm{IH}}{\Leftrightarrow} [[\sigma(\alpha)]] \ulcorner \sigma(\phi) \urcorner\ s \Leftrightarrow \langle\!\langle \{\sigma(\alpha)\}^d \rangle\!\rangle \ulcorner \sigma(\phi) \urcorner\ s \Leftrightarrow \langle\!\langle \sigma(\{\alpha^d\}) \rangle\!\rangle \ulcorner \sigma(\phi) \urcorner\ s$.

Demon cases:

Case $y := g$: From (A2) have (L1) $y \notin Dom(\sigma)$ and (L2) σ is $\{y\}$-admissible for ϕ. Note (*) by (L2) and Lemma B.9 that $\ulcorner \phi \urcorner\ \sigma^*_{\mathsf{set}\ s\ y\ (g\ \sigma_s^*(s))}(\mathsf{set}\ s\ y\ (g\ \sigma_s^*(s))) \Leftrightarrow \ulcorner \phi \urcorner\ \sigma_s^*(\mathsf{set}\ s\ y\ (g\ \sigma_s^*(s)))$ since $s \overset{\{y\}^{\complement}}{=} \mathsf{set}\ s\ y\ (g\ \sigma_s^*(s))$. Step (1) follows from assumption (L1) that $y \notin Dom(\sigma)$.

Then we have

$$
\begin{aligned}
& [[y := g]] \ulcorner \phi \urcorner \ \sigma_s^*(s) \\
\Leftrightarrow\ & \ulcorner \phi \urcorner \ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (g\ (\sigma_s^*(s)))) \\
\Leftrightarrow_1\ & \ulcorner \phi \urcorner \ (\sigma_s^*((\mathsf{set}\ s\ y\ (g\ \sigma_s^*(s))))) \\
\Leftrightarrow_*\ & \ulcorner \phi \urcorner \ (\sigma^*_{\mathsf{set}\ s\ y\ (g\ \sigma_s^*(s))}(\mathsf{set}\ s\ y\ (g\ \sigma_s^*(s)))) \\
\underset{\text{IH}}{\Leftrightarrow}\ & \ulcorner \sigma(\phi) \urcorner \ (\mathsf{set}\ s\ y\ (g\ \sigma_s^*(s))) \\
\Leftrightarrow\ & \ulcorner \sigma(\phi) \urcorner \ (\mathsf{set}\ s\ y\ (\sigma(g)\ s)) \\
\Leftrightarrow\ & [[y := (\sigma(g))]] \ulcorner \sigma(\phi) \urcorner \ s \\
\Leftrightarrow\ & [[\sigma(y := g)]] \ulcorner \sigma(\phi) \urcorner \ s
\end{aligned}
$$

Case $y := *$: From (A2) have (L1) $x \notin Dom(\sigma)$ and (L2) σ is $\{y\}$-admissible for ϕ. Note (*) by (L2) and Lemma B.9 that $\ulcorner \phi \urcorner \ \sigma^*_{\mathsf{set}\ s\ y\ v}(\mathsf{set}\ s\ y\ v) \Leftrightarrow \ulcorner \phi \urcorner \ \sigma_s^*(\mathsf{set}\ s\ y\ v)$ for all v since $s \overset{\{y\}^{\complement}}{=} \mathsf{set}\ s\ y\ v$. Step (1) follows from assumption (L1) that $y \notin Dom(\sigma)$. Then

$$
\begin{aligned}
& [[y := *]] \ulcorner \phi \urcorner \ \sigma_s^*(s) \\
\Leftrightarrow\ & \Pi v : \mathbb{R}. \ulcorner \phi \urcorner \ (\mathsf{set}\ (\sigma_s^*(s))\ y\ v) \\
\Leftrightarrow_1\ & \Pi v : \mathbb{R}. \ulcorner \phi \urcorner \ (\sigma_s^*(\mathsf{set}\ s\ y\ v)) \\
\Leftrightarrow_*\ & \Pi v : \mathbb{R}. \ulcorner \phi \urcorner \ (\sigma^*_{\mathsf{set}\ s\ y\ v}(\mathsf{set}\ s\ y\ v)) \\
\underset{\text{IH}}{\Leftrightarrow}\ & \Pi v : \mathbb{R}. \ulcorner \sigma(\phi) \urcorner \ (\mathsf{set}\ s\ y\ v) \\
\Leftrightarrow\ & [[y := *]] \ulcorner \sigma(\phi) \urcorner \ s \\
\Leftrightarrow\ & [[\sigma(y := *)]] \ulcorner \sigma(\phi) \urcorner \ s
\end{aligned}
$$

Case $?\phi$: Have $[[?\psi]] \ulcorner \phi \urcorner \ \sigma_s^*(s) \Leftrightarrow (\ulcorner \psi \urcorner \ \sigma_s^*(s) \Rightarrow \ulcorner \phi \urcorner \ \sigma_s^*(s)) \underset{\text{IH}}{\Leftrightarrow} (\ulcorner \sigma(\psi) \urcorner \ s \Rightarrow \ulcorner \sigma(\phi) \urcorner \ s) \Leftrightarrow [[\sigma(?\psi)]] \ulcorner \sigma(\phi) \urcorner \ s$.

Case $z' = f \,\&\, \psi$: From (A2) have (L1) $y \notin Dom(\sigma)$ and (L2) σ is $\{y, y'\}$-admissible for ϕ and ψ.

Recall that we are in the case

$$
\begin{aligned}
& [[y' = g \,\&\, \psi]] \ulcorner \phi \urcorner \ \sigma_s^*(s) \\
\Leftrightarrow\ & \Pi d : \mathbb{R}_{\geq 0}. \Pi sol : [0, d] \Rightarrow \mathbb{R}. \\
& \quad (sol, \sigma_s^*(s), d \vDash y' = g) \\
& \Rightarrow (\Pi t : [0, d]. \ulcorner \psi \urcorner \ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ t))) \\
& \Rightarrow \ulcorner \phi \urcorner \ (\mathsf{set}\ (\sigma_s^*(s))\ (y, y')\ (sol\ d, g\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ d))))
\end{aligned}
$$

so the proof will proceed by showing substitution facts for the solution, domain constraint, and postcondition, from which the case will follow. Let $d \geq 0$ be the ODE duration and sol be the solution.

We will use Lemma B.9 (the adjoint lemma) several times. Since $\{y,y'\}^{\complement} \subset \{y\}^{\complement}$, note (Agree) $s \overset{\{y,y'\}^{\complement}}{=} (\mathsf{set}\ s\ y\ (sol\ t))$. Then (L2) says σ is $\{y,y'\}$-admissible for ϕ, so in combination with (Agree) it allows applying Lemma B.9 to get (*1)

$$\ulcorner\psi\urcorner\ (\sigma_s^*(\mathsf{set}\ s\ y\ (sol\ t))) \Leftrightarrow \ulcorner\psi\urcorner\ (\sigma^*_{(\mathsf{set}\ s\ y\ (sol\ t))}(\mathsf{set}\ s\ y\ (sol\ t)))$$

Note (*2) by (L2) and Lemma B.9 that

$$\begin{aligned}&\ulcorner\phi\urcorner\ (\sigma_s^*(\mathsf{set}\ s\ (y,y')\ (sol\ d, g\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ d)))))\\ \Leftrightarrow\ &\ulcorner\phi\urcorner\ (\sigma^*_{\mathsf{set}\ s\ (y,y')\ (sol\,d,g(\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ d)))}((\mathsf{set}\ s\ (y,y')\ (sol\ d, g\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ d))))))\end{aligned}$$

since $s \overset{\{y,y'\}^{\complement}}{=} \mathsf{set}\ s\ (y,y')\ (sol\ d, g\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ d)))$. Note (*3) by (L2) and Lemma B.9 that $g\ (\sigma_s^*(\mathsf{set}\ s\ y\ (sol\ d))) = g\ (\sigma^*_{\mathsf{set}\ s\ y\ (sol\ d)}(\mathsf{set}\ s\ y\ (sol\ d)))$ since $s \overset{\{y\}^{\complement}}{=} \mathsf{set}\ s\ y\ (sol\ d)$.

We are now ready to show all the main facts.

Have (0) $(sol, \sigma_s^*(s), d \vDash y' = g) = (sol, s, d \vDash \sigma((y' = g)))$ by the IH for ODE solutions.

Have (1) for all $t \in [0,d]$, $\ulcorner\psi\urcorner\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ t)) \Leftrightarrow_{(\mathrm{L1})} \ulcorner\psi\urcorner\ (\sigma_s^*(\mathsf{set}\ s\ y\ (sol\ t))) \Leftrightarrow_{(*1)} \ulcorner\psi\urcorner\ (\sigma^*_{(\mathsf{set}\ s\ y\ (sol\ t))}(\mathsf{set}\ s\ y\ (sol\ t))) \underset{\mathrm{IH}}{\Leftrightarrow} \ulcorner\sigma(\psi)\urcorner\ (\mathsf{set}\ s\ y\ (sol\ t))$.

Abbreviate $ss = (\mathsf{set}\ s\ (y,y')\ (sol\ d, g\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ d))))$. Have (2)

$$\begin{aligned}&\ulcorner\phi\urcorner\ (\mathsf{set}\ (\sigma_s^*(s))\ (y,y')\ (sol\ d, g\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ d))))\\ \Leftrightarrow_{(\mathrm{L1})}\ &\ulcorner\phi\urcorner\ (\sigma_s^*(\mathsf{set}\ s\ (y,y')\ (sol\ d, g\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ d)))))\\ \Leftrightarrow_{(*2)}\ &\ulcorner\phi\urcorner\ (\sigma^*_{\mathsf{set}\ s\ (y,y')\ (sol\ d,g\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ d)))}(ss))\\ \Leftrightarrow\ &\ulcorner\sigma(\phi)\urcorner\ (\mathsf{set}\ s\ (y,y')\ (sol\ d, g\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ d))))\\ \Leftrightarrow_{(\mathrm{L1})}\ &\ulcorner\sigma(\phi)\urcorner\ (\mathsf{set}\ s\ (y,y')\ (sol\ d, g\ (\sigma_s^*(\mathsf{set}\ s\ y\ (sol\ d)))))\\ \Leftrightarrow_{(*3)}\ &\ulcorner\sigma(\phi)\urcorner\ (\mathsf{set}\ s\ (y,y')\ (sol\ d, g\ (\sigma^*_{\mathsf{set}\ s\ y\ (sol\ d)}(\mathsf{set}\ s\ y\ (sol\ d)))))\\ \underset{\mathrm{IH}}{\Leftrightarrow}\ &\ulcorner\sigma(\phi)\urcorner\ (\mathsf{set}\ s\ (y,y')\ (sol\ d, \sigma(g)\ (\mathsf{set}\ s\ y\ (sol\ d))))\end{aligned}$$

Now the main claim follows from (1), (2), and (3):

$$\begin{aligned}&[[y' = g \,\&\, \psi]]\ \ulcorner\phi\urcorner\sigma_s^*(s)\\ \Leftrightarrow\ &\Pi d : \mathbb{R}_{\geq 0}.\, \Pi sol : [0,d] \to \mathbb{R}.\\ &\qquad (sol, \sigma_s^*(s), d \vDash y' = g)\\ &\qquad \Rightarrow (\Pi t : [0,d].\ \ulcorner\psi\urcorner\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ t)))\\ &\qquad \Rightarrow \ulcorner\phi\urcorner\ (\mathsf{set}\ (\sigma_s^*(s))\ (y,y')\ (sol\ d, g\ (\mathsf{set}\ (\sigma_s^*(s))\ y\ (sol\ d))))\\ \Leftrightarrow_{(0,1,2)}\ &\Pi d : \mathbb{R}_{\geq 0}.\, \Pi sol : [0,d] \to \mathbb{R}.\\ &\qquad (sol, s, d \vDash \sigma((y' = g)))\\ &\qquad \Rightarrow (\Pi t : [0,d].\ \ulcorner\sigma(\psi)\urcorner\ (\mathsf{set}\ s\ y\ (sol\ t)))\\ &\qquad \Rightarrow \ulcorner\sigma(\phi)\urcorner\ (\mathsf{set}\ s\ (y,y')\ (sol\ d, \sigma(g)\ (\mathsf{set}\ s\ y\ (sol\ d))))\\ \Leftrightarrow\ &[[\sigma(y' = g \,\&\, \psi)]]\ \ulcorner\sigma(\phi)\urcorner\ s\end{aligned}$$

Case $\alpha;\beta$: Admissibility condition (A2) gives that (L1) $Dom(\sigma) \cap \mathrm{BV}(\alpha;\beta) = \emptyset$, (L2) σ is $\mathrm{BV}(\sigma(\alpha))$-admissible for β, and (L3) σ is $\mathrm{BV}(\sigma(\alpha;\beta))$-admissible for ϕ. Facts (1a) and (1b) say that in the relevant steps, the postconditions of α and β respectively are only applied to states of shape $\sigma_s^*(t)$ and $\sigma_t^*(u)$ for some t, u, which follows from Lemma 5.6 and (L1). Formally, the notation $(\lambda\sigma_s^*(t).\ P\ \sigma_s^*(t))$ is definable as $(\lambda u.P\ u \star \Sigma t : \mathfrak{S}.\, u = \sigma_s^*(t))$, i.e., region $(\lambda\sigma_s^*(t).\ P\ \sigma_s^*(t))$ is a total type-level function, but $(\lambda\sigma_s^*(t).\ P\ \sigma_s^*(t))\ u$ is only inhabited if u is provably of form $\sigma_s^*(t)$. The steps (1a) and (1b) hold by Lemma 5.6 and (L1): the final state agrees with initial state $\sigma_s^*(s)$ on the substituted variables of σ, which by definition of adjoint suffices to show the final state has form $\sigma_s^*(t)$ for some state t.

Facts (IHA), (IHB), (IHP) are the IHs on α, β, and ϕ. Step (*1) says that $\sigma_s^*(t) = \sigma_t^*(t)$ because $s \overset{\mathrm{BV}(\sigma(\alpha))^{\complement}}{=} t$ by Lemma 5.6. Step (*2) says that $\sigma_t^*(u) = \sigma_u^*(u)$ because $t \overset{\mathrm{BV}(\sigma(\beta))^{\complement}}{=} u$ by Lemma 5.6.

Next, we have

$$
\begin{aligned}
& [[\alpha;\beta]]\ \ulcorner\phi\urcorner\ \sigma_s^*(s) \\
\Leftrightarrow\ & [[\alpha]]\ ([[\beta]]\ \ulcorner\phi\urcorner)\ \sigma_s^*(s) \\
\Leftrightarrow_{(1a)}\ & [[\alpha]]\ (\lambda\sigma_s^*(t).\ [[\beta]]\ \ulcorner\phi\urcorner\ \sigma_s^*(t))\ \sigma_s^*(s) \\
\Leftrightarrow_{(1b)}\ & [[\alpha]]\ (\lambda\sigma_s^*(t).\ [[\beta]]\ (\lambda\sigma_t^*(u).\ \ulcorner\phi\urcorner\ \sigma_t^*(u))\ \sigma_s^*(t))\ \sigma_s^*(s) \\
\Leftrightarrow_{(IHA)}\ & [[\sigma(\alpha)]]\ (\lambda t.\ [[\beta]]\ (\lambda\sigma_t^*(u).\ \ulcorner\phi\urcorner\ \sigma_t^*(u))\ \sigma_s^*(t))\ s \\
\Leftrightarrow_{(*1)}\ & [[\sigma(\alpha)]]\ (\lambda t.\ [[\beta]]\ (\lambda\sigma_t^*(u).\ \ulcorner\phi\urcorner\ \sigma_t^*(u))\ \sigma_t^*(t))\ s \\
\Leftrightarrow_{(IHB)}\ & [[\sigma(\alpha)]]\ (\lambda t.\ [[\sigma(\beta)]]\ (\lambda u.\ \ulcorner\phi\urcorner\ \sigma_t^*(u))\ t)\ s \\
\Leftrightarrow_{(*2)}\ & [[\sigma(\alpha)]]\ (\lambda t.\ [[\sigma(\beta)]]\ (\lambda u.\ \ulcorner\phi\urcorner\ \sigma_u^*(u))\ t)\ s \\
\Leftrightarrow_{(IHP)}\ & [[\sigma(\alpha)]]\ (\lambda t.\ [[\sigma(\beta)]]\ (\lambda u.\ \ulcorner\sigma(\phi)\urcorner\ u)\ t)\ s \\
\Leftrightarrow\ & [[\sigma(\alpha)]]\ ([[\sigma(\beta)]]\ \ulcorner\sigma(\phi)\urcorner)\ s \\
\Leftrightarrow\ & [[\sigma(\alpha;\beta)]]\ \sigma(\phi)\ s
\end{aligned}
$$

Case $\alpha \cup \beta$: In this case, have $[[\alpha \cup \beta]]\ \ulcorner\phi\urcorner\ \sigma_s^*(s) \Leftrightarrow [[\alpha]]\ \ulcorner\phi\urcorner\ \sigma_s^*(s) \star [[\beta]]\ \ulcorner\phi\urcorner\ \sigma_s^*(s) \Leftrightarrow [[\sigma(\alpha)]]\ \ulcorner\sigma(\phi)\urcorner\ s \star [[\sigma(\beta)]]\ \ulcorner\sigma(\phi)\urcorner\ s \Leftrightarrow [[\sigma(\alpha \cup \beta)]]\ \ulcorner\sigma(\phi)\urcorner\ s$.

Case α^*: Consider two fixed points:

$$
\begin{aligned}
FP_1 &\equiv (\rho\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\, \lambda t : \mathfrak{S}\, (\tau'\ t \Rightarrow [[\alpha]]\ \tau'\ t) \star (\tau'\ t \Rightarrow \ulcorner\psi\urcorner\ t)) \\
FP_2 &\equiv (\rho\tau'' : (\mathfrak{S} \Rightarrow \mathbb{T}).\, \lambda t : \mathfrak{S}\, (\tau''\ t \Rightarrow [[\sigma(\alpha)]]\ \tau''\ t) \star (\tau''\ t \Rightarrow \ulcorner\sigma(\psi)\urcorner\ t))
\end{aligned}
$$

The key fact for this case (*) is $FP_1\ \sigma_s^*(s)$ iff $FP_2\ s$ (for all $s : \mathfrak{S}$). We prove fact (*) coinductively. Technically, each direction is an coinduction on the respective fixed-point definition, but we present a single proof because both coinductions rely on primarily the same bidirectional reasoning. Let families τ' and τ'' be the variables from the fixed-point construction, for which we give a coinductive hypothesis.

Assume $FP_1\ \sigma_s^*(s)$ so that (1A) $\ulcorner\psi\urcorner\ \sigma_s^*(s)$ and (2A) $[[\alpha]]\ \tau'\ \sigma_s^*(s)$.

From (1A) have (by IH on ψ) $\ulcorner\psi\urcorner\ \sigma_s^*(s) \Leftrightarrow \ulcorner\sigma(\psi)\urcorner\ s$, so let (1B) refer to the result $\ulcorner\sigma(\psi)\urcorner\ s$.

We now work on (2A). In this case, (3) refers to the fact $\sigma_s^*(t) = \sigma_t^*(t)$ which holds by Lemma B.9 because we assumed that σ was $\mathrm{BV}(\sigma(\alpha))$-admissible in α and because by Lemma 5.6 we have $s \overset{\mathrm{BV}(\sigma(\alpha))^{\complement}}{=} t$. Fact (OIH) refers to the outer IH on α, which we generalize here to allow an arbitrary goal region, not just a CdGL formula. Fact (IIH) refers to the inner IH, which allows us to rewrite $\tau'\ \sigma_s^*(s)$ iff $\tau''\ s$, but only after executing α in order to ensure well-foundedness. Fact (adj) refers to the fact that in the relevant proof step τ' is only ever invoked on states of shape $\sigma_s^*(t)$, which is the case by Lemma 5.6 and the admissibility assumption $\mathrm{BV}(\alpha) \cap Dom(\sigma) = \emptyset$.

We now show transform (2A): $[[\alpha]]\ \tau'\ \sigma_s^*(s) \Leftrightarrow_{(\mathrm{adj})} [[\alpha]]\ (\lambda\sigma_s^*(t).\ \tau'\ \sigma_s^*(t))\ \sigma_s^*(s) \Leftrightarrow_{(\mathrm{OIH})} [[\sigma(\alpha)]]\ (\lambda t.\ \tau'\ \sigma_s^*(t))\ s \Leftrightarrow_{(3)} [[\sigma(\alpha)]]\ (\lambda t.\ \tau'\ \sigma_t^*(t))\ s \Leftrightarrow_{(\mathrm{IIH})} [[\sigma(\alpha)]]\ \tau'\ s$ so let (2B) refer to the result $[[\sigma(\alpha)]]\ \tau'\ s$.

From (1B) and (2B) have $\ulcorner\sigma(\psi)\urcorner\ s \star [[\sigma(\alpha)]]\ \tau'\ s$ which immediately implies $FP_2\ s$, which completes the coinduction.

Then full proof of the case follows:

$$
\begin{aligned}
& [[\alpha^*]]\ \ulcorner\phi\urcorner\ \sigma_s^*(s) \\
\Leftrightarrow\ & (\rho\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}\ (\tau'\ t \Rightarrow [[\alpha]]\ \tau'\ t) \star (\tau'\ t \Rightarrow \ulcorner\psi\urcorner\ t))\ (\sigma_s^*(s)) \\
\Leftrightarrow_*\ & (\rho\tau'' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}\ (\tau''\ t \Rightarrow [[\sigma(\alpha)]]\ \tau''\ t) \star (\tau''\ t \Rightarrow \ulcorner\sigma(\psi)\urcorner\ t))\ s \\
\Leftrightarrow\ & [[\sigma(\alpha^*)]]\ \ulcorner\sigma(\phi)\urcorner\ s
\end{aligned}
$$

Case α^d: Reasoning by a chain of equalities, $[[\alpha^d]]\ \ulcorner\phi\urcorner\ \sigma_s^*(s) \Leftrightarrow \langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\phi\urcorner\ \sigma_s^*(s) \underset{\mathrm{IH}}{\Leftrightarrow} \langle\!\langle\sigma(\alpha)\rangle\!\rangle\ \ulcorner\sigma(\phi)\urcorner\ s \Leftrightarrow [[\{\sigma(\alpha)\}^d]]\ \ulcorner\sigma(\phi)\urcorner\ s \Leftrightarrow [[\sigma(\alpha^d)]]\ \ulcorner\sigma(\phi)\urcorner\ s$.

□

Theorem 5.9 (Soundness)**.** *Recall that the CdGL sequent* $(\Gamma \vdash \phi)$ *is* valid *iff there exists* M *where* $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash M : (\ulcorner\phi\urcorner\ \mathfrak{s})$.

If sequent $(\Gamma \vdash \phi)$ *has a proof in CdGL, then sequent* $(\Gamma \vdash \phi)$ *is valid in CdGL. As a special case, if sequent* $(\cdot \vdash \phi)$ *has a CdGL proof, then formula* ϕ *is valid in CdGL.*

Proof. In each case, fix a context Γ and assume (A) $\ulcorner\Gamma\urcorner(\mathfrak{s})$. In cases where premises include Γ, we assume that modus ponens has been applied to all premises with (A). Additional antecedents beyond Γ will be explicitly discharged in each case. Recall that in each case, $\mathfrak{s} : \mathfrak{S}$ is a distinguished variable standing for the current state.

Case $[\cup]$I: Assume premises (0) $\Gamma \vdash [\alpha]\phi$ and (1) $\Gamma \vdash [\beta]\phi$ then by the IHs have (0A) $\ulcorner[\alpha]\phi\urcorner\ \mathfrak{s}$ and (1A) $\ulcorner[\beta]\phi\urcorner\ \mathfrak{s}$, which expand to (0B) $[[\alpha]]\ \ulcorner\phi\urcorner\ \mathfrak{s}$ and (1B) $[[\beta]]\ \ulcorner\phi\urcorner\ \mathfrak{s}$, so that by conjunction $[[\alpha]]\ \ulcorner\phi\urcorner\ \mathfrak{s} \star [[\beta]]\ \ulcorner\phi\urcorner\ \mathfrak{s}$, i.e., $[[\alpha \cup \beta]]\ \ulcorner\phi\urcorner\ \mathfrak{s}$ and $\ulcorner[\alpha \cup \beta]\phi\urcorner\ \mathfrak{s}$.

Case $[\cup]$E1: From premise, have (0) $\Gamma \vdash [\alpha \cup \beta]\phi$, by IH have (0A) $\ulcorner[\alpha \cup \beta]\phi\urcorner\ \mathfrak{s}$, which expands to $[[\alpha \cup \beta]]\ \ulcorner\phi\urcorner\ \mathfrak{s} \Leftrightarrow [[\alpha]]\ \ulcorner\phi\urcorner\ \mathfrak{s} \star [[\beta]]\ \ulcorner\phi\urcorner\ \mathfrak{s}$, whose left projection is $[[\alpha]]\ \ulcorner\phi\urcorner\ \mathfrak{s} \Leftrightarrow \ulcorner[\alpha]\phi\urcorner\ \mathfrak{s}$.

Case $[\cup]$E2: From premise, have (0) $\Gamma \vdash [\alpha \cup \beta]\phi$, by IH have (0A) $\ulcorner[\alpha \cup \beta]\phi\urcorner\ \mathfrak{s}$, which expands to $[[\alpha \cup \beta]]\ \ulcorner\phi\urcorner\ \mathfrak{s} \Leftrightarrow [[\alpha]]\ \ulcorner\phi\urcorner\ \mathfrak{s} \star [[\beta]]\ \ulcorner\phi\urcorner\ \mathfrak{s}$, whose right projection is $[[\beta]]\ \ulcorner\phi\urcorner\ \mathfrak{s} \Leftrightarrow \ulcorner[\beta]\phi\urcorner\ \mathfrak{s}$.

Case $\langle\cup\rangle$E: Assume (0) $\Gamma \vdash \langle\alpha \cup \beta\rangle\phi$ and (1) $\Gamma, \langle\alpha\rangle\phi \vdash \psi$ and (2) $\Gamma, \langle\beta\rangle\phi \vdash \psi$.

By the IHs, have

(0A) $\ulcorner\langle\alpha\cup\beta\rangle\phi\urcorner\,\mathfrak{s}$ and (1A) $\ulcorner\langle\alpha\rangle\phi\urcorner\,\mathfrak{s}\vdash\ulcorner\psi\urcorner\,\mathfrak{s}$ and (2A) $\ulcorner\langle\beta\rangle\phi\urcorner\,\mathfrak{s}\vdash\ulcorner\psi\urcorner\,\mathfrak{s}$, which expand to (0B) $\langle\!\langle\alpha\cup\beta\rangle\!\rangle\ \ulcorner\phi\urcorner\,\mathfrak{s}\Leftrightarrow\langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\phi\urcorner\,\mathfrak{s}+\langle\!\langle\beta\rangle\!\rangle\ \ulcorner\phi\urcorner\,\mathfrak{s}$ and (1B) $\langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\phi\urcorner\,\mathfrak{s}\vdash\ulcorner\psi\urcorner\,\mathfrak{s}$ and (2B) $\langle\!\langle\beta\rangle\!\rangle\ \ulcorner\phi\urcorner\,\mathfrak{s}\vdash\ulcorner\psi\urcorner\,\mathfrak{s}$.

From (0B) have two cases: (L) $\langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\phi\urcorner\,\mathfrak{s}$ or (R) $\langle\!\langle\beta\rangle\!\rangle\ \ulcorner\phi\urcorner\,\mathfrak{s}$. In the first case, apply (1B) to (L), yielding $\ulcorner\psi\urcorner\,\mathfrak{s}$, or in the second case, apply (2B) to (R), also yielding $\ulcorner\psi\urcorner\,\mathfrak{s}$ in each case as desired.

Case $\langle\cup\rangle$I1: Assume (0) $\Gamma\vdash\langle\alpha\rangle\phi$. By IH, (0A) $\ulcorner\langle\alpha\rangle\phi\urcorner\,\mathfrak{s}$, which expands to (0B) $\langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\phi\urcorner\,\mathfrak{s}$ then by left injection, $\langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\phi\urcorner\,\mathfrak{s}+\langle\!\langle\beta\rangle\!\rangle\ \ulcorner\phi\urcorner\,\mathfrak{s}\Leftrightarrow\langle\!\langle\alpha\cup\beta\rangle\!\rangle\ \ulcorner\phi\urcorner\,\mathfrak{s}\Leftrightarrow\ulcorner\langle\alpha\cup\beta\rangle\phi\urcorner\,\mathfrak{s}$.

Case $\langle\cup\rangle$I2: Assume (0) $\Gamma\vdash\langle\beta\rangle\phi$. By IH, (0A) $\ulcorner\langle\beta\rangle\phi\urcorner\,\mathfrak{s}$, which expands to (0B) $\langle\!\langle\beta\rangle\!\rangle\ \ulcorner\phi\urcorner\,\mathfrak{s}$, then by right injection, $\langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\phi\urcorner\,\mathfrak{s}+\langle\!\langle\beta\rangle\!\rangle\ \ulcorner\phi\urcorner\,\mathfrak{s}\Leftrightarrow\langle\!\langle\alpha\cup\beta\rangle\!\rangle\ \ulcorner\phi\urcorner\,\mathfrak{s}\Leftrightarrow\ulcorner\langle\alpha\cup\beta\rangle\phi\urcorner\,\mathfrak{s}$.

Case $\langle?\rangle$I: Assume (0) $\Gamma\vdash\phi$ and (1) $\Gamma\vdash\psi$. By IH, (0A) $\ulcorner\phi\urcorner\,\mathfrak{s}$ and (1A) $\ulcorner\psi\urcorner\,\mathfrak{s}$, i.e., $\langle\!\langle?\phi\rangle\!\rangle\ \ulcorner\psi\urcorner\,\mathfrak{s}\Leftrightarrow\ulcorner\langle?\phi\rangle\psi\urcorner\,\mathfrak{s}$.

Case $\langle?\rangle$E1: Assume (0) $\Gamma\vdash\langle?\phi\rangle\psi$, so by IH, (0A) $\ulcorner\langle?\phi\rangle\psi\urcorner\,\mathfrak{s}$, which expands to (0B) $\langle\!\langle?\phi\rangle\!\rangle\ \ulcorner\psi\urcorner\,\mathfrak{s}\Leftrightarrow\ulcorner\phi\urcorner\,\mathfrak{s}\mathbin{\star}\ulcorner\psi\urcorner\,\mathfrak{s}$ whose left projection is $\ulcorner\phi\urcorner\,\mathfrak{s}$ as desired.

Case $\langle?\rangle$E2: Assume (0) $\Gamma\vdash\langle?\phi\rangle\psi$, so by IH, (0A) $\ulcorner\langle?\phi\rangle\psi\urcorner\,\mathfrak{s}$, which expands to (0B) $\langle\!\langle?\phi\rangle\!\rangle\ \ulcorner\psi\urcorner\,\mathfrak{s}\Leftrightarrow\ulcorner\phi\urcorner\,\mathfrak{s}\mathbin{\star}\ulcorner\psi\urcorner\,\mathfrak{s}$ whose right projection is likewise $\ulcorner\psi\urcorner\,\mathfrak{s}$ as desired.

Case $[?]$I: Assume (0) $\Gamma,\phi\vdash\psi$. By IH, (0A) $\ulcorner\phi\urcorner\,\mathfrak{s}\vdash\ulcorner\psi\urcorner\,\mathfrak{s}$, i.e., $[[?\phi]]\ \ulcorner\psi\urcorner\,\mathfrak{s}$ which is $\ulcorner[?\phi]\psi\urcorner\,\mathfrak{s}$.

Case $[?]$E: Assume (0) $\Gamma\vdash[?\phi]\psi$ and (1) $\Gamma\vdash\phi$. By the IHs, have (0A) $\ulcorner[?\phi]\psi\urcorner\,\mathfrak{s}\Leftrightarrow[[?\phi]]\ \ulcorner\psi\urcorner\,\mathfrak{s}\Leftrightarrow(\ulcorner\phi\urcorner\,\mathfrak{s}\Rightarrow\ulcorner\psi\urcorner\,\mathfrak{s})$ and (1A) $\ulcorner\phi\urcorner\,\mathfrak{s}$. By modus ponens have $\ulcorner\psi\urcorner\,\mathfrak{s}$.

Case hyp: The side condition says conclusion ϕ satisfies $\phi\in\Gamma$. Then assumption (A) is $\ulcorner\Gamma\urcorner(\mathfrak{s})\Leftrightarrow(\mathfrak{s}:\mathfrak{S},\star_{i\in|\Gamma|}\ulcorner\psi_i\urcorner\,\mathfrak{s})\Leftrightarrow(\mathfrak{s}:\mathfrak{S},\ulcorner\psi_1\urcorner\,\mathfrak{s}\star\cdots\star\ulcorner\phi\urcorner\,\mathfrak{s}\star\cdots\star\ulcorner\psi_{|\Gamma|}\urcorner\,\mathfrak{s}$, that is, $\phi=\psi_i$ for some $i\in[1,|\Gamma|]$. By projection, $\ulcorner\phi\urcorner\,\mathfrak{s}$ as desired.

Case $[:*]$I: Assume side condition that y is fresh. WTS for all s such that (A) then $\ulcorner[x:=*]\phi\urcorner\,\mathfrak{s}$, which is $(\Pi v:\mathbb{R}.\ulcorner\phi\urcorner\,(\mathsf{set}\ \mathfrak{s}\ x\ v))$, so assume any $v:\mathbb{R}$ to prove $\ulcorner\phi\urcorner\,(\mathsf{set}\ \mathfrak{s}\ x\ v)$.

From Lemma 5.2 on (A) have $\ulcorner\Gamma\frac{y}{x}\urcorner(\mathfrak{s}\frac{y}{x})$. By Lemma 5.4 have (A1) $\ulcorner\Gamma\frac{y}{x}\urcorner(\mathsf{set}\ (\mathfrak{s}\frac{y}{x})\ x\ v)$. Then by the IH have $\ulcorner\phi\urcorner\ (\mathsf{set}\ (\mathfrak{s}\frac{y}{x})\ x\ v)$ and by Lemma 5.4 and the side condition have $\ulcorner\Gamma\urcorner(\mathsf{set}\ \mathfrak{s}\ x\ v)$. Since v was assumed to be arbitrary, we have $\ulcorner[x:=*]\phi\urcorner\,\mathfrak{s}$ as desired.

Case $\langle:*\rangle$I: Assume side condition that y is fresh. Apply Lemma 5.2 to (A) to get (A0) $\ulcorner\Gamma\frac{y}{x}\urcorner(\mathfrak{s}\frac{y}{x})$. By Lemma 5.4 have (A1) $\ulcorner\Gamma\frac{y}{x}\urcorner(\mathsf{set}\ (\mathfrak{s}\frac{y}{x})\ x\ (f\ \mathfrak{s}))$ since x is fresh in $\Gamma\frac{y}{x}$ by the side condition for y. Note also (A2) $\ulcorner x=f\frac{y}{x}\urcorner\ (\mathsf{set}\ (\mathfrak{s}\frac{y}{x})\ x\ (f\ \mathfrak{s}))$ since by construction $(\mathsf{set}\ (\mathfrak{s}\frac{y}{x})\ x\ (f\ \mathfrak{s}))\ x=(f\ \mathfrak{s})$ and $(f\ \mathfrak{s})=(f\frac{y}{x}\ \mathfrak{s}\frac{y}{x})$ by Lemma 5.2 and lastly $(f\frac{y}{x}\ \mathfrak{s}\frac{y}{x})=(f\frac{y}{x}\ (\mathsf{set}\ (\mathfrak{s}\frac{y}{x})\ x\ (f\ \mathfrak{s})))$ by Lemma 5.4 and the side condition. Apply (A1) and (A2) to the IH to get $\ulcorner\phi\urcorner\ (\mathsf{set}\ (\mathfrak{s}\frac{y}{x})\ x\ (f\ \mathfrak{s}))$ which gives $\ulcorner\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (f\ \mathfrak{s}))$ by Lemma 5.4 once more since y is fresh in ϕ. Thus, $f\ \mathfrak{s}$ is a witness to $\Sigma v:\mathbb{R}.\ulcorner\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ v)$ which is to say $\ulcorner\langle x:=*\rangle\phi\urcorner\,\mathfrak{s}$ as desired.

Case $\langle\!\langle;\rangle\!\rangle$I: We give the diamond case, the box case is symmetric. Assumption gives (0) $\Gamma\vdash\langle\alpha\rangle\langle\beta\rangle\phi$ and by IH, (0A) $\ulcorner\langle\alpha\rangle\langle\beta\rangle\phi\urcorner\,\mathfrak{s}\Leftrightarrow\langle\!\langle\alpha\rangle\!\rangle\ (\langle\!\langle\beta\rangle\!\rangle\ \ulcorner\phi\urcorner)\,\mathfrak{s}\Leftrightarrow\langle\!\langle\alpha;\beta\rangle\!\rangle\ \ulcorner\phi\urcorner\,\mathfrak{s}\Leftrightarrow\ulcorner\langle\alpha;\beta\rangle\phi\urcorner\,\mathfrak{s}$.

Case $\langle\!\langle:=\rangle\!\rangle$I: We give the diamond case, the box case is symmetric. Assume y fresh. Assume (0) $\Gamma\frac{y}{x},x=f\frac{y}{x}\vdash\phi$. By Lemma 5.2 on (A) have (A1) $\ulcorner\Gamma\frac{y}{x}\urcorner(\mathfrak{s}\frac{y}{x})$ and by Lemma 5.4 have (A2) $\ulcorner(\Gamma\frac{y}{x})\urcorner(\mathsf{set}\ (\mathfrak{s}\frac{y}{x})\ x\ (f\ \mathfrak{s}))$ since x is fresh in $\Gamma\frac{y}{x}$ by the freshness assumption on y. Note (A3) $\ulcorner x=f\frac{y}{x}\urcorner\ (\mathsf{set}\ (\mathfrak{s}\frac{y}{x})\ x\ (f\ \mathfrak{s}))$ since $(\mathsf{set}\ (\mathfrak{s}\frac{y}{x})\ x\ (f\ \mathfrak{s}))\ x=f\ \mathfrak{s}$ by construction

and $f\ \mathfrak{s} = f\frac{y}{x}\ \mathfrak{s}\frac{y}{x}$ Lemma 5.2 and lastly $f\frac{y}{x}\ \mathfrak{s}\frac{y}{x} = f\frac{y}{x}\ (\mathsf{set}\ (\mathfrak{s}\frac{y}{x})\ x\ (f\ \mathfrak{s}))$ by Lemma 5.4.

Apply the IH and plug in (A2) and (A3) to get (0A) $\ulcorner\phi\urcorner\ (\mathsf{set}\ (\mathfrak{s}\frac{y}{x})\ x\ (f\ \mathfrak{s}))$. By Lemma 5.4 again have (0B)$\ulcorner\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (f\ \mathfrak{s}))$ since y is fresh in ϕ. We immediately have $\ulcorner\langle x := f\rangle\phi\urcorner\ \mathfrak{s}$ from (0B), which is what we want to show.

Case $\langle{:}*\rangle$E: Assume side condition that y is fresh and $x \notin \mathrm{FV}(\psi)$. Assumed (0) $\Gamma \vdash \langle x := *\rangle\phi$ and (1) $\Gamma\frac{y}{x}, \phi \vdash \psi$. By the first IH have (0A) $\ulcorner\langle x := *\rangle\phi\urcorner\ \mathfrak{s}$ which expands to $(\Sigma v : \mathbb{R}.\ \ulcorner\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ v))$. Unpack $v : \mathbb{R}$ from (0A) so that (0B) $\ulcorner\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ v)$.

From Lemma 5.2 on (A) have $\ulcorner\Gamma\frac{y}{x}\urcorner(\mathfrak{s}\frac{y}{x})$ and by Lemma 5.4 have (A0) $\ulcorner\Gamma\frac{y}{x}\urcorner(\mathsf{set}\ (\mathfrak{s}\frac{y}{x})\ x\ v)$. By Lemma 5.4 on (0B) have (A1) $\ulcorner\phi\urcorner\ (\mathsf{set}\ (\mathfrak{s}\frac{y}{x})\ x\ v)$ since y is fresh. Then (A0) and (A1) allow the second premise to be applied, giving $\ulcorner\psi\urcorner\ (\mathsf{set}\ (\mathfrak{s}\frac{y}{x})\ x\ v)$. Note the side condition that x and y are both fresh in ψ so $\ulcorner\psi\urcorner\ \mathfrak{s}$ by Lemma 5.4, completing the case.

Case $[{:}*]$E: Assume side condition that ϕ_x^f is admissible. Assume the first premise (0) $\Gamma \vdash [x := *]\phi$ so by IH (0A) $\ulcorner[x := *]\phi\urcorner\ \mathfrak{s}$ so (1) $\ulcorner\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ v)$ for all $v : \mathbb{R}$. Pick $v = (f\ \mathfrak{s})$, then by Lemma 5.7 since ϕ_x^f is admissible have $\ulcorner\phi_x^f\urcorner\ \mathfrak{s}$ as desired.

Case M: Assume (0) $\Gamma \vdash \langle\alpha\rangle\phi$ and (1) $\Gamma\frac{\vec{y}}{\mathrm{BV}(\alpha)}, \phi \vdash \psi$

Let x be the vector of variables $x_i \in \mathrm{BV}(\alpha)$ in some canonical (e.g. lexicographic) order. Let y be a fresh vector of variables of the same length as x. Let z be $\mathrm{FV}(\Gamma) \setminus (\mathrm{BV}(\alpha))$.

Recall the discrete ghost rule defined in Chapter 4:

$$\text{(iG)}\quad \frac{\Gamma, p : x = f \vdash \phi}{\Gamma \vdash \phi}\quad \text{where } x \text{ fresh except free in } M,\ p \text{ fresh}$$

Applying rule iG repeatedly to (1) have $\Gamma, y = x \vdash \langle\alpha\rangle\phi$. The IH applies because the context $(\Gamma, y = x)$ is satisfied by taking assumption (A) $\ulcorner\Gamma\urcorner(s)$ and showing (AA) $\ulcorner\Gamma\urcorner(\mathsf{set}\ \mathfrak{s}\ y\ x)$ which reflexivity satisfies $y = x$ while preserving Γ by Lemma 5.4 since definitionally $y \cap \mathrm{FV}(\Gamma) = \emptyset$. The IH gives $\ulcorner\langle\alpha\rangle\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ y\ (\mathfrak{s}\ x)) \Leftrightarrow \langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ y\ (\mathfrak{s}\ x))$, then apply Lemma 5.6 with $V = \{y, z\}$ giving (B) $\langle\!\langle\alpha\rangle\!\rangle\ (\lambda t.\ \ulcorner\phi\urcorner\ t\ \star(\mathsf{set}\ \mathfrak{s}\ y\ (\mathfrak{s}\ x)) \overset{V}{=} t)\ (\mathsf{set}\ \mathfrak{s}\ y\ (\mathfrak{s}\ x))$. For short, abbreviate $\hat{\mathfrak{s}} = (\mathsf{set}\ \mathfrak{s}\ y\ (\mathfrak{s}\ x))$ so that (B1) $\langle\!\langle\alpha\rangle\!\rangle\ (\lambda t.\ \ulcorner\phi\urcorner\ t\ \star(\hat{\mathfrak{s}} \overset{V}{=} t)\ (\hat{\mathfrak{s}}))$. To prove the assumption of (1) from (B), first prove a monotone lemma (Mon):

Claim 2 (Mon). Assume $\ulcorner\Gamma\urcorner(s)$. Then for all $t : \mathfrak{S}$ have $\ulcorner\phi\urcorner\ t\star(\hat{\mathfrak{s}} \overset{V}{=} t) \Rightarrow \ulcorner\phi\urcorner\ t\star\ulcorner\Gamma\frac{\vec{y}}{\mathrm{BV}(\alpha)}\urcorner(t)$

Subproof. The first conjunct holds trivially from the assumption. For the latter, decompose Γ into two contexts $\Gamma = \Delta_1, \Delta_2$ such that context Δ_1 contains formulas ψ such that $\mathrm{FV}(\psi) \cap \mathrm{BV}(\alpha) = \emptyset$, while all other formulas are in Δ_2.

We have $\ulcorner\Delta_1\urcorner\ t$ from (A) by repeating Lemma 5.4 because for each $\psi \in \Delta_1$, we have $\hat{\mathfrak{s}} \overset{\mathrm{FV}(\psi)}{=} t$ as a consequent of $\hat{\mathfrak{s}} \overset{V}{=} t$ specifically because $V \supseteq z \supseteq \mathrm{FV}(\psi)$.

Consider Δ_2 next. Recall $\Delta_2\frac{\vec{y}}{\mathrm{BV}(\alpha)} = \wedge_{\psi\in\Delta_2}(\psi\frac{y}{x})$. For each such ψ we appeal $\hat{\mathfrak{s}}\ y = t\ y$ (since $y \subseteq V$) and $\hat{\mathfrak{s}}\ y = \mathfrak{s}\ x$ by ghosting, thus $t\ y = \mathfrak{s}\ x$ by transitivity.

To show the desired $\ulcorner\psi\frac{y}{x}\urcorner\ t$ we note by Lemma 5.2 it suffices to show $\ulcorner\psi\urcorner\ t\frac{y}{x}$, which, because $x \in \mathrm{FV}(\psi)$ and $y \notin \mathrm{FV}(\psi)$, reduces to $\ulcorner\psi\urcorner(\mathsf{set}\ t\ x\ (\mathfrak{s}\ y))$.

Since (AA) includes $\ulcorner\psi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ y\ x)$, it suffices to apply Lemma 5.4, providing the assumption that for $\mathrm{FV}(\psi) \subseteq \{z\} \cup \mathrm{BV}(\psi) = \{x, z\}$ we have $(\mathsf{set}\ \mathfrak{s}\ y\ (\mathfrak{s}\ x)) \overset{\mathrm{FV}(\psi)}{=} (\mathsf{set}\ t\ x\ (\mathfrak{s}\ y))$.

We already have $\mathfrak{s} \overset{\{y,z\}}{=} t$. Then $\mathfrak{s}\ y$ is set to $\mathfrak{s}\ x$ but $\mathfrak{s}\ x = \mathfrak{s}\ y$ anyway, a no-op. $t\ x$ is set to $\mathfrak{s}\ y = \mathfrak{s}\ x$ so that $(\mathsf{set}\ \mathfrak{s}\ y\ (\mathfrak{s}\ x)) \overset{\{x,y,z\}}{=} (\mathsf{set}\ t\ x\ (\mathfrak{s}\ y))$ as desired. □

We observe the first assumption of (Mon) is just (A), call the resulting (universal) statement (Mono). By Lemma 5.1 on (Mono) and (B) have $\langle\!\langle\alpha\rangle\!\rangle\ (\lambda t.\ \ulcorner\phi\urcorner\ t \star \ulcorner\Gamma_{\overline{\mathrm{BV}(\alpha)}}^{\vec{y}}\urcorner(t))\ \mathfrak{s}$ so we reach (Mid) $\langle\!\langle\alpha\rangle\!\rangle\ (\lambda t.\ \ulcorner\phi\urcorner\ t \star \ulcorner\Gamma_{\overline{\mathrm{BV}(\alpha)}}^{\vec{y}}\urcorner(t))\ (\mathsf{set}\ \mathfrak{s}\ y\ (\mathfrak{s}\ x))$. Commuting ϕ and $\Gamma_{\overline{\mathrm{BV}(\alpha)}}^{\vec{y}}$ in (Mid) gives the precondition for Lemma 5.1 on (1), yielding $\langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\psi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ y\ (\mathfrak{s}\ x))$. Lastly, y is fresh in α and ϕ, so Lemma 5.5 gives the desired $\langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\psi\urcorner\ \mathfrak{s}$.

Case $\langle\!\langle^d]\!\rangle$I: We give the case for diamonds, the case for boxes is symmetric. Assumed (0) $\Gamma \vdash [\alpha]\phi$, then by IH have (0A) $\ulcorner[\alpha]\phi\urcorner\ \mathfrak{s} \Leftrightarrow [[\alpha]]\ \ulcorner\phi\urcorner\ \mathfrak{s} \Leftrightarrow \langle\!\langle\alpha^d\rangle\!\rangle\ \ulcorner\phi\urcorner\ \mathfrak{s} \Leftrightarrow \ulcorner\langle\alpha^d\rangle\phi\urcorner\ \mathfrak{s}$ as desired, completing the case.

Case $\langle*\rangle$C: Have assumptions (0) $\Gamma \vdash \langle\alpha^*\rangle\phi$ (1) $\Gamma, \phi \vdash \psi$ (2) $\Gamma, \langle\alpha\rangle\langle\alpha^*\rangle\phi \vdash \psi$ so by IH have (0A) $\ulcorner\langle\alpha^*\rangle\phi\urcorner\ \mathfrak{s}$ (1A) $\ulcorner\phi\urcorner\ \mathfrak{s} \vdash \ulcorner\psi\urcorner\ \mathfrak{s}$ (2A) $\ulcorner\langle\alpha\rangle\langle\alpha^*\rangle\phi\urcorner\ \mathfrak{s} \vdash \ulcorner\psi\urcorner\ \mathfrak{s}$.

From (0A) and the α^* semantics, since s belongs to the least fixed-point construction

$$(\mu\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}\ (\ulcorner\psi\urcorner\ t \Rightarrow \tau'\ t) \star (\langle\!\langle\alpha\rangle\!\rangle\ \tau'\ t \Rightarrow \tau'\ \mathfrak{s}))$$

then we have (3)$\ulcorner\psi\urcorner\ \mathfrak{s} + (\langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\langle\alpha^*\rangle\phi\urcorner\ \mathfrak{s} \Rightarrow \tau'\ \mathfrak{s})$, so we proceed on cases on (3). In case (L) $\ulcorner\psi\urcorner\ \mathfrak{s}$ apply modus ponens on (1A) giving $\ulcorner\psi\urcorner\ \mathfrak{s}$ as desired. In case (R) $\langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\langle\alpha^*\rangle\phi\urcorner\ \mathfrak{s} \Rightarrow \tau'\ \mathfrak{s}$ thus $\ulcorner\langle\alpha\rangle\langle\alpha^*\rangle\phi\urcorner\ \mathfrak{s}$ and by modus ponens on (2A) have $\ulcorner\psi\urcorner\ \mathfrak{s}$ as desired.

Case $[*]$E: Assume (0) $\Gamma \vdash [\alpha^*]\phi$ and by IH (0A) $\ulcorner[\alpha^*]\phi\urcorner\ \mathfrak{s}$. Since $\mathfrak{s}$ belongs to the fixed point $(\rho\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}\ (\tau'\ t \Rightarrow [[\alpha]]\ \tau'\ t) \star (\tau'\ t \Rightarrow \ulcorner\phi\urcorner\ t))$ then (1) $[[\alpha]]\ \ulcorner[\alpha^*]\phi\urcorner\ \mathfrak{s} \star \ulcorner\phi\urcorner\ \mathfrak{s}$. Fact (1) then commutes to $\ulcorner\phi\urcorner\ \mathfrak{s} \star [[\alpha]]\ \ulcorner[\alpha^*]\phi\urcorner\ \mathfrak{s}$ which simplifies to $\ulcorner\phi \wedge [\alpha][\alpha^*]\phi\urcorner\ \mathfrak{s}$.

Case $\langle*\rangle$S: Assume (0) $\Gamma \vdash \phi$ and by IH (0A) $\ulcorner\phi\urcorner\ \mathfrak{s}$, thus $\mathfrak{s}$ belongs to the fixed point $(\mu\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}\ (\ulcorner\psi\urcorner\ t \Rightarrow \tau'\ t) \star (\langle\!\langle\alpha\rangle\!\rangle\ \tau'\ t \Rightarrow \tau'\ t))$ and $\ulcorner\langle\alpha^*\rangle\phi\urcorner\ \mathfrak{s}$ as desired.

Case $\langle*\rangle$G: Assume (0) $\Gamma \vdash \langle\alpha\rangle\langle\alpha^*\rangle\phi$ and by IH (0A) $\ulcorner\langle\alpha\rangle\langle\alpha^*\rangle\phi\urcorner\ \mathfrak{s}$, thus $\langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\langle\alpha^*\rangle\phi\urcorner\ \mathfrak{s}$, thus $\mathfrak{s}$ belongs to $(\mu\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}\ (\ulcorner\psi\urcorner\ t \Rightarrow \tau'\ t) \star (\langle\!\langle\alpha\rangle\!\rangle\ \tau'\ t \Rightarrow \tau'\ \mathfrak{s}))$ and $\ulcorner\langle\alpha^*\rangle\phi\urcorner\ \mathfrak{s}$ as desired, completing the case.

Case $[*]$R: Assume (0) $\Gamma \vdash \phi \wedge [\alpha][\alpha^*]\phi$ and by IH (0A) $\ulcorner\phi \wedge [\alpha][\alpha^*]\phi\urcorner\ \mathfrak{s}$ so $\ulcorner\phi\urcorner\ \mathfrak{s}$ and $[[\alpha]]\ \ulcorner[\alpha^*]\phi\urcorner\ \mathfrak{s}$ so $\mathfrak{s}$ belongs to the fixed point $(\rho\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}\ (\tau'\ t \Rightarrow [[\alpha]]\ \tau'\ t) \star (\tau'\ t \Rightarrow \ulcorner\phi\urcorner\ t))$ thus $\ulcorner[\alpha^*]\phi\urcorner\ \mathfrak{s}$.

Case $[*]$I: Assume (0) $\Gamma \vdash \psi$ (1) $\psi \vdash [\alpha]\psi$. (2) $\psi \vdash \phi$. By IHs, have (0A) $\ulcorner\psi\urcorner\ \mathfrak{s}$ and (1A) $\ulcorner\psi\urcorner\ \mathfrak{s} \vdash \ulcorner[\alpha]\psi\urcorner\ \mathfrak{s}$ and (2A) $\ulcorner\psi\urcorner\ \mathfrak{s} \vdash \ulcorner\phi\urcorner\ \mathfrak{s}$. From (1A) and (2A), $\mathfrak{s}$ is a fixed point τ' of the equivalence $\forall t,\ ((\tau'\ t) = ((\tau'\ t \Rightarrow [[\alpha]]\ \tau'\ t) \star (\tau'\ t \Rightarrow \ulcorner\phi\urcorner\ t)))$, and from (0A) we have that $\mathfrak{s}$ belongs to $\ulcorner\psi\urcorner$, which is a (non-strict) subregion of the greatest fixed point. Then $\mathfrak{s}$ belongs also to the greatest fixed point, that is $\mathfrak{s}$ belongs to $(\rho\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}\ (\tau'\ t \Rightarrow [[\alpha]]\ \tau'\ t) \star (\tau'\ t \Rightarrow \ulcorner\phi\urcorner\ t))$, which is to say $\ulcorner[\alpha^*]\phi\urcorner\ \mathfrak{s}$.

Case FP: Assume (0) $\Gamma \vdash \langle\alpha^*\rangle\phi$ (1) $\phi \vdash \psi$ (2) $\langle\alpha\rangle\psi \vdash \psi$ so by IH have (0A) $\ulcorner\langle\alpha^*\rangle\phi\urcorner\ \mathfrak{s}$ (1A) $\ulcorner\phi\urcorner\ \mathfrak{s} \vdash \ulcorner\psi\urcorner\ \mathfrak{s}$ (2A) $\ulcorner\langle\alpha\rangle\psi\urcorner\ \mathfrak{s} \vdash \ulcorner\psi\urcorner\ \mathfrak{s}$. (0A) simplifies to (3)

$$(\mu\tau' : (\mathfrak{S} \Rightarrow \mathbb{T}).\ \lambda t : \mathfrak{S}\ (\ulcorner\phi\urcorner\ t \Rightarrow \tau'\ t) \star (\langle\!\langle\alpha\rangle\!\rangle\ \tau'\ t \Rightarrow \tau'\ t))\ \mathfrak{s}$$

from which we proceed by induction on the membership of $\mathfrak{s}$ in the fixed point.

In the first case, (L) $\ulcorner\phi\urcorner\ \mathfrak{s}$ so by modus ponens with (1) have $\ulcorner\psi\urcorner\ \mathfrak{s}$. In the second case have $(\langle\!\langle\alpha\rangle\!\rangle\ \tau'\ \mathfrak{s})$ where $\tau'\ \mathfrak{s} = \ulcorner\psi\urcorner\ \mathfrak{s}$, so $\langle\!\langle\alpha\rangle\!\rangle\ \ulcorner\psi\urcorner\ \mathfrak{s}$ and $\ulcorner\langle\alpha\rangle\psi\urcorner\ \mathfrak{s}$ and by modus ponens on (2A) have $\ulcorner\psi\urcorner\ \mathfrak{s}$. So in each case the conclusion $\ulcorner\psi\urcorner\ \mathfrak{s}$ holds.

Case $\langle*\rangle$I: Assume (0) $\Gamma \vdash \varphi$ (1) $\varphi, (\mathcal{M}_0 = \mathcal{M} \succ \mathbf{0}) \vdash \langle\alpha\rangle(\varphi \wedge \mathcal{M}_0 \succ \mathcal{M})$ (2) $(\varphi \wedge \mathbf{0} \succcurlyeq \mathcal{M}) \vdash \phi$. By the IH have (0A) $\ulcorner\varphi\urcorner\ \mathfrak{s}$ and have (1A)

$$\ulcorner\varphi\urcorner\ \mathfrak{s}, \ulcorner\mathcal{M}_0 = \mathcal{M} \succ \mathbf{0}\urcorner\ \mathfrak{s} \vdash \ulcorner\langle\alpha\rangle(\varphi \wedge \mathcal{M}_0 \succ \mathcal{M})\urcorner\ \mathfrak{s}$$

and have (2A) $\ulcorner\varphi \wedge \mathbf{0} \succcurlyeq \mathcal{M}\urcorner\ \mathfrak{s} \vdash \ulcorner\phi\urcorner\ \mathfrak{s}$.

By the side condition, the order $\succ$ is effectively well-founded and thus satisfies the effective descending chain condition, meaning it is constructively provable that all strictly descending chains are finite. Our main proof (Main) shows that $\mathfrak{s}$ belongs to the fixed point $\ulcorner\langle\alpha^*\rangle(\varphi \wedge \mathbf{0} \succcurlyeq \mathcal{M})\urcorner\ \mathfrak{s}$. We proceed by induction on the value of $\mathcal{M}\ \mathfrak{s}$; by the descending chain condition, this induction is well-founded. Specifically, we show that $\mathcal{M}\ \mathfrak{s}$ decreases with every iteration α until $\mathcal{M} = \mathbf{0}$ where the loop terminates. Invariant φ is maintained in every iteration.

For the base case, (0A) gave $\ulcorner\varphi\urcorner\ \mathfrak{s}$ and by conjunction we have $\ulcorner(\varphi \wedge \mathbf{0} \succcurlyeq \mathcal{M})\urcorner\ \mathfrak{s}$ which fulfills the base case of the fixed point.

In the inductive case have (case) $\mathcal{M}_0 = \mathcal{M} \succ \mathbf{0}$ and still have assumption $\ulcorner\varphi\urcorner\ \mathfrak{s}$. Let m be the constant term equal to $\mathcal{M}\ \mathfrak{s}$. Introduce a fresh discrete ghost (rule iG) $\mathcal{M}_0 = m$. From (case) we have the following fact $\ulcorner\varphi \wedge (\mathcal{M}_0 = \mathcal{M} \succ \mathbf{0})\urcorner\ (\mathsf{set}\ \mathfrak{s}\ \mathcal{M}_0\ (\mathcal{M}\ s))$ by Lemma 5.4 since $\mathcal{M}_0$ was fresh in φ. By modus ponens, applying (1A) yields the formula $\ulcorner\langle\alpha\rangle(\varphi \wedge \mathcal{M}_0 \succ \mathcal{M})\urcorner\ (\mathsf{set}\ \mathfrak{s}\ \mathcal{M}_0\ (\mathcal{M}\ s))$. By Lemma 5.7, have $\ulcorner\langle\alpha\rangle(\varphi \wedge m \succ \mathcal{M})\urcorner\ \mathfrak{s}$. Having eliminated variable $\mathcal{M}_0$ we apply Lemma 5.4 again to get (4) $\ulcorner\langle\alpha\rangle(\varphi \wedge m \succ \mathcal{M})\urcorner\ \mathfrak{s}$. Since every point t satisfying $\ulcorner(\varphi \wedge m \succ \mathcal{M})\urcorner\ t$ trivially has $\ulcorner m \succ \mathcal{M}\urcorner t$ we can apply the IH on region $\ulcorner(\varphi \wedge \mathcal{M}_0 \succ \mathcal{M})\urcorner$, yielding for all t (where $\mathcal{M}_0\ t = \mathcal{M}\ \mathfrak{s}$) that $\ulcorner(\varphi \wedge \mathcal{M}_0 \succ \mathcal{M})\urcorner\ t \vdash \ulcorner\langle\alpha^*\rangle(\varphi \wedge \mathbf{0} \succcurlyeq \mathcal{M})\urcorner\ t$. Then Lemma 5.1 gives (5) $\ulcorner\langle\alpha\rangle\langle\alpha^*\rangle(\varphi \wedge \mathbf{0} \succ \mathcal{M})\urcorner\ (\mathsf{set}\ s\ \mathcal{M}_0\ (\mathcal{M}\ \mathfrak{s}))$ as a result, which satisfies the inductive case, giving

$$\ulcorner\langle\alpha^*\rangle(\varphi \wedge \mathbf{0} \succ \mathcal{M})\urcorner(\mathsf{set}\ \mathfrak{s}\ \mathcal{M}_0\ (\mathcal{M}\ \mathfrak{s}))$$

By freshness of $\mathcal{M}_0$, Lemma 5.4 gives $\ulcorner\langle\alpha^*\rangle(\varphi \wedge \mathbf{0} \succ \mathcal{M})\urcorner\ \mathfrak{s}$. This completes the inner induction. By Lemma 5.1 and (2A) have $\ulcorner\langle\alpha^*\rangle\phi\urcorner\ \mathfrak{s}$ as desired.

Case DI: Recall the notation $Sol(t) = \mathsf{set}\ \mathfrak{s}\ (x, x')\ (sol\ t, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t)))$ from Lemma B.7 for readability. Assume (0) $\Gamma \vdash \phi$ (1) $\Gamma \vdash \forall x\,(\psi \to [x' := f](\phi)')$ so by IH have (I0) $\ulcorner\phi\urcorner\ \mathfrak{s}$ and (2) for all $v : \mathbb{R}$, $\ulcorner\psi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ v) \Rightarrow \ulcorner(\phi)'\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (x, x')\ (v, f\ (\mathsf{set}\ \mathfrak{s}\ x\ v)))$. To show the conclusion, first assume d and sol such that (I1) $(sol, \mathfrak{s}, d \vDash x' = f)$ and (4) $\ulcorner\psi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t))$ for all $t \in [0, d]$ then want $\ulcorner\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (x, x')\ (sol\ t, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t))))$ for all $t \in [0, d]$, that is, (P) $\ulcorner\phi\urcorner\ (Sol(t))$ for all $t \in [0, d]$. In showing (P), we observe from (2) and (4) that we have $\ulcorner(\phi)'\urcorner\ (\mathsf{set}\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t))\ x'\ (f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t))))$ for all $t \in [0, d]$ or equivalently $\ulcorner(\phi)'\urcorner\ (Sol(t))$ (I2) for all $t \in [0, d]$ by unpacking the definition of $(sol, \mathfrak{s}, d \vDash x' = f)$ in (I1).

We proceed to prove the conclusion (P) by induction on differentiable formula ϕ. Specifically, we prove by induction on ϕ that assumptions (I0), (I1), and (I2) imply (P). Each case reduces to a lemma proven elsewhere in the literature. However, those prior works

appeal to the time-derivative of a term f, while our term $(f)'$ is a spatial derivative. The differential lemma (Lemma B.7) shows that these derivatives are equal in the postcondition of an ODE, thus the lemmas are applicable.

Case $g = h$: We start by letting $G(t) = g\ (Sol(t))$ and letting $H(t) = h\ (Sol(t))$. In this case (I0) specializes to $g\ \mathfrak{s} = h\ \mathfrak{s}$. Because $G(0) = g\ (Sol(0)$ and $H(0) = h\ (Sol(0))$ we can then show (Init) $G(0) = H(0)$ using Lemma 5.4 by showing the precondition that $Sol(0) = \mathfrak{s}$ on $\mathrm{FV}(g) \cup \mathrm{FV}(h)$. The precondition holds by noting $Sol(0) = \mathfrak{s}$ on $\mathcal{V} \setminus \{x, x'\}$ by construction and $Sol(0) = \mathfrak{s}$ on $\{x\}$ because $sol\ 0 = \mathfrak{s}\ x$ by (I1), and lastly $x' \notin \mathrm{FV}(g) \cup \mathrm{FV}(h)$ by syntactic constraint for differentiable terms. Then, (I2) specializes for all $t \in [0, d]$ to $(g)'\ (Sol(t)) = (h)'\ (Sol(t))$, that is (EqT) $G'(t) = H'(t)$ for $t \in [0, d]$. Apply the $=$ case of Theorem B.5 to (Init) and (EqT) on interval $[0, d]$, thus $G(t) = H(t)$ for $t \in [0, d]$. The $=$ case of Theorem B.5 is applicable because its assumption is that $(F)' = (G)'$ on $[0, d]$. Therefore (Post) $\ulcorner g = h \urcorner\ (Sol(t))$. Since (Post) holds under the assumptions (I1) and (4) then $\ulcorner [x' = f \,\&\, \psi] g = h \urcorner\ \mathfrak{s}$ as desired.

Case $g > h$:

Let $F(t) = g\ (Sol(t)) - h\ (Sol(t))$. In this case (I0) specializes to $g\ \mathfrak{s} > h\ \mathfrak{s}$. Then by Lemma 5.4 have (Init) $F(0) > 0$ since because $\mathfrak{s} = Sol(0)$ on $\mathrm{FV}(g) \cup \mathrm{FV}(h)$, specifically they agree on x because $sol\ 0 = \mathfrak{s}\ x$ by (I1), they agree on $\mathcal{V} \setminus \{x, x'\}$ by construction, and by syntactic constraint on differentiable terms have $x' \notin \mathrm{FV}(g) \cup \mathrm{FV}(h)$. Then, (I2) specializes for all $t \in [0, d]$ to $(g)'\ (Sol(t)) > (h)'\ (Sol(t))$ so that (GrT) $F'(t) > 0$ for $t \in [0, d]$. Apply the $>$ case of Theorem B.5 to (Init) and (GrT) on interval $[0, d]$, thus $F(d) > F(0)$ so by transitivity with (Init), have $F(d) > 0$ so (Post) $\ulcorner g > h \urcorner\ (Sol(t))$. Specifically, the $>$ case of Theorem B.5 applies because its precondition is that $0 < (F)'$ on $[0, d]$. Since (Post) holds under the assumptions (I1) and (4) then $\ulcorner [x' = f \,\&\, \psi] g > h \urcorner\ \mathfrak{s}$ as desired.

Case $g \geq h$: Let $F(t) = g\ (Sol(t)) - h\ (Sol(t))$. In this case (I0) specializes to $g\ \mathfrak{s} \geq h\ \mathfrak{s}$. Then by Lemma 5.4 have (Init) $F(0) \geq 0$ because $\mathfrak{s} = Sol(0)$ on $\mathrm{FV}(g) \cup \mathrm{FV}(h)$, specifically they agree on x because $sol\ 0 = \mathfrak{s}\ x$ by (I1), they agree on $\mathcal{V} \setminus \{x, x'\}$ by construction, and by syntactic constraint on differentiable terms have $x' \notin \mathrm{FV}(g) \cup \mathrm{FV}(h)$. Then, (I2) specializes for all $t \in [0, d]$ to $(g)'\ (Sol(t)) \geq (h)'\ (Sol(t))$, so that (GeqT) $F'(t) \geq 0$ for $t \in [0, d]$. Apply the $\geq$ case of Theorem B.5 to (Init) and (GeqT) on interval $[0, d]$, thus $F(d) \geq F(0)$ so by transitivity with (Init), have $F(d) \geq 0$ so (Post) $\ulcorner g \geq h \urcorner\ (Sol(t))$. Specifically, the $\geq$ case of Theorem B.5 applies because its precondition is that $0 \leq (F)'$ on $[0, d]$.

Since (Post) holds assuming (I1) and (4), then $\ulcorner [x' = f \,\&\, \psi] g \geq h \urcorner\ \mathfrak{s}$ as desired.

Case $\phi_1 \wedge \phi_2$: In this case (I0) specializes to $\ulcorner \phi_1 \wedge \phi_2 \urcorner\ \mathfrak{s}$ so (L0) $\ulcorner \phi_1 \urcorner\ \mathfrak{s}$ and (R0) $\ulcorner \phi_2 \urcorner\ \mathfrak{s}$. Then since $(\phi_1 \wedge \phi_2)' = (\phi_1)' \wedge (\phi_2)'$ we also have (L2) $\ulcorner (\phi_1)' \urcorner\ (Sol(t))$ and (R2) $\ulcorner (\phi_2)' \urcorner\ (Sol(t))$. By the IH on (L0) (I1) (L2) and have (IHL) $\ulcorner [x' = f \,\&\, \psi] \phi_1 \urcorner\ \mathfrak{s}$. By the IH on (R0) (I1) (R2) and then we have (IHR) $\ulcorner [x' = f \,\&\, \psi] \phi_2 \urcorner\ \mathfrak{s}$. Next we apply assumptions (I1) and (4) to (IHL) and (IHR) so that we get the facts (L3) $\ulcorner \phi_1 \urcorner\ (Sol(t))$ and (R3) $\ulcorner \phi_2 \urcorner\ (Sol(t))$ so (LR) $\ulcorner \phi_1 \wedge \phi_2 \urcorner\ (Sol(t))$ thus $\ulcorner [x' = f \,\&\, \psi] (\phi_1 \wedge \phi_2) \urcorner\ \mathfrak{s}$ as desired.

Case $\phi_1 \vee \phi_2$: Since by definition $(\phi_1 \vee \phi_2)' = (\phi_1)' \wedge (\phi_2)'$ then we have the facts (L2) $\ulcorner (\phi_1)' \urcorner\ (Sol(t))$ and (R2) $\ulcorner (\phi_2)' \urcorner\ (Sol(t))$. By the inductive hypothesis on (I1) (L2) we have that (IHL) $\ulcorner \phi_1 \urcorner\ \mathfrak{s} \vdash \ulcorner [x' = f \,\&\, \psi] \phi_1 \urcorner\ \mathfrak{s}$. By the IH on (I1) (R2) have (IHR) $\ulcorner \phi_2 \urcorner\ \mathfrak{s}$

$\vdash \ulcorner[x' = f \,\&\, \psi]\phi_2\urcorner\ \mathfrak{s}$. In this case (I0) specializes to $\ulcorner\phi_1 \vee \phi_2\urcorner\ \mathfrak{s}$, proceed by cases. First case: (L0) $\ulcorner\phi_1\urcorner\ \mathfrak{s}$, by (IHL) have $\ulcorner[x' = f \,\&\, \psi]\phi_1\urcorner\ \mathfrak{s}$ thus by monotonicity (Lemma 5.1) $\ulcorner[x' = f \,\&\, \psi](\phi_1 \vee \phi_2)\urcorner\ \mathfrak{s}$. Second case: (R0) $\ulcorner\phi_2\urcorner\ \mathfrak{s}$, by (IHR) have $\ulcorner[x' = f \,\&\, \psi]\phi_2\urcorner\ \mathfrak{s}$ thus by monotonicity (Lemma 5.1) $\ulcorner[x' = f \,\&\, \psi](\phi_1 \vee \phi_2)\urcorner\ \mathfrak{s}$.

Case DC: Assume (0) $\Gamma \vdash [x' = f \,\&\, \psi]\rho$ and (1) $\Gamma \vdash [x' = f \,\&\, \psi \wedge \rho]\phi$ so by IH have (0A) $\ulcorner[x' = f \,\&\, \psi]\rho\urcorner\ \mathfrak{s}$ and (1A) $\ulcorner[x' = f \,\&\, \psi \wedge \rho]\phi\urcorner\ \mathfrak{s}$. To show the conclusion, first assume a duration d and solution sol such that (S) $(sol, \mathfrak{s}, d \vDash x' = f)$ and (C) $(\Pi t : [0, d].\ulcorner\psi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t)))$ then show (P) $\ulcorner\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t))))$. Because "solves" and universal quantification are prefix-closed, we also have: (S0) $(sol, \mathfrak{s}, r \vDash x' = f)$ for all $r \in [0, d]$ (C0) $(\Pi t : [0, r].\ulcorner\psi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t)))$ for all $r \in [0, d]$. Thus, apply (S0) and (C0) to (0) for each $r \in [0, d]$, getting (2)

$$\ulcorner\rho\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (x, x')\ (sol\ r, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ r)))) \text{ for all } r \in [0, d]$$

Since ρ appears in a domain constraint, we must have $x' \notin \mathrm{FV}(\rho)$ by syntactic restriction, so by Lemma 5.4 have (2A) $\ulcorner\rho\urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ r))$ for all $r \in [0, d]$. Conjoining (2A) with (C) we have (CA) $(\Pi t : [0, d].\ulcorner\psi \wedge \rho\urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t)))$. Applying (S) and (CA) to (1) we have (P) $\ulcorner\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (x, x')\ (sol\ r, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ r))))$ as desired.

Case DW: By assumption we have (0) $\Gamma\frac{y}{x}, \psi \vdash \phi$ for fresh y. so by IH have (0A) $\ulcorner\psi \to \phi\urcorner\ s$ for all s where $\ulcorner\Gamma\frac{y}{x}\urcorner(\mathfrak{s})$ is inhabited. By (A) and Lemma 5.2 we have $\ulcorner\Gamma\frac{y}{x}\urcorner(\mathfrak{s}\frac{y}{x})$. By Lemma 5.4 we have $\ulcorner\Gamma\frac{y}{x}\urcorner(\mathsf{set}\ (\mathfrak{s}\frac{y}{x})\ (x, x')\ (sol\ t, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t))))$ because the states only differ on x and x' which are not free variables of $\Gamma\frac{y}{x}$ by the freshness assumption on y. Thus, the IH gives (0B) $\ulcorner\psi \to \phi\urcorner\ (\mathsf{set}\ (\mathfrak{s}\frac{y}{x})\ (x, x')\ (sol\ t, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t))))$. Since y is fresh in $\psi \to \phi$, then we can apply Lemma 5.4 again. The application results in (0C) $\ulcorner\psi \to \phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (x, x')\ (sol\ t, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t))))$ as the states only differ on y and y'. To show $[x' = f \,\&\, \psi]\phi$ assume some $d > 0$ and $sol : [0, d] \Rightarrow \mathbb{R}$ such that (1) $(sol, \mathfrak{s}, d \vDash x' = f)$ and (2) $\ulcorner\psi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t))$ for $t \in [0, d]$ and then show $\ulcorner\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ d))$. For each t we can apply Lemma 5.4 in (2) because $x' \notin \mathrm{FV}(\psi)$ by syntactic restriction, giving (3) $\ulcorner\psi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (x, x')\ (sol\ t, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t))))$. Apply Lemma 5.4 on (0B) Apply (3) to (0C) to get $\ulcorner\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (x, x')\ (sol\ t, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t))))$ as desired.

Case DG: Assume side condition (Fresh) that $y \notin \mathrm{FV}(\Gamma) \cup \mathrm{FV}(f_0) \cup \mathrm{FV}(f) \cup \mathrm{FV}(a) \cup \mathrm{FV}(b) \cup \mathrm{FV}(\psi) \cup \{x\}$. Note (Cont) a, b are continuous because they are CdGL terms. In contrast to the corresponding dGL rule (Platzer, 2015a), y need not be fresh in ϕ because the postcondition of rule DG wraps the postcondition ϕ in existential quantification of y and y'. Assume (0) $\Gamma, y = f_0 \vdash [x'{=}f, y' = a(x)y + b(x) \,\&\, \psi]\phi$. From (A) and (Fresh) freshness of y in Γ we have (A1) $\ulcorner(\Gamma, y = f_0)\urcorner(\mathsf{set}\ \mathfrak{s}\ y\ (f\ \mathfrak{s}))$ so by (0) have (0A) $\ulcorner[x' = f, y' = a(x)y + b(x) \,\&\, \psi]\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ y\ (f\ \mathfrak{s}))$. Recall that the succedent formula we wish to show is $\ulcorner[\{x' = f \,\&\, \psi\}; \{y := *; y' := *\}^d]\phi\urcorner\ \mathfrak{s}$. First, assume d and sol are such that (1) $(sol, \mathfrak{s}, d \vDash x' = f)$ and (2) $(\Pi t : [0, d].\ulcorner\psi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t)))$ to show (P)

$$\ulcorner[\{y := *; y' := *\}^d]\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ d))))$$

We will prove the goal by applying (0A), which first requires constructing a solution $xysol$ to $x' = f, y' = a(x)y + b(x)$ of the same duration as sol. We construct $xysol$ according

to form $xysol(t) = (sol(t), ysol(t))$ for some $ysol(t)$ which solves $y'(t) = a(sol\ t)y + b(sol\ t)$. We choose initial value $v = (f\ \mathfrak{s})$, after which we construct $ysol$ as the solution of the initial value problem:

$$ysol(0) = v = f\ \mathfrak{s}$$
$$ysol'(t, v_y) = a(sol(t)) \cdot v_y + b(sol(t))$$

We show that $(ysol)'(t, v_y)$ is effectively Lipschitz continuous in v_y with some Lipschitz constant L. Specifically, let $L = \max_{t\in[0,d]}(a(sol(t)))$, where the maximum exists because it is the maximum of a continuous function on a compact interval. The function inside the maximum is continuous because composition preserves continuity, because a is continuous by (Cont), and because sol is continuous since it is the solution of an ODE.

Note the Lipschitz condition holds with constant L because for all $u, v \in \mathbb{R}$ have

$$\|(ysol)'(t, u) - (ysol)'(t, v)\| = \|u - v\|a(sol(t)) \leq \|u - v\| \max_{t\in[0,d]}(a(sol(t))$$

We now apply Theorem B.4 on time domain $X = [0, d]$, so that there constructively exists a solution $ysol$ on $[0, d]$ such that (solY) $(ysol, (\mathsf{set}\ \mathfrak{s}\ y\ v), d \vDash \{y' = a(sol\ t) \cdot y + b(sol\ t)\})$. From (solY) and (1) have (1XY) $(xysol, (\mathsf{set}\ \mathfrak{s}\ y\ v), d \vDash \{x' = f, y' = a(sol\ t) \cdot y + b(sol\ t)\})$. From (2) have (2XY) $(\Pi t : [0, d].\ulcorner\psi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (x, y)\ (xysol\ t)))$ by Lemma 5.4 using the assumption (Fresh) that y was fresh in ψ. Now we abbreviate $ss = (\mathsf{set}\ (\mathsf{set}\ \mathfrak{s}\ y\ v)\ (x, y)\ (xysol\ d)) = \mathsf{set}\ \mathfrak{s}\ (x, y)\ (xysol\ d)$. Apply $xysol, d$, (1XY), and (2XY) to (0A) to get $\ulcorner\phi\urcorner\ (\mathsf{set}\ ss\ (x', y'))\ (f\ ss, (a(x) \cdot y + b(x))\ ss)$, then (P) $\ulcorner[\{y := *; y' := *\}^d]\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ \mathfrak{s}\ x\ sol\ d)))$ by choosing $y = ysol\ d$ and $y' = (a(x) \cdot y + b(x))\ ss$ and because $\pi_0(xysol) = sol$. This is exactly what we wanted to show to complete the case.

Case DV: Assume side conditions (V1) t fresh and (V2) x, x', t, t' not free in d, ε. Note there is no additional side condition for existence of a solution to $\{t' = 1, x' = f\,\&\,\psi\}$ because (the IH applied to) (0) includes the existence of a solution to the ODE. Assume (0) $\Gamma \vdash \langle t := 0; \{t' = 1, x' = f\,\&\,\psi\}\rangle t \geq d$. Assume (1) $\Gamma \vdash [x' = f]((h)' - (g)') \geq \varepsilon$. Assume (2) $\Gamma \vdash d > 0 \wedge \varepsilon > 0 \wedge h - g \geq -d\varepsilon$. Assume (3) $\psi, h \geq g \vdash \phi$, then by IH have:

$$\text{(0A)}\ \ulcorner\langle t := 0; \{t' = 1, x' = f\,\&\,\psi\}\rangle t \geq d\urcorner\ \mathfrak{s}$$
$$\text{(1A)}\ \ulcorner[x' = f]((h)' - (g)') \geq \varepsilon\urcorner\ \mathfrak{s}$$
$$\text{(2A)}\ \ulcorner d > 0 \wedge \varepsilon > 0 \wedge h - g \geq -d\varepsilon\urcorner\ \mathfrak{s}$$
$$\text{(3A)}\ \ulcorner\psi\urcorner\ \mathfrak{s}, \ulcorner h \geq g\urcorner\ \mathfrak{s} \vdash \ulcorner\phi\urcorner\ \mathfrak{s}$$

To show (P) $\ulcorner\langle x' = f\,\&\,\psi\rangle\phi\urcorner\ \mathfrak{s}$ it suffices to choose sol and d such that

$$\text{(P0)}\ (sol, \mathfrak{s}, d \vDash x' = f)$$
$$\text{(P1)}\ (\Pi t : [0, d].\ulcorner\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t)))$$
$$\text{(P2)}\ \ulcorner\psi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (x, x')\ (sol\ d, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ d))))$$

From (0A) unpack $tsol$ and dur and get facts:

- (B) $(tsol, (\mathsf{set}\ \mathfrak{s}\ t\ 0), dur \vDash \{t' = 1, x' = f\})$,
- (C) $(\Pi tt : [0, dur].\ulcorner\psi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (t, x)\ (sol\ tt)))$, and
- (D) $\ulcorner t \geq d\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (t, x, t', x')\ ((tsol\ dur), (1, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ dur)))))$.

Since $(\lambda x.\ x)$ is the unique solution of $t' = 1$ we have $tsol(r) = (r, sol(r))$ for some solution sol. This also yields (Dur)

$$dur \geq d\ (\mathsf{set}\ \mathfrak{s}\ (t, x, t', x')\ ((tsol\ dur), (1, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ dur))))) = d\ s$$

by Lemma 5.4 and (V2). Fact (Dur) is a crucial fact which says semantic duration dur is at least the duration bound d at and after which the the postcondition of the variant argument will hold.

We project the solution for f; Note Lemma 5.4 applies since $t \notin \mathrm{FV}(f) \cup \mathrm{FV}(\psi) \cup \mathrm{FV}(\phi)$ by (V1). We get (B1) $(sol, \mathfrak{s}, dur \vDash x' = f)$ and (C1) $\Pi t : [0, dur].\ulcorner\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (x)\ (sol\ t))$ and (D1) $\ulcorner\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (x, x')\ (sol\ dur, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ dur))))$. Note that solution sol exists for the same duration dur for which $tsol$ exists because sol is merely the right component of $tsol$ and thus exists at least as long. Because sol has the same duration as $tsol$, fact (D) also means that sol exists for at least the desired duration d.

We can now instantiate (1A) with solution sol and its duration dur using (B1) and (CT1) to get (Der) $\ulcorner(h)' - (g)' \geq \varepsilon\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (x, x')\ (sol\ dur, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ dur))))$, which also holds for $t \in [0, dur]$ since solutions and quantifiers are prefix-closed. From (2A) have (Dpos) $d\,\mathfrak{s} > 0$ with d constant by (V2) and (Epos) $\varepsilon\,\mathfrak{s} > 0$ and ε constant by (V2) and (HG) $h\,\mathfrak{s} - g\,\mathfrak{s} \geq -d\varepsilon$ with d, ε constant. Fact (HG) crucially says $h\,\mathfrak{s} - g\,\mathfrak{s}$ changes quickly enough to achieve $h \geq g$ by time d. Let $HG(t) = (h - g)\ (\mathsf{set}\ \mathfrak{s}\ (x, x')\ (sol\ t, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t))))$ so that $HG'(t) = \ulcorner(h)' - (g)'\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (x, x')\ (sol\ dur, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ dur))))$. Recall that the value of ε is constant by (V2) and Assumption 1 (term coincidence), so we write $\hat{\varepsilon}$ for ε's constant value, i.e., we define $\hat{\varepsilon}$ so that $\hat{\varepsilon} = \varepsilon\ (\mathsf{set}\ \mathfrak{s}\ (x, x')\ (sol\ dur, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ dur)))) = \varepsilon\ (\mathsf{set}\ \mathfrak{s}\ (x, x')\ (sol\ t, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t)))) = \varepsilon\ \mathfrak{s}$ (for all $t \in [0, d]$).

Now Lemma B.6 applies to HG and c on $[0, dur]$ by (Der) so that (Progress) $HG(dur) - HG(0) \geq \hat{\varepsilon}\,dur > \hat{\varepsilon}\,d$, i.e., $HG(dur) \geq HG(0) + \hat{\varepsilon}\,dur > \hat{\varepsilon}\,d$.

By (HG) have $HG(0) \geq -d\,\hat{\varepsilon}$; add this inequality with the inequality of (Progress) to get (Big) $HG(dur) \geq 0$. From (Big) and instantiating (C1) at (Dur) which shows sufficiently long duration, we have the premises of (3A), so that (Post)

$$\ulcorner\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (x, x')\ (sol\ dur, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ dur))))$$

We conclude $\ulcorner\langle x' = f\,\&\,\psi\rangle\phi\urcorner\ \mathfrak{s}$ by choosing solution sol, duration dur, and noting (B1), (C1), and (Post).

Case bsolve: Assume the side condition (Fresh) that y is fresh. Assume the side condition (V1) that $x' \notin \mathrm{FV}(\phi)$. Assume the side condition (V2) that $\{t, t'\} \cap \mathrm{FV}(\Gamma) = \emptyset$. Assume the side condition (Sol) that the term sln is the unique, global solution of $x' = f$ such that $sln\ \mathfrak{s} = \mathfrak{s}\ x$ when $\mathfrak{s}\ t = 0$ for time variable t. Not every syntactically valid CdGL ODE has unique solutions, nor global solutions, let alone global solutions which lend themselves to tractable first-order proof obligations. In practical implementations, the latter requirement demands nilpotence so that solutions are first-order-expressible, which

trivially implies unique global solutions because nilpotent ODEs are linear and all linear equations not only have unique solutions, but global ones.

To show $[t:=0;x'=f\,\&\,\psi]\phi$, assume some d and sol such that (1) $(sol, (\mathsf{set}\ \mathfrak{s}\ t\ 0), d \vDash x'=f)$ and (2) $\Pi t:[0,d].\ulcorner\psi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (t,x)\ (t, sol\ t))$ so we can show (P)

$$\ulcorner\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (t,t',x,x')\ (d,1,sol\ d, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ d))))$$

By side condition (Sol), solutions are unique, so $sln\ u = sol\ (u\ t)$ for all states u. Then (2) and $\ulcorner\forall t:[0,d]\ [x:=sln]\psi\urcorner\ \mathfrak{s}$ are equivalent (3).

We now work on the premise in order to finish the proof of the conclusion. Assume the premise (0) $\Gamma\frac{y}{x}, t\geq 0, (\forall 0\leq s\leq t\,[t:=s;x:=sln]\psi), x=sln\frac{y}{x}, x'=f\vdash\phi$. By Lemma 5.2 on (A) have (A1) $\ulcorner\Gamma\frac{y}{x}\urcorner(\mathfrak{s}\frac{y}{x})$. Let $ss=\mathsf{set}\ (\mathfrak{s}\frac{y}{x})\ (t,t',x,x')\ (d,1,sol\ d, f\ (\mathsf{set}\ \mathfrak{s}\ (t,x)\ (d,sol\ d)))$ and show (A2) $\ulcorner(\Gamma\frac{y}{x}, t\geq 0, (\forall 0\leq s\leq t\,[t:=s;x:=sln]\psi), x=sln\frac{y}{x}, x'=f)\urcorner(ss)$ by showing each element of the context. The first element $\ulcorner\Gamma\frac{y}{x}\urcorner(ss)$ holds by (A1) and Lemma 5.4 using (V2) and (Fresh) to show that ss and $\mathfrak{s}\frac{y}{x}$ agree on $\{t,t',y,y'\}$. Formula $\ulcorner t\geq 0\urcorner\ ss$ holds by assumption on d. Formula $\ulcorner\forall 0\leq s\leq t\,[t:=s;x:=sln]\psi\urcorner\ ss$ holds from (2) by α-renaming t to s, by definition $ss\ t = d$, and by freshness condition (Fresh) of y with Lemma 5.4. Formula $\ulcorner x=sln\frac{y}{x}\urcorner\ ss$ holds since $sol\ d = sln\frac{y}{x}\ ss = sln\ (\mathsf{set}\ \mathfrak{s}\ t\ d) = ss\ x$. Formula $\ulcorner x'=f\urcorner\ ss$ holds since $ss\ x' = f\ (\mathsf{set}\ \mathfrak{s}\ (t,x)\ (d, sol\ d))$ by construction and Lemma 5.4 applies because y is fresh by assumption (Fresh) and x' cannot appear in f by side condition (V1).

Having proved (A2), we apply it to (0) and get (3) $\ulcorner\phi\urcorner\ ss$ as a result. By Lemma 5.4, the freshness (Fresh) of y, and the side condition (V1) on (3) and as a result we get $\ulcorner\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (t,t',x,x')\ (d,1,sln\ d, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sln\ d))))$, which is exactly the formula (P) we needed to prove.

Case dsolve: Assume the side condition (V1) that $x'\notin \mathrm{FV}(\phi)$ and also assume y is fresh as well as side condition (V2) $\{t,t'\}\cap\mathrm{FV}(\Gamma)=\emptyset$.. Assume the side condition (Sol) that term sln is a global solution of $x'=f$ such that $sln\ \mathfrak{s} = \mathfrak{s}\ x$ when $\mathfrak{s}\ t = 0$. In practice it is also the unique solution, but that is not strictly required in this rule. We assume a term (named d in the rule) expressing the duration of the ODE, which (as a side condition) does not depend on t, x, x'. Since term d is constant over time, we interchangeably write d for the value of the term, which is constant across all states discussed here.

To show the conclusion $\langle t:=0;x'=f\,\&\,\psi\rangle\phi$ we first choose some d and sol and then prove that (1) $(sol, (\mathsf{set}\ \mathfrak{s}\ t\ 0), d\vDash x'=f)$ and (2) $\Pi t:[0,d].\ulcorner\psi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (t,x)\ (t, sol\ t,))$ then show (3) $\ulcorner\phi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (t,t',x,x')\ (d,1,sol\ d, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ d))))$.

We choose sol such that $sln\ u = sol\ (u\ t)$ for all states u, i.e., sol be the same solution as sln, but as a term. We choose duration d to be the value of term d given in the proof rule application.

We now work on the premises in order to prove (1), (2), and (3). By (A) on the premise $\Gamma\vdash d\geq 0$ have (D) $(\ulcorner d\urcorner\ \mathfrak{s}) = 0$ as well as $\ulcorner d\urcorner\ u$ for all u that only disagree with $\mathfrak{s}$ on at most t,t',x,x',y,y' by Lemma 5.4. By Lemma 5.2 on (A) have (A1) $\ulcorner\Gamma\frac{y}{x}\urcorner(\mathfrak{s}\frac{y}{x})$. Let $ss=\mathsf{set}\ (\mathfrak{s}\frac{y}{x})\ (t,t',x,x')\ (d,1,sol\ d, f\ (\mathsf{set}\ \mathfrak{s}\ (t,x)\ (d,sol\ d)))$ and show (A2)

$$\ulcorner(\Gamma\tfrac{y}{x}, 0\leq t=d, x=sln\tfrac{y}{x}, x'=f)\urcorner(ss)$$

by showing each element of the context. The first component follows from (A1) by Lemma 5.4 using assumptions (Fresh) and (V2) to show agreement of ss with $\mathfrak{s}\frac{y}{x}$ on $\{t, t', y, y'\}$. The second formula $\ulcorner 0 \leq t = d \urcorner\ ss$ is by construction of ss and by (D). Formula $\ulcorner x = sln\frac{y}{x} \urcorner\ ss$ holds since $sol\ d = sln\frac{y}{x}\ ss = sln\ (\mathsf{set}\ \mathfrak{s}\ t\ d) = ss\ x$. Formula $\ulcorner x' = f \urcorner\ ss$ holds since $ss\ x' = f\ (\mathsf{set}\ \mathfrak{s}\ (t, x)\ (d, sol\ d))$ by construction and Lemma 5.4 applies because y is fresh by assumption and x' cannot appear in f by syntactic constraint. By applying (A2) on premise $(\Gamma\frac{y}{x}, 0 \leq t = d, x = sln\frac{y}{x}, x' = f) \vdash \phi$ we get (B1) $\ulcorner \phi \urcorner\ ss$. By the same argument as (A3), we have $\ulcorner (\Gamma\frac{y}{x}, 0 \leq t \leq d, x = sln\frac{y}{x}, x' = f) \urcorner (\mathsf{set}\ ss\ t\ tt)$ for all $0 \leq tt \leq d$. Applying (A3) to premise $\Gamma\frac{y}{x}, 0 \leq t \leq d, x = sln\frac{y}{x}, x' = f \vdash \psi$ we have (B2) $\ulcorner \psi \urcorner\ (\mathsf{set}\ ss\ t\ tt)$ for all $0 \leq tt \leq d$.

We can now prove (1), (2), and (3). Fact (1) is simply the side condition (Sol). Fact (2) follows from (B2) by Lemma 5.4 and freshness for y. Fact (3) follows from (B1) by Lemma 5.4 and freshness for y. Thus, the case is complete. □

Lemma 5.10 (Existence Property)**.** *Let $s : \mathfrak{S}$. If $\ulcorner \Gamma \urcorner(s) \vdash M : (\ulcorner \exists x\, \phi \urcorner\ s)$ then there constructively exists number $v : \mathbb{R}$ and CIC proof term N such that $\ulcorner \Gamma \urcorner(s) \vdash N : (\ulcorner \phi^v_x \urcorner\ s)$.*

Proof. Assume without loss of generality that the CIC proof term M is canonical. Note that $(\ulcorner \exists x\, \phi \urcorner\ s) \Leftrightarrow \Sigma v : \mathbb{R}. (\ulcorner \phi \urcorner\ (\mathsf{set}\ s\ x\ v))$ so by canonical forms, M is an instance of the Σ introduction rule. So by inversion on M the premise of the Σ introduction rule is satisfied, i.e., there constructively exist $v : \mathbb{R}$ and N such that $N : (\ulcorner \phi \urcorner\ (\mathsf{set}\ s\ x\ v))$ and such that the substitution ϕ^v_x is admissible. Then by Lemma 5.4 have $N^v_x : (\ulcorner \phi^v_x \urcorner\ s)$ as desired. □

Lemma 5.11 (Disjunction Property)**.** *If $\ulcorner \Gamma \urcorner(s) \vdash M : (\ulcorner \phi \vee \psi \urcorner\ s)$ then there constructively exists a proof term N such that $\ulcorner \Gamma \urcorner(s) \vdash N : (\ulcorner \phi \urcorner\ s)$ or $\ulcorner \Gamma \urcorner(s) \vdash N : (\ulcorner \psi \urcorner\ s)$.*

Proof. Recall that $(\ulcorner \Gamma \urcorner(s) \vdash M : (\ulcorner \phi \vee \psi \urcorner\ s)) \Leftrightarrow (\ulcorner \Gamma \urcorner(s) \vdash M : (\ulcorner \phi \urcorner\ s + \ulcorner \psi \urcorner\ s))$. Assume without loss of generality that M is canonical. So by canonical forms for coproducts in CIC, M is either a left injection or right injection. In each case, apply inversion so that the premise of the left (respectively, right) coproduct introduction rule is satisfied, i.e., either $(\ulcorner \Gamma \urcorner(s) \vdash N : (\ulcorner \phi \urcorner\ s))$ for some N or respectively $(\ulcorner \Gamma \urcorner(s) \vdash N : (\ulcorner \psi \urcorner\ s))$ for some N, as desired, completing each case and thus the proof. □

Appendix C

Appendix to Chapter 6

The appendix to Chapter 6 consists of proofs of general properties of the CdGL refinement calculus. Because Chapter 6 merely extends CdGL with refinement, we use the inductive proofs of properties such as the semantic lemmas and substitution from CdGL, but add new inductive cases for refinement. The soundness proof shows soundness of the new CdGL refinement rules; soundness of the non-refinement CdGL rules is assumed because it was proved in Chapter 5.

C.1 Theory Proofs

We first introduce preliminaries and semantic lemmas, then move on to substitution, soundness, and reification.

C.1.1 Notations and Preliminaries

We briefly review the CdGL static semantics and their extension to CdGL-with-refinement. The CdGL static semantics are discussed in greater detail in Appendix B.2.3. Recall (Fig. A.11 in Appendix A.4, Appendix B.2.3) that the syntactic free variables $\mathrm{FV}(e)$ of expression e are those which appear in free position, syntactic bound variables $\mathrm{BV}(\alpha)$ of a game are those which may be (syntactically) assigned during play, and syntactic must-bound variables $\mathrm{MBV}(\alpha)$ are those which are assigned on *every* execution path. Also recall (Appendix B.2.3) the semantic counterparts to the syntactic static semantics computations. The semantic free variables $\mathsf{FV}(e)$ of expression e are those which can influence the value of e; $\mathsf{FV}(e)$ is a non-strict subset of $\mathrm{FV}(e)$, often equal. The bound variables $\mathsf{BV}(\alpha)$ are the smallest set satisfying the bound-effect property for α, i.e., the complement of the largest set of variables whose values are preserved by playing α. The semantic must-bound variables $\mathsf{MBV}(\alpha)$ are those whose final values always agree when playing α twice from any two initial states which agree on $\mathsf{FV}(\alpha)$. Semantic must-bound variables are not used in practice; their definition was provided for the sake of completeness and is based on the coincidence lemma for games.

The CdGL refinement languages adds a single core connective to CdGL: Demonic refinement (from which Angelic refinement is also definable). Thus, our only addition to the semantic statics is a definition for the free variables of a refinement formula:

$$\mathrm{FV}(\alpha \leq_{[]} \beta) = \mathrm{FV}(\alpha) \cup \mathrm{FV}(\beta)$$

In particular, we will discuss the free variables of refinement formulas in greater detail in Appendix C.1.2.

As in Chapter 5, we use syntactic free and bound variables for CdGL formulas and games. Because the CdGL term language is defined as a semantic embedding, semantic free variables are used for terms, but the syntactic and semantic free variable notations $\mathrm{FV}(f)$ and $\mathsf{FV}(f)$ for terms may be used interchangeably, as we have *no* distinction between syntactic and semantic term free variable functions. We make the same assumptions on the static semantics of terms that are described in Appendix B.2.3.

Syntactic computations of the static semantics are emphasized because they have a clear computational interpretation, but semantic free variables play an important role in semantic proofs. Semantic free variables play a particularly important role in proofs about the refinement connective because the semantics of refinement quantifies over a CIC type family rather than a CdGL formula for its goal region; a semantic treatment of free variables is essential to reason about free variables of a semantic goal region which, in principle, need not be representable by any CdGL formula.

We notate vectors with arrows. For example, we write $\vec{x}$ for vectors of program variables in vectorial assignments.

The proofs in this appendix use a more concise notation for reification than the notation used in the main text of the chapter: rather than accepting a proof tree as an argument, we accept a proof term which is in one-to-one correspondence with the proof tree but consumes less space when written. We write $\alpha \,\mathsf{mod}\, M$ for the system that results from reifying proof term M in game α.

Because we present reification for proof terms, we also introduce a proof term notation for the differential equation rules of Chapter 5, whose proof terms are not discussed in the main text of the thesis. The differential equation proof rules are presented with their proof terms in Fig. C.1.

C.1.2 Properties of Refinement

While the semantics we gave for refinement are concise and general, their generality makes some proofs more difficult: the refinement semantics consider arbitrary postconditions, which might depend on arbitrary variables. However, the arbitrary-variable-dependency semantics is provably equivalent to a tighter semantics where postconditions only depend on bound variables of α or β, as we will show. Intuitively, in comparing two games (even if their free variables exceed their bound variables) the role of the postcondition is solely to capture the *output* behavior of the games. This tightening of the semantics is important because it allows us to use a tight definition of free variables for refinement formulas: $\mathrm{FV}(\alpha \leq_{[]} \beta) = \mathrm{FV}(\alpha) \cup \mathrm{FV}(\beta)$. Tight definitions of free variables are also important in turn

$$\text{(DC)}\quad \frac{\Gamma \vdash M : [x'{=}f \,\&\, \psi]\rho \quad \Gamma \vdash N : [x'{=}f \,\&\, \psi \wedge \rho]\phi}{\Gamma \vdash DC(M,N) : [x'{=}f \,\&\, \psi]\phi}$$

$$\text{(DG)}\quad \frac{\Gamma, y = f_0 \vdash M : [x'{=}f, y' = a(x)y + b(x) \,\&\, \psi]\phi}{\Gamma \vdash DG(y_0, a, b, M) : [x'{=}f \,\&\, \psi; \{y := *; y' := *\}^d]\phi}\ {}^{1}$$

$$\text{(DI)}\quad \frac{\Gamma \vdash M : \phi \quad \Gamma \vdash N : \forall x\, (\psi \to [x' := f](\phi)')}{\Gamma \vdash DI(M,N) : [x'{=}f \,\&\, \psi]\phi}$$

$$\text{(DW)}\quad \frac{\Gamma\tfrac{y}{x}, \psi \vdash M : \phi}{\Gamma \vdash DW(M) : [x'{=}f \,\&\, \psi]\phi}$$

$$\text{(bsolve)}\quad \frac{\Gamma\tfrac{y}{x}, t \geq 0, \hat{\psi}, x = sln\tfrac{y}{x}, x' = f \vdash M : \phi}{\Gamma \vdash DS(sln, M) : [t := 0; \{t' = 1, x' = f \,\&\, \psi\}]\phi}\ {}^{2}$$

$$\text{(dsolve)}\quad \frac{\begin{array}{c}\Gamma \vdash M : d \geq 0 \quad \vdash \psi \\ \Gamma\tfrac{y}{x}, 0 \leq t = d, x = sln\tfrac{y}{x}, x' = f \vdash N : \phi\end{array}}{\Gamma \vdash AS(d, sln, M, N) : \langle t := 0; \{t' = 1, x'{=}f \,\&\, \psi\}\rangle\phi}\ {}^{3}$$

[1] $y \notin \mathrm{FV}(\Gamma) \cup \mathrm{FV}(f_0) \cup \mathrm{FV}(f) \cup \mathrm{FV}(a) \cup \mathrm{FV}(b) \cup \mathrm{FV}(\psi) \cup \{x\}$
[2] y fresh, $x' \notin \mathrm{FV}(\phi)$, $\{t, t'\} \cap \mathrm{FV}(\Gamma) = \emptyset$, and sln solves $\{t' = 1, x' = f \,\&\, \psi\}$
[3] y fresh, $x' \notin \mathrm{FV}(\phi)$, $\{t, t', x, x'\} \cap \mathrm{FV}(d) = \emptyset$, $\{t, t\}' \cap \mathrm{FV}(\Gamma) = \emptyset$, and sln solves $\{t' = 1, x' = f \,\&\, \psi\}$

Figure C.1: CdGL proof proof-terms: ODEs.

because they ensure our refinement rules can be applied in the widest possible variety of cases.

Lemma C.1 (Domain restriction)**.** *The following semantics are equivalent to the semantics of $\alpha \leq^i_{\langle\rangle} \beta$ and $\alpha \leq^i_{[]} \beta$, respectively:*

- $\forall P : \mathcal{S} \Rightarrow \mathbb{T}_i\ (\mathsf{FV}(P) \subseteq \mathrm{BV}(\alpha) \cup \mathrm{BV}(\beta) \Rightarrow \langle\!\langle \alpha \rangle\!\rangle\ P\ s \Rightarrow \langle\!\langle \beta \rangle\!\rangle\ P\ s)$
- $\forall P : \mathcal{S} \Rightarrow \mathbb{T}_i\ (\mathsf{FV}(P) \subseteq \mathrm{BV}(\alpha) \cup \mathrm{BV}(\beta) \Rightarrow [[\alpha]]\ P\ s \Rightarrow [[\beta]]\ P\ s)$

That is, quantifying only over postconditions fully determined by the bound variables of α and β is sufficient to characterize the refinement relation between α and β.

Proof. We give the cases for $\alpha \leq^i_{[]} \beta$ and the cases for $\alpha \leq^i_{\langle\rangle} \beta$ are symmetric. We show that each version of the semantics implies the other.

The first direction is trivial. Assume $\ulcorner \alpha \leq^i_{[]} \beta \urcorner\ s$, so (0) $\forall P : \mathcal{S} \Rightarrow \mathbb{T}_i\ ([[\alpha]]\ P\ s \Rightarrow [[\beta]]\ P\ s)$. Now fix some $P \in (\mathcal{S} \Rightarrow \mathbb{T}_i)$ and assume $\mathsf{FV}(P) \subseteq \mathrm{BV}(\alpha) \cup \mathrm{BV}(\beta)$ and $[[\alpha]]\ P\ s$. Then by (0) have $[[\beta]]\ P\ s$. Thus, $\forall P : \mathcal{S} \Rightarrow \mathbb{T}_i\ (\mathsf{FV}(P) \subseteq \mathrm{BV}(\alpha) \cup \mathrm{BV}(\beta) \Rightarrow [[\alpha]]\ P\ s \Rightarrow [[\beta]]\ P\ s)$ as desired.

We show the converse direction. Assume (0) $\forall P : \mathcal{S} \Rightarrow \mathbb{T}_i\ (\mathsf{FV}(P) \subseteq \mathrm{BV}(\alpha) \cup \mathrm{BV}(\beta) \Rightarrow [[\alpha]]\ P\ s \Rightarrow [[\beta]]\ P\ s)$. Now fix $P \in (\mathcal{S} \Rightarrow \mathbb{T}_i)$ and assume (1) $[[\alpha]]\ P\ s$. Now define $\hat{P} = (\lambda r : \mathcal{S}.\ P\ (\mathsf{set}\ r\ \vec{x}\ (s\ \vec{x})))$ where $\vec{x} = \mathcal{V} \setminus (\mathrm{BV}(\alpha) \cup \mathrm{BV}(\beta))$, where $\mathcal{V}$ is the set of all variables. By construction (2) $\mathsf{FV}(\hat{P}) \subseteq \mathrm{BV}(\alpha) \cup \mathrm{BV}(\beta)$. Note that (3a) $(P\ r\star r \overset{\vec{x}}{=} s) \Rightarrow \hat{P}\ r$ holds for all r by arithmetic substitution and that (3b) $(\hat{P}\ r\star r \overset{\vec{x}}{=} s) \Rightarrow P\ r$ holds for all r by arithmetic substitution.

From (1) by Lemma 5.6 have $[[\alpha]]\ (\lambda r.\ t\ r\star r \overset{\vec{x}}{=} s)\ s$, then by monotonicity and (3a) have $[[\alpha]]\ \hat{t}\ s$. Combined with (2) this satisfies the assumption of (0) so $[[\beta]]\ \hat{t}\ s$, then by

Lemma 5.6 again $[[\beta]]$ $(\lambda r.\ \hat{t}\ r\star r \overset{\vec{x}}{=} s)$ s and by Lemma 5.1 and (3b) have $[[\beta]]$ t s. This held for all t so finally $\ulcorner \alpha \leq^i_{[]} \beta \urcorner\ s$ as desired. □

C.1.3 Substitution

The following lemmas about free variables, renaming, and substitution will be used in the soundness proof of the proof calculus. These lemmas were all proved inductively for CdGL (Chapter 5). For constructs introduced in Chapter 5, the semantics of Chapter 5 and Chapter 6 differ only by a universal quantification over type universes, so the proof cases from Chapter 5 transfer trivially. Thus, the proofs here merely add cases for refinement formulas within the inductive proofs of Chapter 5. We repeat the lemma statements verbatim from Chapter 5 to emphasize that we prove the same lemmas while only adding new cases. The only difference between the statements of the (coincidence, renaming, and substitution) lemmas in Chapter 5 vs. here is that our statements range over the language of CdGL with refinement rather than plain CdGL.

Lemma 5.4 (Formula Coincidence)**.** *Let $s,t : \mathfrak{S}$ and let $V \supseteq \mathrm{FV}(\phi)$ be a set of variables. Assume $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash s \overset{V}{=} t$, meaning s and t assign equal values to each variable in V. Assume $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash (\ulcorner\phi\urcorner\ s)$, then $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash (\ulcorner\phi\urcorner\ t)$.*

Coincidence for contexts also holds. Note that the states s and t at which we consider the conclusion context Γ_2 need not be the distinguished state $\mathfrak{s}$ of the context Γ_1. The claim can also be generalized to allow an arbitrary state argument for Γ_1 as a consequence by applying it again with $\Gamma_2 = \Gamma_1$.

If $\ulcorner\Gamma_1\urcorner(\mathfrak{s}) \vdash s \overset{V}{=} t$ for $V \supseteq \mathrm{FV}(\Gamma_2)$ then $\ulcorner\Gamma_1\urcorner(\mathfrak{s}) \vdash \ulcorner\Gamma_2\urcorner(s)$ is inhabited iff $\ulcorner\Gamma_1\urcorner(\mathfrak{s}) \vdash \ulcorner\Gamma_2\urcorner(t)$ is inhabited.

Coincidence for the construct $(sol, s, d \vDash x' = f)$ also holds: If $s \overset{\mathrm{FV}(f)\cup\{x\}}{=} t$ then $(sol, s, d \vDash x' = f) \Leftrightarrow (sol, t, d \vDash x' = f)$.

Proof. We prove the case for $\alpha \leq^i_{[]} \beta$, the case for $\alpha \leq^i_{\langle\rangle} \beta$ is symmetric, and the other cases are given in Chapter 5.

By Lemma C.1 have $\ulcorner \alpha \leq^i_{[]} \beta \urcorner\ s = (\forall P : \mathcal{S} \Rightarrow \mathbb{T}_i\ (\mathsf{FV}(P) \subseteq \mathrm{BV}(\alpha) \cup \mathrm{BV}(\beta) \Rightarrow [[\alpha]]\ P\ s \Rightarrow [[\beta]]\ P\ s))$ and likewise for state t, so it suffices to show that (P0) $\langle\!\langle\alpha\rangle\!\rangle\ P\ s = [[\alpha]]\ P\ t$ and (P1) $\langle\!\langle\beta\rangle\!\rangle\ P\ s = [[\beta]]\ P\ t$ assuming (0) $\mathsf{FV}(P) \subseteq \mathrm{BV}(\alpha) \cup \mathrm{BV}(\beta)$. From (0) have (0A) $\mathsf{FV}(P) \subseteq \mathrm{BV}(\alpha)$ and (0B) $\mathsf{FV}(P) \subseteq \mathrm{BV}(\beta)$.

Then the IH applies on α and β satisfying (P0) and (P1) as desired. □

Lemma 5.2 (Formula uniform renaming)**.** *If $\ulcorner\Gamma\urcorner(s) \vdash M : \ulcorner\phi\urcorner\ s$ then there exists some proof term (call it $M\frac{y}{x}$) such that $\ulcorner\Gamma\urcorner(s\frac{y}{x}) \vdash M\frac{y}{x} : \ulcorner\phi\frac{y}{x}\urcorner\ s\frac{y}{x}$. Also, $(sol, s, d \vDash z' = f) \Leftrightarrow (sol, s\frac{y}{x}, d \vDash (z\frac{y}{x})' = f\frac{y}{x})$. The function sol requires no renaming as it is just a univariate function from time to the value of a single scalar variable, as opposed to a function which accepts or returns a state.*

Proof. Recall that renaming x to y also renames x' to y'. It suffices to consider formulas ϕ without their surrounding context Γ because, as shown in the CdGL proofs

for Chapter 5, the contextual case reduces to the non-contextual case by a context IH ($\ulcorner\Gamma\urcorner(s) \Leftrightarrow \ulcorner\Gamma\frac{y}{x}\urcorner(s\frac{y}{x})$).

We give the case for $\alpha \leq^i_{[]} \beta$:

$$\begin{aligned}
&\ulcorner \alpha \leq^i_{[]} \beta \urcorner\ s\\
&= \forall P : \mathcal{S} \Rightarrow \mathbb{T}_i\ ([[\alpha]]\ P\ s \Rightarrow [[\beta]]\ P\ s)\\
&\underset{\text{IH}}{=} \forall P : \mathcal{S} \Rightarrow \mathbb{T}_i\ ([[\alpha\tfrac{y}{x}]]\ P\tfrac{y}{x}\ s\tfrac{y}{x} \Rightarrow [[\beta\tfrac{y}{x}]]\ P\tfrac{y}{x}\ s\tfrac{y}{x})\\
&=_{(*)} \forall P : \mathcal{S} \Rightarrow \mathbb{T}_i\ ([[\alpha\tfrac{y}{x}]]\ P\ s\tfrac{y}{x} \Rightarrow [[\beta\tfrac{y}{x}]]\ P\ s\tfrac{y}{x})\\
&= \ulcorner \{\alpha\tfrac{y}{x}\} \leq^i_{[]} \{\beta\tfrac{y}{x}\} \urcorner\ s\tfrac{y}{x}\\
&= \ulcorner (\alpha \leq^i_{[]} \beta)\tfrac{y}{x} \urcorner\ s\tfrac{y}{x}
\end{aligned}$$

where the step marked $(*)$ holds because the set of all transposition renamings $P\frac{y}{x}$ (for all P and fixed x, y) is identical to the set of all P. □

Lemma 5.7 (Formula substitution)**.** *If σ is admissible for Γ, then $\ulcorner\sigma(\Gamma)\urcorner(s) \Leftrightarrow \ulcorner\Gamma\urcorner(\sigma^*_s(s))$. If σ is additionally admissible for ϕ then $\ulcorner\Gamma\urcorner(\sigma^*_s(s)) \vdash M : \ulcorner\phi\urcorner\ \sigma^*_s(s)$ iff there exists a CIC proof term (call it $\sigma(M)$) such that $\ulcorner\sigma(\Gamma)\urcorner(s) \vdash \sigma(M) : \ulcorner\sigma(\phi)\urcorner\ s$. Likewise for predicate $(sol, s, d \vDash y' = g)$. In the converse direction, the witness is not necessarily M.*

Because the proof (in Appendix B.2) is constructive, it amounts to an algorithm for computing the substitution result $\sigma(M)$ for CIC proof terms M of the formula and context semantics. From here onward when M is a CIC proof term for the formula or context semantics, notation $\sigma(M)$ refers specifically to the proof term constructed in accordance to the proof.

Proof. We prove only the case for $\alpha \leq^i_{[]} \beta$, since all other cases are proven in Chapter 5.

We prove each direction separately. In each direction, the step marked "IH" refers to invocation of the IH on games; recall that formula substitution and game substitution are proven by simultaneous induction with one another. We are only showing the new case of this induction.

$$\begin{aligned}
&\ulcorner \sigma(\alpha \leq^i_{[]} \beta) \urcorner\ s\\
&\Rightarrow \ulcorner \sigma(\alpha) \leq^i_{[]} \sigma(\beta) \urcorner\ s\\
&\Rightarrow (\forall P : \mathcal{S} \Rightarrow \mathbb{T}_i\ ([[\sigma(\alpha)]]\ P\ s \Rightarrow [[\sigma(\beta)]]\ P\ s))\\
&\Rightarrow_* (\forall P : \mathcal{S} \Rightarrow \mathbb{T}_i\ ([[\sigma(\alpha)]]\ \sigma(P)\ s \Rightarrow [[\sigma(\beta)]]\ \sigma(P)\ s))\\
&\underset{\text{IH}}{\Rightarrow} (\forall P : \mathcal{S} \Rightarrow \mathbb{T}_i\ ([[\alpha]]\ P\ \sigma^*_s(s) \Rightarrow [[\beta]]\ P\ \sigma^*_s(s)))\\
&\Rightarrow \ulcorner \alpha \leq^i_{[]} \beta \urcorner\ \sigma^*_s(s)
\end{aligned}$$

The step marked $*$ holds because the universe $\mathcal{S} \Rightarrow \mathbb{T}_i$ is closed under substitution, so any universally quantified statement which holds of all P trivially holds of those P

which have form $\sigma(Q)$ for some Q. Here $\sigma(Q)$ refers to the *semantic* substitution $(\lambda s : \mathcal{S}.\ Q\ (\sigma^*_s(s)))$.

$$
\begin{aligned}
&\ulcorner \alpha \leq^i_{[]} \beta \urcorner\ \sigma^*_s(s)\\
\Rightarrow_{\text{Dom}}\ &(\forall P : \mathcal{S} \Rightarrow \mathbb{T}_i\ (\mathsf{FV}(P) \subseteq \text{BV}(\alpha) \cup \text{BV}(\beta) \Rightarrow [[\alpha]]\ P\ \sigma^*_s(s) \Rightarrow [[\beta]]\ P\ \sigma^*_s(s)))\\
\underset{\text{IH}}{\Rightarrow}\ &(\forall P : \mathcal{S} \Rightarrow \mathbb{T}_i\ (\mathsf{FV}(P) \subseteq \text{BV}(\alpha) \cup \text{BV}(\beta) \Rightarrow [[\sigma(\alpha)]]\ \sigma(P)\ s \Rightarrow [[\sigma(\beta)]]\ \sigma(P)\ s))\\
\Rightarrow_{\text{BV}}\ &(\forall P : \mathcal{S} \Rightarrow \mathbb{T}_i\ (\mathsf{FV}(P) \subseteq \text{BV}(\sigma(\alpha)) \cup \text{BV}(\sigma(\beta)) \Rightarrow [[\sigma(\alpha)]]\ \sigma(P)\ s \Rightarrow [[\sigma(\beta)]]\ \sigma(P)\ s))\\
\Rightarrow_{*}\ &(\forall P : \mathcal{S} \Rightarrow \mathbb{T}_i\ (\mathsf{FV}(P) \subseteq \text{BV}(\sigma(\alpha)) \cup \text{BV}(\sigma(\beta)) \Rightarrow [[\sigma(\alpha)]]\ P\ s \Rightarrow [[\sigma(\beta)]]\ P\ s))\\
\Rightarrow_{\text{Dom}}\ &\ulcorner \sigma(\alpha) \leq^i_{[]} \sigma(\beta) \urcorner\ s
\end{aligned}
$$

The steps marked Dom hold by Lemma C.1. The step marked BV observes that $\text{BV}(\alpha) = \text{BV}(\sigma(\alpha))$ and $\text{BV}(\beta) = \text{BV}(\sigma(\beta))$ because term substitutions do not introduce or remove bound variables. To show the step marked $*$, assume an arbitrary P such that $\mathsf{FV}(P) \subseteq \text{BV}(\sigma(\alpha))$. Here $\sigma(P)$ refers to the *semantic* substitution $(\lambda s : \mathcal{S}.\ P\ (\sigma^*_s(s)))$. Consider each $x \in Dom(\sigma)$. Since $x \notin \text{BV}(\alpha) \cup \text{BV}(\beta)$ by admissibility, then by the variable assumption on P we have that $x \notin \mathsf{FV}(P)$ for all such x, thus $s = \sigma^*_s(s)$ on $\mathsf{FV}(P)$ and thus by (semantic) Lemma 5.4 have $P\ s = (\sigma(P))\ s$ for all such P, which suffices to prove the step. □

C.1.4 Soundness

We now show the main soundness theorem using the previous lemmas. Our proof also appeals to proof rules from Chapter 5. We often wish to use those axioms on CIC regions $P : \mathfrak{S} \Rightarrow \mathbb{T}_i$ rather than CdGL formulas ϕ. Thus, we do not literally use each CdGL rule but use its generalization to regions. By inspection on the CdGL soundness proof (Theorem 5.9), the rules cited here generalize to regions by the compositional semantics of CdGL, i.e., by the fact that the CdGL game semantics are agnostic to whether their postcondition is the interpretation $\ulcorner \phi \urcorner$ of some formula ϕ or some other region.

Theorem 6.1 (Soundness of Proof Calculus)**.** *If sequent $\Gamma \vdash \phi$ is provable in CdGL with refinement then $\Gamma \vdash \phi$ is valid in CdGL with refinement. As a special case, if $(\cdot \vdash \phi)$ is provable, then ϕ is valid.*

Proof. By induction on the derivation. In each case, let $\mathfrak{s} : \mathfrak{S}$ be a designated CIC variable standing for the current state and assume (G) $\ulcorner \Gamma \urcorner(\mathfrak{s})$. In each case, fix $i \in \mathbb{N}$ and $P : \mathfrak{S} \Rightarrow \mathbb{T}_i$. In each case, we show some refinement of form $\alpha \leq_{[]} \beta$ by assuming $[[\alpha]]\ P\ \mathfrak{s}$ to show $[[\beta]]\ P\ \mathfrak{s}$. In every premise whose context is also Γ, assume modus ponens with assumption (G) has been applied.

We first mention several derived axioms which we use. The axioms K, Kd, and $[]\wedge$, while not sound for all games α, are sound for systems $\boldsymbol{\alpha}$.

(K) $[\boldsymbol{\alpha}](\phi \to \psi) \to ([\boldsymbol{\alpha}]\phi \to [\boldsymbol{\alpha}]\psi)$ [1]

(Kd) $[\boldsymbol{\alpha}](\phi \to \psi) \to (\langle\boldsymbol{\alpha}\rangle\phi \to \langle\boldsymbol{\alpha}\rangle\psi)$ [1]

([]∧) $[\boldsymbol{\alpha}]\phi \wedge [\boldsymbol{\alpha}]\psi \leftrightarrow [\boldsymbol{\alpha}](\phi \wedge \psi)$ [1]

[1] $\boldsymbol{\alpha}$ is a system

Soundness of K and Kd can be proved sound by an induction on $\boldsymbol{\alpha}$, then axiom []∧ derives from axiom K by duality. In classical logics, Kd is derivable from axiom K, but constructively the axioms are independent.

Case :=nop: We show $x := x \cong ?true$. We give the proof outline by equivalence reasoning first, then justify the numbered steps.

$$
\begin{aligned}
&(\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash [[x := x]]\ P\ \mathfrak{s})\\
\Leftrightarrow\ &(\ulcorner(\Gamma\tfrac{y}{x}, x = y)\urcorner(\mathfrak{s}) \vdash P\ \mathfrak{s})\\
\Leftrightarrow_{(1)}\ &(\ulcorner(\Gamma, (x = y)\tfrac{y}{x})\urcorner(\mathfrak{s}) \vdash P\tfrac{y}{x}\ \mathfrak{s})\\
\Leftrightarrow_{(2)}\ &(\ulcorner(\Gamma, (y = x))\urcorner(\mathfrak{s}) \vdash P\tfrac{y}{x}\ \mathfrak{s})\\
\Leftrightarrow_{(3)}\ &(\ulcorner(\Gamma, (x = y))\urcorner(\mathfrak{s}) \vdash P\tfrac{y}{x}\ \mathfrak{s})\\
\Leftrightarrow_{(4)}\ &(\ulcorner(\Gamma, (x = y))\urcorner(\mathfrak{s}) \vdash P\ \mathfrak{s}\tfrac{y}{x})\\
\Leftrightarrow_{(5)}\ &(\ulcorner(\Gamma, (x = y))\urcorner(\mathfrak{s}) \vdash P\ \mathfrak{s})\\
\Leftrightarrow_{(6)}\ &(\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash P\ \mathfrak{s})\\
\Leftrightarrow\ &(\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash \ulcorner true\urcorner\ \mathfrak{s} \Rightarrow P\ \mathfrak{s})\\
\Leftrightarrow\ &(\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash [[?true]]\ P\ \mathfrak{s})
\end{aligned}
$$

Step (1) is by Lemma 5.2. Step (2) is by simplifying the renaming. Step (3) is by symmetry of equality. Step (4) is by Lemma 5.2 and the fact $\ulcorner(\Gamma, (x = y))\urcorner(\mathfrak{s}) \Leftrightarrow \ulcorner(\Gamma, (x = y))\urcorner(\mathfrak{s}\frac{y}{x})$ which holds trivially from the equality $x = y$. Step (5) is because $\mathfrak{s} = \mathfrak{s}\frac{y}{x}$ under the assumption $x = y$. Step (6) holds because y was fresh.

Case ∪idem: We show $\alpha \cup \alpha \cong \alpha$ by showing $[[\alpha \cup \alpha]]\ P\ \mathfrak{s} \Leftrightarrow [[\alpha]]\ P\ \mathfrak{s}\star[[\alpha]]\ P\ \mathfrak{s} \Leftrightarrow [[\alpha]]\ P\ \mathfrak{s}$.

Case ∪c: We show $\alpha \cup \beta \cong \beta \cup \alpha$ by showing $[[\alpha \cup \beta]]\ P\ \mathfrak{s} \Leftrightarrow [[\alpha]]\ P\ \mathfrak{s}\star[[\beta]]\ P\ \mathfrak{s} \Leftrightarrow [[\beta]]\ P\ \mathfrak{s}\star[[\alpha]]\ P\ \mathfrak{s} \Leftrightarrow [[\beta \cup \alpha]]\ P\ \mathfrak{s}$.

Case ∪A: We show $\{\alpha \cup \beta\} \cup \gamma \cong \alpha \cup \{\beta \cup \gamma\}$ by showing

$$
\begin{aligned}
&[[\{\alpha \cup \beta\} \cup \gamma]]\ P\ \mathfrak{s} \Leftrightarrow [[\alpha \cup \beta]]\ P\ \mathfrak{s}\star[[\gamma]]\ P\ \mathfrak{s} \Leftrightarrow ([[\alpha]]\ P\ \mathfrak{s}\star[[\beta]]\ P\ \mathfrak{s})\star[[\gamma]]\ P\ \mathfrak{s}\\
\Leftrightarrow\ &[[\alpha]]\ P\ \mathfrak{s}\star([[\beta]]\ P\ \mathfrak{s}\star[[\gamma]]\ P\ \mathfrak{s}) \Leftrightarrow [[\alpha]]\ P\ \mathfrak{s}\star[[\beta \cup \gamma]]\ P\ \mathfrak{s} \Leftrightarrow [[\alpha \cup \{\beta \cup \gamma\}]]\ P\ \mathfrak{s}
\end{aligned}
$$

Case annih$_l$: In order to show $?false;\alpha \cong ?false$, we show a chain of equivalences: $[[?false;\alpha]]\ P\ \mathfrak{s} \Leftrightarrow [[?false]]([[\alpha]]\ P)\ \mathfrak{s} \Leftrightarrow (\ulcorner false\urcorner\ \mathfrak{s} \Rightarrow [[\alpha]]\ P\ \mathfrak{s}) \Leftrightarrow \ulcorner true\urcorner\ \mathfrak{s} \Leftrightarrow (\ulcorner false\urcorner\ \mathfrak{s} \Rightarrow P\ \mathfrak{s}) \Leftrightarrow [[?false]]\ P\ \mathfrak{s}$.

Case DDE: We show $\{\alpha^d\}^d \cong \alpha$ by showing $[[\{\alpha^d\}^d]]\ P\ \mathfrak{s} \Leftrightarrow \langle\langle\alpha^d\rangle\rangle\ P\ \mathfrak{s} \Leftrightarrow [[\alpha]]\ P\ \mathfrak{s}$.

Case R$\langle\cdot\rangle$: Assume $\mathfrak{R}(\phi) \leq i$ by side condition. Assume (D1) $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash \langle\!\langle\alpha\rangle\!\rangle \ulcorner\phi\urcorner\ \mathfrak{s}$ and (D2) $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash \ulcorner\alpha \leq^i_{\langle\rangle} \beta\urcorner\ \mathfrak{s}$, that is, (D2) $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash \ulcorner\alpha^d \leq^i_{[]} \beta^d\urcorner\ \mathfrak{s}$, that is, so that from (D2) we have $\forall P : (\mathfrak{S} \Rightarrow \mathbb{T}_i)\ ([[\alpha^d]]\ P\ \mathfrak{s} \Rightarrow [[\beta^d]]\ P\ \mathfrak{s})$ and then $\forall P : (\mathfrak{S} \Rightarrow \mathbb{T}_i)\ (\langle\!\langle\alpha\rangle\!\rangle\ P\ \mathfrak{s} \Rightarrow \langle\!\langle\beta\rangle\!\rangle\ P\ \mathfrak{s})$ by semantics. Since $\mathfrak{R}(\phi) \leq i$ then $\ulcorner\phi\urcorner\ s \in \mathbb{T}_i$, so by specialization and modus ponens on (D1) have $\langle\!\langle\beta\rangle\!\rangle \ulcorner\phi\urcorner\ \mathfrak{s}$ as desired.

Case R$[\cdot]$: Assume $\mathfrak{R}(\phi) \leq i$ by side condition. Assume (D1) $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash [[\alpha]] \ulcorner\phi\urcorner\ \mathfrak{s}$ and (D2) $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash \ulcorner\alpha \leq^i_{[]} \beta\urcorner\ \mathfrak{s}$ and from (D2) have $\forall P : (\mathfrak{S} \Rightarrow \mathbb{T}_i)\ ([[\alpha]]\ P\ \mathfrak{s} \Rightarrow [[\beta]]\ P\ \mathfrak{s})$. Since $\mathfrak{R}(\phi) \leq i$ then $\ulcorner\phi\urcorner\ s \in \mathbb{T}_i$, so by specialization and modus ponens on (D1) have $[[\beta]] \ulcorner\phi\urcorner\ \mathfrak{s}$ as desired.

Case $\langle?\rangle$: Assume (D) $\ulcorner\phi \to \psi\urcorner\ \mathfrak{s}$. Reason by chain of $\Leftrightarrow$ and $\Rightarrow$: $\langle?\phi\rangle\ P\ \mathfrak{s} \Leftrightarrow (\ulcorner\phi\urcorner\ \mathfrak{s}\star P\ \mathfrak{s}) \Rightarrow_{(\mathrm{D})} (\ulcorner\psi\urcorner\ \mathfrak{s}\star P\ \mathfrak{s}) \Leftrightarrow (\langle\!\langle?\psi\rangle\!\rangle\ P\ \mathfrak{s})$ so $\langle\!\langle?\phi\rangle\!\rangle\ P\ \mathfrak{s} \Rightarrow \langle\!\langle?\psi\rangle\!\rangle\ P\ \mathfrak{s}$ as desired.

Case $[?]$: Assume (D) $\ulcorner\psi \to \phi\urcorner\ \mathfrak{s}$. Reason by chain of $\Leftrightarrow$ and $\Rightarrow$: $[[?\phi]]\ P\ \mathfrak{s} \Leftrightarrow (\ulcorner\phi\urcorner\ \mathfrak{s} \Rightarrow P\ \mathfrak{s}) \Rightarrow_{(\mathrm{D})} (\ulcorner\psi\urcorner\ \mathfrak{s} \Rightarrow P\ \mathfrak{s}) \Leftrightarrow [[?\psi]]\ P\ \mathfrak{s}$ so that $[[?\phi]]\ P\ \mathfrak{s} \Rightarrow [[?\psi]]\ P\ \mathfrak{s}$ as desired.

Case un$*$: Assume as a side condition that (SC1) $\boldsymbol{\alpha}$ is a system. Let i be the rank of the refinement, which as a side condition (SC2) is at least the rank of $\boldsymbol{\alpha}$ and β. Assume (D) $[[\boldsymbol{\alpha}^*]] \ulcorner(\boldsymbol{\alpha} \leq^i_{[]} \beta)\urcorner\ \mathfrak{s}$. Assume (A) $[[\boldsymbol{\alpha}^*]]\ P\ \mathfrak{s}$ to show $[[\beta^*]]\ P\ \mathfrak{s}$.

By inversion on (D) and (A) there exist $J, K : \mathfrak{S} \Rightarrow \mathbb{T}_i$ such that (D1) $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash J\ \mathfrak{s}$ (D2) $\mathfrak{s} : \mathfrak{S}, J\ \mathfrak{s} \vdash [[\boldsymbol{\alpha}]]\ J\ \mathfrak{s}$, and (D3) $\mathfrak{s} : \mathfrak{S}, J\ \mathfrak{s} \vdash \ulcorner\boldsymbol{\alpha} \leq^i_{[]} \beta\urcorner\ \mathfrak{s}$. (A1) $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash K\ \mathfrak{s}$ (A2) $\mathfrak{s} : \mathfrak{S}, K\ \mathfrak{s} \vdash [[\boldsymbol{\alpha}]]\ K\ \mathfrak{s}$, and (A3) $\mathfrak{s} : \mathfrak{S}, K\ \mathfrak{s} \vdash P\ \mathfrak{s}$. We show $[[\beta^*]]\ P\ \mathfrak{s}$ by coinduction with invariant[1] $(\lambda t : \mathfrak{S}.\ J\ t\star K\ t)$. The first premise $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash J\ \mathfrak{s}\star K\ \mathfrak{s}$ is immediate by (D1) and (A1). The third premise $\mathfrak{s} : \mathfrak{S}, (J\ \mathfrak{s}\star K\ \mathfrak{s}) \vdash P\ \mathfrak{s}$ is immediate by (A3) for all $t : \mathfrak{S}$. The second premise is $\mathfrak{s} : \mathfrak{S}, (J\ \mathfrak{s}\star K\ \mathfrak{s}) \vdash [[\beta]]\ (\lambda s.\ J\ s\star K\ s)\ \mathfrak{s}$. By (D3) we have $\mathfrak{s} : \mathfrak{S}, (J\ \mathfrak{s}\star K\ \mathfrak{s}) \vdash \ulcorner\boldsymbol{\alpha} \leq^i_{[]} \beta\urcorner\ \mathfrak{s}$ so we apply R$[\cdot]$, which is applicable because $(\lambda s.\ J\ s\star K\ s) : \mathfrak{S} \Rightarrow \mathbb{T}_i$ and by (SC2). Then by soundness of R$[\cdot]$ it suffices to show $\mathfrak{s} : \mathfrak{S}, J\ \mathfrak{s}\star K\ \mathfrak{s} \vdash [[\boldsymbol{\alpha}]]\ (\lambda t.\ J\ t\star K\ t)\ \mathfrak{s}$. By (SC1), $\boldsymbol{\alpha}$ is a system so axiom K applies, thus $\mathfrak{s} : \mathfrak{S}, (J\ \mathfrak{s}\star K\ \mathfrak{s}) \vdash [[\boldsymbol{\alpha}]]\ (\lambda t.\ J\ t\star K\ t)\ \mathfrak{s}$ holds by (D2) and (A2).

Case roll$_l$: To show $?true \cup \{\alpha; \alpha^*\} \cong \alpha^*$ we show each direction of the equivalence.

We show the left-to-right case: Assume (A) $[[?true \cup \{\alpha; \alpha^*\}]]\ P\ \mathfrak{s}$ to show $[[\alpha^*]]\ P\ \mathfrak{s}$. From rules $[\cup]$E1 and $[\cup]$E2 on (A) have (A1) $[[?true]]\ P\ \mathfrak{s}$ and (A2) $[[\alpha; \alpha^*]]\ P\ \mathfrak{s}$. Respectively, from (A1) have (A3) $P\ \mathfrak{s}$ by rule $[?]$I and modus ponens on verum *true*, then from (A2) have (A4) $[[\alpha]]\ ([[\alpha^*]]\ P)\ \mathfrak{s}$ by the semantics of sequential composition. From (A2) and (A4) by rule $[*]$R have $[[\alpha^*]]\ P\ \mathfrak{s}$ as desired.

We show the right-to-left case: Assume (A) $[[\alpha^*]]\ P\ \mathfrak{s}$ to show $[[?true \cup \alpha; \alpha^*]]\ P\ \mathfrak{s}$. By rule $[*]$E on (A) have (A1) $P\ \mathfrak{s}$ and (A2) $[[\alpha]]([[\alpha^*]]\ P)\ \mathfrak{s}$. Then respectively have (A3) $[[?true]]\ P\ \mathfrak{s}$ by weakening and $[?]$I on (A1), and (A4) $[[\alpha; \alpha^*]]\ P\ \mathfrak{s}$ by rule $\langle\!\langle;\rangle\!\rangle$I on (A2). By rule $[\cup]$I on (A3) and (A4) have $[[?true \cup \alpha; \alpha^*]]\ P\ \mathfrak{s}$ as desired.

Case $\langle:*\rangle$: Have $\langle\!\langle x := f\rangle\!\rangle\ P\ \mathfrak{s} \Rightarrow (P\ (\mathsf{set}\ \mathfrak{s}\ x\ (f\ \mathfrak{s}))) \Rightarrow (\Sigma v : \mathbb{R}.\, P\ (\mathsf{set}\ \mathfrak{s}\ x\ v)) \Rightarrow \langle\!\langle x := *\rangle\!\rangle\ P\ \mathfrak{s}$.

Case $[:*]$: Have $[[x := *]]\ P\ \mathfrak{s} \Rightarrow (\Pi v : \mathbb{R}.\, P\ (\mathsf{set}\ \mathfrak{s}\ x\ v)) \Rightarrow (P\ (\mathsf{set}\ \mathfrak{s}\ x\ (f\ s))) \Rightarrow [[x := f]]\ P\ \mathfrak{s}$.

Case ;S: Assume a side condition (SC) that $\boldsymbol{\alpha}_1$ is a system. Assume (D1) $\ulcorner\boldsymbol{\alpha}_1 \leq_{[]} \alpha_2\urcorner\ \mathfrak{s}$

[1]Our argument's structure is like for $[*]$I, but it uses arbitrary regions, not formulas, as invariants.

and (D2) $[[\boldsymbol{\alpha}_1]]$ ($\ulcorner \beta_1 \leq_{[]} \beta_2 \urcorner$) $\mathfrak{s}$ and (A) $[[\boldsymbol{\alpha}_1; \beta_1]]$ P $\mathfrak{s}$ to show $[[\alpha_2; \beta_2]]$ P $\mathfrak{s}$. From (A) by the semantics of sequential composition have (1) $[[\boldsymbol{\alpha}_1]]$ ($[[/\beta_1]]$ P) $\mathfrak{s}$. Axiom K is applicable to (D2) and (1) because of (SC), yielding (2) $[[\boldsymbol{\alpha}_1]]$ ($\lambda t.\ \ulcorner \beta_1 \leq_{[]} \beta_2 \urcorner\ t\star[[\beta_1]]\ P\ t$) $\mathfrak{s}$. Then by monotonicity rule M and rule R[·] have (3) $[[\boldsymbol{\alpha}_1]]$ ($[[\beta_2]]$ P) $\mathfrak{s}$. By rule R[·] on the postcondition ($\lambda t.\ \ulcorner \beta_1 \leq_{[]} \beta_2 \urcorner\ t\star[[\beta_1]]\ P\ t$) from (3) and (D1) have (4) $[[\alpha_2]]$ ($[[\beta_2]]$ P), so by rule $\langle\!\![;]\!\!\rangle$I have $[[\alpha_2; \beta_2]]$ P $\mathfrak{s}$ as desired.

Case ;G: Assume (D1) $\ulcorner \alpha_1 \leq_{[]} \alpha_2 \urcorner$ $\mathfrak{s}$ and (D2) $\ulcorner \cdot \urcorner(\mathfrak{s}) \vdash \ulcorner \beta_1 \leq_{[]} \beta_2 \urcorner$ $\mathfrak{s}$. Then from (A) have $[[\alpha_1]]$ ($[[\beta_1]]$ P) $\mathfrak{s}$. Apply R[·] with (D1) to get (1) $[[\alpha_2]]$ ($[[\beta_1]]$ P) $\mathfrak{s}$. Since (D2) is valid then plug in P to get that (2) $\mathfrak{s} : \mathfrak{S}, [[\beta_1]]\ P\ \mathfrak{s} \vdash [[\beta_2]]\ P\ \mathfrak{s}$. Then monotonicity rule M on (1) and (2) gives $[[\alpha_2]]([[\beta_2]]\ P)\ \mathfrak{s}$ which immediately gives $[[\alpha_2; \beta_2]]$ P $\mathfrak{s}$ by rule $\langle\!\![;]\!\!\rangle$I as desired.

Case [∪]R: Assume (D1) $\ulcorner \alpha \leq_{[]} \beta \urcorner$ $\mathfrak{s}$ and (D2) $\ulcorner \alpha \leq_{[]} \gamma \urcorner$ $\mathfrak{s}$ and (A) $[[\alpha]]$ P $\mathfrak{s}$ so by rule R[·] on (D1) and (D2) have $[[\beta]]$ P $\mathfrak{s}$ and $[[\gamma]]$ P $\mathfrak{s}$ so that $[[\beta \cup \gamma]]$ P $\mathfrak{s}$ as desired, so that $\ulcorner \alpha \leq_{[]} \beta \cup \gamma \urcorner$ $\mathfrak{s}$.

Case trans: Assume (D1) $\ulcorner \Gamma \urcorner(\mathfrak{s}) \vdash \ulcorner \alpha \leq_{[]} \beta \urcorner$ $\mathfrak{s}$ and (D2) $\ulcorner \Gamma \urcorner(\mathfrak{s}) \vdash \ulcorner \beta \leq_{[]} \gamma \urcorner$ $\mathfrak{s}$. Want to show $\ulcorner \Gamma \urcorner(\mathfrak{s}) \vdash \ulcorner \alpha \leq_{[]} \gamma \urcorner$ $\mathfrak{s}$. (A) $[[\alpha]]$ P $\mathfrak{s}$. From (D1) and (D2) have $[[\alpha]]$ P $\mathfrak{s}$ $\Rightarrow$ $[[\beta]]$ P $\mathfrak{s}$ and $[[\beta]]$ P $\mathfrak{s}$ $\Rightarrow$ $[[\gamma]]$ P $\mathfrak{s}$ so by modus ponens twice from (A) have $[[\gamma]]$ P $\mathfrak{s}$, so finally $[[\alpha]]$ P $\mathfrak{s}$ $\Rightarrow$ $[[\gamma]]$ P $\mathfrak{s}$, i.e., $\ulcorner \alpha \leq_{[]} \gamma \urcorner$ $\mathfrak{s}$.

Case refl: Want to show $\ulcorner \Gamma \urcorner(\mathfrak{s}) \vdash \ulcorner \alpha \leq_{[]} \alpha \urcorner$ $\mathfrak{s}$. Assume (A) $[[\alpha]]$ P $\mathfrak{s}$ so $[[\alpha]]$ P $\mathfrak{s}$ by rule hyp, thus $[[\alpha]]$ P $\mathfrak{s}$ $\Rightarrow$ $[[\alpha]]$ P $\mathfrak{s}$, i.e., $\ulcorner \alpha \leq_{[]} \alpha \urcorner$ $\mathfrak{s}$.

Case ;A: Each step is reversible so that this case is an equivalence $\{\alpha; \beta\}; \gamma \cong \alpha; \beta; \gamma$. $[[\{\alpha; \beta\}; \gamma]]$ P $\mathfrak{s}$ $\Leftrightarrow$ $[[\alpha; \beta]]$ ($[[\gamma]]$ P) $\mathfrak{s}$ $\Leftrightarrow$ $[[\alpha]]$ ($[[\beta]]$ ($[[\gamma]]$ P)) $\mathfrak{s}$ $\Leftrightarrow_*$ $[[\alpha]]$ ($[[\beta; \gamma]]$ P) $\mathfrak{s}$ $\Leftrightarrow$ $[[\alpha; \{\beta; \gamma\}]]$ P $\mathfrak{s}$ where step (*) also uses monotonicity rule M and every step uses rule $\langle\!\![;]\!\!\rangle$I or eliminates ; by appeal to the semantics of sequential composition.

Case :=:=: Assume side condition $x \notin \mathrm{FV}(g)$. Then note $(\mathsf{set}\ \mathfrak{s}\ x\ (f\ \mathfrak{s})) = \mathfrak{s}$ on $\mathrm{FV}(g)^{\complement}$ by Lemma 5.6, then by Assumption 1 have (1) $g\ (\mathsf{set}\ \mathfrak{s}\ x\ (f\ \mathfrak{s})) = g\ \mathfrak{s}$. Then:

$$
\begin{aligned}
& [[x := f; x := g]]\ P\ \mathfrak{s} \\
\Leftrightarrow\ & [[x := f]]\ ([[x := g]]\ P)\ \mathfrak{s} \\
\Leftrightarrow\ & [[x := g]]\ P\ (\mathsf{set}\ \mathfrak{s}\ x\ (f\ \mathfrak{s})) \\
\Leftrightarrow\ & P\ (\mathsf{set}\ (\mathsf{set}\ \mathfrak{s}\ x\ (f\ \mathfrak{s}))\ x\ (g\ (\mathsf{set}\ \mathfrak{s}\ x\ (f\ \mathfrak{s})))) \\
\Leftrightarrow_{(1)}\ & P\ (\mathsf{set}\ (\mathsf{set}\ \mathfrak{s}\ x\ (f\ \mathfrak{s}))\ x\ (g\ \mathfrak{s})) \\
\Leftrightarrow\ & P\ (\mathsf{set}\ \mathfrak{s}\ x\ (g\ \mathfrak{s})) \\
\Leftrightarrow\ & [[x := g]]\ P\ \mathfrak{s}
\end{aligned}
$$

Case ;d_r: By a chain of equalities, $[[\{\alpha \cup \beta\}; \gamma]]$ P $\mathfrak{s}$ $\Leftrightarrow$ $[[\alpha \cup \beta]]$ ($[[\gamma]]$ P) $\mathfrak{s}$ $\Leftrightarrow$ $[[\alpha]]$ ($[[\gamma]]$ P) $\mathfrak{s}\star[[\beta]]$ ($[[\gamma]]$ P) $\mathfrak{s}$ $\Leftrightarrow$ $[[\alpha; \gamma]]$ P $\mathfrak{s}\star[[\beta; \gamma]]$ P $\mathfrak{s}$ $\Leftrightarrow$ $[[\{\alpha; \gamma\} \cup \{\beta; \gamma\}]]$ P $\mathfrak{s}$.

Case [∪]L1: Have $[[\alpha \cup \beta]]$ P $\mathfrak{s}$ $\Rightarrow$ $[[\alpha]]$ P $\mathfrak{s}\star[[\beta]]$ P $\mathfrak{s}$ $\Rightarrow$ $[[\alpha]]$ P $\mathfrak{s}$.

Case [∪]L2: Have $[[\alpha \cup \beta]]$ P $\mathfrak{s}$ $\Rightarrow$ $[[\alpha]]$ P $\mathfrak{s}\star[[\beta]]$ P $\mathfrak{s}$ $\Rightarrow$ $[[\beta]]$ P $\mathfrak{s}$.

Case ⟨∪⟩R1: Have $[[\alpha^d]]$ P $\mathfrak{s}$ $\Rightarrow$ $\langle\!\langle \alpha \rangle\!\rangle$ P $\mathfrak{s}$ $\Rightarrow$ $\langle\!\langle \alpha \cup \beta \rangle\!\rangle$ P $\mathfrak{s}$ $\Rightarrow$ $[[\{\alpha \cup \beta\}^d]]$ P $\mathfrak{s}$.

Case ⟨∪⟩R2: Have $[[\beta^d]]$ P $\mathfrak{s}$ $\Rightarrow$ $\langle\!\langle \beta \rangle\!\rangle$ P $\mathfrak{s}$ $\Rightarrow$ $\langle\!\langle \alpha \cup \beta \rangle\!\rangle$ P $\mathfrak{s}$ $\Rightarrow$ $[[\{\alpha \cup \beta\}^d]]$ P $\mathfrak{s}$.

Case ;id_l: Have $[[?\mathit{true}; \alpha]]$ P $\mathfrak{s}$ $\Leftrightarrow$ $[[?\mathit{true}]]$ ($[[\alpha]]$ P $\mathfrak{s}$) $\Leftrightarrow$ ($\ulcorner \mathit{true} \urcorner$ $\mathfrak{s}$ $\Rightarrow$ $[[\alpha]]$ P $\mathfrak{s}$) $\Leftrightarrow$ $[[\alpha]]$ P $\mathfrak{s}$.

Case skip^d: Have $[[\mathsf{skip}]]\ P\ \mathfrak{s} \Leftrightarrow [[?true]]\ P\ \mathfrak{s} \Leftrightarrow (\ulcorner true \urcorner\ \mathfrak{s} \Rightarrow P\ \mathfrak{s}) \Leftrightarrow P\ \mathfrak{s} \Leftrightarrow \ulcorner true \urcorner\ \mathfrak{s} \star P\ \mathfrak{s} \Leftrightarrow \langle\!\langle ?true \rangle\!\rangle\ P\ \mathfrak{s} \Leftrightarrow [[?true^d]]\ P\ \mathfrak{s} \Leftrightarrow [[\mathsf{skip}^d]]\ P\ \mathfrak{s}$.

Case $;^d$: Have $[[\{\alpha;\beta\}^d]]\ P\ \mathfrak{s} \Leftrightarrow \langle\!\langle \alpha;\beta \rangle\!\rangle\ P\ \mathfrak{s} \Leftrightarrow \langle\!\langle \alpha \rangle\!\rangle\ (\langle\!\langle \beta \rangle\!\rangle\ P)\ \mathfrak{s} \Leftrightarrow \langle\!\langle \alpha \rangle\!\rangle\ ([[\beta^d]]\ P)\ \mathfrak{s} \Leftrightarrow [[\alpha^d]]\ ([[\beta^d]]\ P)\ \mathfrak{s} \Leftrightarrow [[\alpha^d;\beta^d]]\ P\ \mathfrak{s}$.

Case $:=^d$: Have $[[x := f^d]]\ P\ \mathfrak{s} \Leftrightarrow \langle\!\langle x := f \rangle\!\rangle\ P\ \mathfrak{s} \Leftrightarrow P\ (\mathsf{set}\ \mathfrak{s}\ x\ (f\ \mathfrak{s})) \Leftrightarrow [[x := f]]\ P\ \mathfrak{s}$.

The following cases address proof rules for ODEs. Note that these rules use t as a program variable name. This usage should not be confused with the use of t as a state metavariable name.

Case DC: Assume (D) the premise $[[x' = f\,\&\,\phi]]\ \ulcorner \psi \urcorner\ \mathfrak{s}$ holds. Now we wish to show the equivalence $[[x' = f\,\&\,\phi]]\ P\ \mathfrak{s} \Leftrightarrow [[x' = f\,\&\,\phi \wedge \psi]]\ P\ \mathfrak{s}$. Show the forward implication, then converse implication.

Forward implication: Assume (A)

$$
\begin{array}{l}
[[x' = f\,\&\,\phi]]\ P\ \mathfrak{s} \Leftrightarrow \\
\quad \Pi d : \mathbb{R}_{\geq 0}.\, \Pi sol : [0,d] \Rightarrow \mathbb{R}. \\
\qquad (sol, \mathfrak{s}, d \vDash x' = f) \\
\quad \Rightarrow (\Pi t : [0,d].\ulcorner \phi \urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t))) \\
\quad \Rightarrow P\ (\mathsf{set}\ \mathfrak{s}\ (x,x')\ (sol\ d, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ d))))
\end{array}
$$

Want to show

$$
\begin{array}{l}
[[x' = f\,\&\,\phi \wedge \psi]]\ P\ \mathfrak{s} \Leftrightarrow \\
\quad \Pi d : \mathbb{R}_{\geq 0}.\, \Pi sol : [0,d] \Rightarrow \mathbb{R}. \\
\qquad (sol, \mathfrak{s}, d \vDash x' = f) \\
\quad \Rightarrow (\Pi t : [0,d].\ulcorner \phi \wedge \psi \urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t))) \\
\quad \Rightarrow P\ (\mathsf{set}\ \mathfrak{s}\ (x,x')\ (sol\ d, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ d))))
\end{array}
$$

So assume (B1) $(sol, \mathfrak{s}, d \vDash x' = f)$ and (B2) $(\Pi t : [0,d].\ulcorner \phi \wedge \psi \urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t)))$ then by left projection have (B3) $(\Pi t : [0,d].\ulcorner \phi \urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t)))$. By applying (B1) and (B3) to (A) have $P\ (\mathsf{set}\ \mathfrak{s}\ (x,x')\ (sol\ d, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ d))))$ as desired.

Converse implication: Assume (A)

$$
\begin{array}{l}
[[x' = f\,\&\,\phi \wedge \psi]]\ P\ \mathfrak{s} \Leftrightarrow \\
\quad \Pi d : \mathbb{R}_{\geq 0}.\, \Pi sol : [0,d] \Rightarrow \mathbb{R}. \\
\qquad (sol, \mathfrak{s}, d \vDash x' = f) \\
\quad \Rightarrow (\Pi t : [0,d].\ulcorner \phi \wedge \psi \urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t))) \\
\quad \Rightarrow P\ (\mathsf{set}\ \mathfrak{s}\ (x,x')\ (sol\ d, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ d))))
\end{array}
$$

Want to show

$$\begin{aligned}&[[x' = f \,\&\, \phi]]\ P\ \mathfrak{s} \Leftrightarrow \\ &\quad \Pi d : \mathbb{R}_{\geq 0}.\, \Pi sol : [0,d] \Rightarrow \mathbb{R}. \\ &\qquad (sol, \mathfrak{s}, d \vDash x' = f) \\ &\quad \Rightarrow (\Pi t : [0,d].\, \ulcorner \phi \urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t))) \\ &\quad \Rightarrow P\ (\mathsf{set}\ \mathfrak{s}\ (x,x')\ (sol\ d, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ d))))\end{aligned}$$

So assume (B1) $(sol, \mathfrak{s}, d \vDash x' = f)$ and (B2) $(\Pi t : [0,d].\, \ulcorner \phi \urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t)))$. Now for every $t \in [0,d]$ have $\ulcorner \psi \urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t))$ from (D) because solutions and domain-constraints are prefix-closed: from (B1) and (B2) have (C1) $(sol, \mathfrak{s}, t \vDash x' = f)$ and (C2) $(\Pi r : [0,t].\, \ulcorner \phi \urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ r)))$. That is, (B3) $(\Pi t : [0,d].\, \ulcorner \psi \urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t)))$. Then (B4) $(\Pi t : [0,d].\, \ulcorner \phi \wedge \psi \urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t)))$ by conjunction with (B2). Applying (B1) and (B4) to (A) have $P\ (\mathsf{set}\ \mathfrak{s}\ (x,x')\ (sol\ d, f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ d))))$ as desired.

Case DW: Assume (A)$[[x := *; x' := f; ?\psi]]\ P\ \mathfrak{s}$ so $[[x := *]]\ ([[x' := f]]\ ([[?\psi]]\ P))\ \mathfrak{s}$ and (A1):

$$\Pi v : \mathbb{R}.\, (\ulcorner \psi \urcorner\ (\mathsf{set}\ (\mathsf{set}\ \mathfrak{s}\ x\ v)\ x'\ (f\ (\mathsf{set}\ \mathfrak{s}\ x\ v))) \Rightarrow P\ (\mathsf{set}\ (\mathsf{set}\ \mathfrak{s}\ x\ v)\ x'\ (f\ (\mathsf{set}\ \mathfrak{s}\ x\ v))))$$

To show $[[x' = f \,\&\, \psi]]\ P\ \mathfrak{s}$ we assume some $d > 0$ and sol such that (B1) $(sol, \mathfrak{s}, d \vDash x' = f)$ (B2) $(\Pi t : [0,d].\, \ulcorner \psi \urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ t)))$. Specialize (B2) to $t = d$ which gives us $\ulcorner \psi \urcorner\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ d))$. Then we can apply Lemma 5.4 since $x' \notin \mathrm{FV}(\psi)$ by syntactic constraints so (C) $\ulcorner \psi \urcorner\ (\mathsf{set}\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ d))\ x'\ (f\ \mathsf{set}\ \mathfrak{s}\ x\ (sol\ d)))$. Specialize (A1) to $v = sol\ d$ and apply (C) yielding $P\ (\mathsf{set}\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ d))\ x'\ (f\ (\mathsf{set}\ \mathfrak{s}\ x\ (sol\ d))))$ as desired.

Case solve: Assume side condition (SC1) that term d satisfies $\{x, x', t, t'\} \cap \mathrm{FV}(d) = \emptyset$, so by Lemma 5.5 we know that d is constant, i.e.,

$$d\ \mathfrak{s} = d\ (\mathsf{set}\ \mathfrak{s}\ (x,t)\ (sol\ r, r)) = d\ (\mathsf{set}\ \mathfrak{s}\ (x,t,x',t')\ (sol\ r, r, f\ (\mathsf{set}\ \mathfrak{s}\ (x,t)\ (sol\ r, r)), 1))$$

for all r. Throughout this case we write $\hat{d}$ for the real number $d\ \mathfrak{s}$.

Assume side condition that sln solves the ODE on $[0,d]$ meaning there exists a function sol such that (SC2) $sol\ (s\ t) = sln\ s$ for states s such that $s\ t \in [0,\hat{d}]$ and $(sol, s, \hat{d} \vDash t' = 1, x' = f)$, meaning sol is a solution of the ODE.

Assume (G1) $\ulcorner t = 0 \urcorner\ \mathfrak{s}$ (G2) $\ulcorner d \geq 0 \urcorner\ \mathfrak{s}$ (D) $[[t := *; ?0 \leq t \leq d; x := sln]]\ (\ulcorner \psi \urcorner)\ \mathfrak{s}$ and (A) $[[t := d; ?t \geq 0; x := sln; x' := f; t' := 1]]\ P\ \mathfrak{s}$ so that (A1)

$$P\ (\mathsf{set}\ \mathfrak{s}\ (x,x',t,t')\ (sln\ (\mathsf{set}\ \mathfrak{s}\ t\ \hat{d}), f\ (\mathsf{set}\ \mathfrak{s}\ (x,t)\ (sln\ (\mathsf{set}\ \mathfrak{s}\ t\ \hat{d}))), \hat{d}, 1))$$

Want to show $[[\{t' = 1, x' = f \,\&\, \psi\}^d]]\ P\ \mathfrak{s}$, i.e., $\langle\langle t' = 1, x' = f \,\&\, \psi \rangle\rangle\ P\ \mathfrak{s}$ for which it suffices to show for some sol and d (specifically sol and d above) that (P1) $(sol, \mathfrak{s}, \hat{d} \vDash t' = 1, x' = f)$, (P2) $(\Pi r : [0,\hat{d}].\, \ulcorner \psi \urcorner\ (\mathsf{set}\ \mathfrak{s}\ (x,t)\ (sol\ r, r)))$, and (P3)

$$P\ (\mathsf{set}\ v\ (x,t,x',t')\ (sol\ \hat{d}, \hat{d}, f\ (\mathsf{set}\ \mathfrak{s}\ (x,t)\ (sol\ \hat{d}, \hat{d})), 1))$$

Fact (P1) is immediate by (SC2). To show (P2), note from (D) have

$$\begin{aligned} & \Pi r : [0,\hat{d}].\ulcorner\psi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (t,x)\ (r, sln\ (\mathsf{set}\ \mathfrak{s}\ t\ r))) \\ = \quad & \Pi r : [0,\hat{d}].\ulcorner\psi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (x,t)\ (sln\ (\mathsf{set}\ \mathfrak{s}\ t\ r), r)) \\ =_{(\text{SC2})}\ & \Pi r : [0,\hat{d}].\ulcorner\psi\urcorner\ (\mathsf{set}\ \mathfrak{s}\ (x,t)\ (sol\ r, r)) \end{aligned}$$

which is (P2) as desired.

Then (P3) follows from (A1) by specializing $t = d$ and since by (SC2) $sln\ (\mathsf{set}\ \mathfrak{s}\ t\ d) = sol\ \hat{d}$. This completes the case.

Case DG: Assume side condition that (SC) $y \notin \mathrm{FV}(\Gamma) \cup \mathrm{FV}(f_0) \cup \mathrm{FV}(f) \cup \mathrm{FV}(a) \cup \mathrm{FV}(b) \cup \mathrm{FV}(\psi) \cup \{x\}$. Also assume that (A) $[[y := f_0; \{x' = f, y' = a(x)y + b(x)\,\&\,\psi\}]]\ P\ \mathfrak{s}$ and then proceed to show $[[x' = f\,\&\,\psi; \{y := *; y' := *\}^d]]\ P\ \mathfrak{s}$ holds, which is the conclusion. Let $Q \equiv (\lambda s.\ \Sigma a : \mathbb{R}.\, \Sigma b : \mathbb{R}.\, P\ (\mathsf{set}\ \mathfrak{s}\ (y, y')\ (a, b)))$. Next, from assumption (A) we have

$$[[x' = f, y' = a(x)y + b(x)\,\&\,\psi]]\ P(\mathsf{set}\ \mathfrak{s}\ y\ (f_0\ s))$$

With witness $a = (f_0\ \mathfrak{s})$ then have (1) $\Sigma a : \mathbb{R}.\, [[x' = f, y' = a(x)y + b(x)\,\&\,\psi]]\ P\ (\mathsf{set}\ \mathfrak{s}\ y\ a)$. Next we apply rule M. Since for all s have $P\ s \Rightarrow (\Sigma a : \mathbb{R}.\, \Sigma b : \mathbb{R}.\, P\ (\mathsf{set}\ \mathfrak{s}\ (y, y')\ (a, b))) \Leftrightarrow Q\ s$ we have (2) $\Sigma a : \mathbb{R}.\, [[x' = f, y' = a(x)y + b(x)\,\&\,\psi]]\ Q\ (\mathsf{set}\ s\ y\ a)$. By (SC) and because y and y' are not semantic free variables of Q, then we can semantically apply rule DG of CdGL (Chapter 5) to (2) yielding (3) $[[x' = f\,\&\,\psi]]\ Q\ \mathfrak{s}$. Then note for all states $t : \mathcal{S}$ that $(Q\ t) \Rightarrow (\langle\!\langle y := *; y' := * \rangle\!\rangle\ P\ t) \Leftrightarrow ([[\{y := *; y' := *\}^d]]\ P\ t)$ by rules $\langle\!\langle^d \rangle\!\rangle$I, $\langle\!\langle ; \rangle\!\rangle$I, and $\langle : * \rangle$I, then by monotonicity rule M on (3) have $[[x' = f\,\&\,\psi]]\ Q\ \mathfrak{s} \Leftrightarrow [[x' = f\,\&\,\psi]]\ ([[\{y := *; y' := *\}^d]]\ P)\ \mathfrak{s} \Leftrightarrow [[x' = f\,\&\,\psi; \{y := *; y' := *\}^d]]\ P\ \mathfrak{s}$ as desired.

Case R$\langle * \rangle$: Let $J : \mathfrak{S} \Rightarrow \mathbb{T}_i$ for some i. This case is actually part of the inductive proof of Theorem 6.4. Assume (D1) $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash J\ \mathfrak{s}$ and (D2) $\mathfrak{s} : \mathfrak{S}, J\ s, \mathcal{M}_0\ s = \mathcal{M}\ s \succ \mathbf{0}\ s \vdash [[\alpha]]\ (\lambda t.\ J\ t \star \mathcal{M}_0\ t \succ \mathcal{M}\ t)\ s$ for all $s : \mathfrak{S}$ and (A) $[[\hat{\boldsymbol{\alpha}}^*; \beta]]\ P\ s = [[\hat{\boldsymbol{\alpha}}^*]]\ ([[\beta]]\ P)\ s$ for all $s : \mathfrak{S}$ to prove $[[\gamma]]\ P\ \mathfrak{s}$. Let N stand for the inductive step proof term corresponding to (D2). Here $\hat{\boldsymbol{\alpha}}, \beta$, and γ are defined as

$$\hat{\boldsymbol{\alpha}} \equiv ?\mathcal{M} \succ \mathbf{0}; \{\alpha^d\ \mathsf{mod}\ N\} \qquad \beta \equiv ?\mathbf{0} \succcurlyeq \mathcal{M} \qquad \gamma \equiv \alpha^{*d}$$

From (D1) and (D2), we can apply the reification IH to get: $\mathfrak{s} : \mathfrak{S}, J\ \mathfrak{s}, \mathcal{M}\ \mathfrak{s} \succ \mathbf{0}\ \mathfrak{s} \vdash \ulcorner\alpha^d\ \mathsf{mod}\ N \leq_{[]} \alpha^{d}\urcorner\ \mathfrak{s}$ and by rule [?]I then (IH) $\mathfrak{s} : \mathfrak{S}, J\ \mathfrak{s}, \mathcal{M}\ \mathfrak{s} \succ \mathbf{0}\ \mathfrak{s} \vdash \ulcorner\hat{\boldsymbol{\alpha}} \leq_{[]} \alpha^{d}\urcorner\ \mathfrak{s}$. By Theorem 6.3 also have (T) $\mathfrak{s} : \mathfrak{S}, J\ \mathfrak{s}, \mathcal{M}\ \mathfrak{s} \succ \mathbf{0}\ \mathfrak{s} \vdash [[\hat{\boldsymbol{\alpha}}]]\ (\lambda t.\ J\ t \star \mathcal{M}_0\ t \succ \mathcal{M}\ t)\ \mathfrak{s}$.

Prove $\langle\!\langle \alpha^* \rangle\!\rangle\ P\ \mathfrak{s}$ with metric $\mathcal{M}$ and invariant $(\lambda t.\ J\ t \star [[\alpha^*]]\ ([[\beta]]\ P)\ t)$. Show each premise. Below, abbreviate $Q \equiv (\lambda t.\ J\ t \star [[\hat{\boldsymbol{\alpha}}^*]]\ ([[\beta]]\ P)\ t \star \mathcal{M}_0\ t \succ \mathcal{M}\ t)$.

Premise (P1) $\ulcorner\Gamma\urcorner(\mathfrak{s}) \vdash J\ \mathfrak{s} \star [[\hat{\boldsymbol{\alpha}}^*]]\ ([[\beta]]\ P)\ \mathfrak{s}$ holds by (D1) and (A). Premise (P2)

$$\mathfrak{s} : \mathfrak{S}, J\ \mathfrak{s} \star [[\alpha^*]]([[\beta]]\ P)\ \mathfrak{s}, \mathcal{M}_0\ \mathfrak{s} = \mathcal{M}\ \mathfrak{s} \succ \mathbf{0}\ \mathfrak{s} \vdash \langle\!\langle \alpha \rangle\!\rangle\ Q\ \mathfrak{s}$$

By rule $\langle\!\langle^d \rangle\!\rangle$I the semantic functions $\langle\!\langle \alpha \rangle\!\rangle$ and $[[\alpha^d]]$ are equivalent, so it suffices to show

$$\mathfrak{s} : \mathfrak{S}, J\ \mathfrak{s} \star [[\hat{\boldsymbol{\alpha}}^*]]\ ([[\beta]]\ P)\ \mathfrak{s}, \mathcal{M}_0\ \mathfrak{s} = \mathcal{M}\ \mathfrak{s} \succ \mathbf{0}\ \mathfrak{s} \vdash [[\alpha^d]]\ Q\ \mathfrak{s}$$

Note (IH) applies since $J\ \mathfrak{s}$ and $\mathcal{M}\ \mathfrak{s} \succ \mathbf{0}\ \mathfrak{s}$ are assumed in the context, thus by rule R[·] it suffices to show (BR)

$$\mathfrak{s} : \mathfrak{S}, J\ \mathfrak{s}\star[[\hat{\boldsymbol{\alpha}}^*]]\ ([[\beta]]\ P)\ \mathfrak{s}, \mathcal{M}_0\ \mathfrak{s} = \mathcal{M}\ \mathfrak{s} \succ \mathbf{0}\ \mathfrak{s}$$
$$\vdash [[\hat{\boldsymbol{\alpha}}]]\ (\lambda t.\ J\ t\star[[\hat{\boldsymbol{\alpha}}^*]]\ ([[\beta]]\ P)\ t\star\mathcal{M}_0\ t \succ \mathcal{M}\ t)\ \mathfrak{s}$$

Recall by Theorem 6.2 that $\hat{\boldsymbol{\alpha}}$ is a system and so admits axiom []∧. By axiom []∧ on (BR) and by (D2) it suffices to show $\mathfrak{s} : \mathfrak{S}, J\ \mathfrak{s}\star[[\hat{\boldsymbol{\alpha}}^*]]\ ([[\beta]]\ P)\ \mathfrak{s}, \mathcal{M}_0\ \mathfrak{s} = \mathcal{M}\ \mathfrak{s} \succ \mathbf{0}\ \mathfrak{s} \vdash [[\hat{\boldsymbol{\alpha}}]]\ ([[\hat{\boldsymbol{\alpha}}^*]]\ ([[\beta]]\ P))\ \mathfrak{s}$. which is the right projection of assumption $[[\hat{\boldsymbol{\alpha}}^*]]\ ([[\beta]]\ P)\ \mathfrak{s}$.

(P3) $\mathfrak{s} : \mathfrak{S}, J\ \mathfrak{s}\star[[\alpha^*]]\ ([[\beta]]\ P)\ \mathfrak{s}, \mathbf{0}\ \mathfrak{s} \succcurlyeq \mathcal{M}\ \mathfrak{s} \vdash P\ \mathfrak{s}$. By base case of $[[\alpha^*]]\ ([[\beta]]\ P)\ \mathfrak{s}$ have $[[?\mathbf{0} \succcurlyeq \mathcal{M}]]\ P\ \mathfrak{s}$ so by modus ponens on $\mathbf{0} \succcurlyeq \mathcal{M}$ have $P\ \mathfrak{s}$. □

C.1.5 Reification

We now show the results on reification. Systemhood helps prove transfer, which helps prove refinement. Recall that Γ, α, ϕ, and M are in the *system-test* fragment of CdGL. A system-test formula or context contains system box modalities, first-order formulas, and formulas that are propositionally equivalent to them. That is, we prohibit diamond modalities, equivalently box modalities containing dualities, except those that can be eliminated by trivial propositional rewriting. A system-test game has only system-test formulas in tests and domain constraints. A system-test proof only ever introduces system-test formulas and games in the context. A proof of a system-test game is not automatically system-test, e.g., if it mentions a game in a cut. The system-test requirement can be relaxed to allow diamond and game formulas if they are trivially equivalent to some system-test formula.

The system-test requirement ensures that proof variables p can be reified because $\Gamma(p)$ can always be expressed as a system box modality. Only one case uses the relaxed requirement that $\Gamma(p)$ is *equivalent to* a box system modality or first-order formula: $\langle\cup\rangle$E. Normal-form proofs can contain case statements which are irreducible (i.e., whose branching depends on the state). Because case analysis is expressed through Angelic choices, rule $\langle\cup\rangle$E introduces Angelic formulas $\langle?\phi\rangle\rho$ and $\langle?\psi\rangle\rho$ to the context. The system-test requirement says ϕ, ψ, and ρ are first-order in $\langle\cup\rangle$E so that the context formulas are trivially equivalent to system-test formulas $\phi \wedge \rho$ and $\psi \wedge \rho$ respectively, so that the output of reification remains a system.

Theorem 6.2 (Systemhood)**.** *If $\Gamma \vdash \mathcal{A} : [\alpha]\phi$ in CdGL without refinement for system-test $\Gamma, \mathcal{A}$, and hybrid game α and $\mathcal{A} \rightsquigarrow_\alpha \boldsymbol{\alpha}$ then $\boldsymbol{\alpha}$ is a system, i.e., it does not contain dualities.*

Proof. Recall that the notations $M \rightsquigarrow_\alpha \boldsymbol{\alpha}$ and $\alpha\ \mathsf{mod}\ M = \boldsymbol{\alpha}$ are equivalent for $\Gamma \vdash M : [\alpha]\phi$. The proof is by induction on M. In the hypothesis case, it suffices that Γ is system-test. In each inductive case, the IH applies because renaming, assignment, and (system) tests preserve the fact that Γ is system-test. In each case the right-hand side of $M\ \mathsf{mod}\ \alpha$ does not contain $\cdot^d$ by inspection. □

Theorem 6.3 (Reification transfer)**.** *If $\Gamma \vdash \mathcal{A} : [\alpha]\phi$ in CdGL without refinement for system-test $\Gamma, \mathcal{A}$, and hybrid game α and $\mathcal{A} \rightsquigarrow_\alpha \boldsymbol{\alpha}$ then $\Gamma \vdash [\boldsymbol{\alpha}]\phi$ is provable in CdGL without refinement.*

Proof. Recall that the notations $M \rightsquigarrow_\alpha \boldsymbol{\alpha}$ and $\alpha \,\mathsf{mod}\, M = \boldsymbol{\alpha}$ are equivalent for $\Gamma \vdash M : [\alpha]\phi$. The proof is by induction on the normal natural deduction proof M. However, the game argument α also serves an important role in the proof structure: we terminate early when game α is exhausted. The reason for the use of early termination is subtle: we were asked to only reify the strategy for α, even if the postcondition ϕ is another modality $[\beta]\psi$. Early termination avoids reifying the strategy for any game β which appears in a nested modality. The nested modality case $[\alpha][\beta]\phi$ corresponds directly to the sequential composition case $[\alpha;\beta]\phi$ by rule $\langle\!\langle;\rangle\!\rangle$I. Because sequential compositions appear so extensively in this proof, we use the equivalence $[\alpha][\beta]\phi \leftrightarrow [\alpha;\beta]\phi$ implicitly.

While sequential composition and nested modalities are responsible for much of the complexity in this proof, we found it surprisingly useful to assume without loss of generality[2] that the game argument is *always* a sequential composition. That is, rather than writing α for the game argument, we will write $\alpha; L$ for the game argument for some games α and L (mnemonic: "list" of games). We found the use of a sequential composition $\alpha; L$ useful because it allows us to more easily describe how the game argument changes in each recursive call: α represents the game currently being operated on while L represents any remaining game which must be reified after α has been processed. Careful maintenance of the game argument is crucial when the top-level postcondition is a nested modality $[\alpha; L][\beta]\phi$: throughout the recursive calls, α and L will take on different values, but they will never contain the game β, whose reification we are explicitly avoiding. The composition structure is particularly useful when $\alpha; L$ has shape $\{\gamma;\delta\}; L$: the game is reassociated to $\gamma; \{\delta; L\}$ so that α may be reified first after which $\{\beta; L\}$ are revisited. In the sense that the sequential case prepends a game β to L, game L serves as a stack.

We do not explicitly write the base cases where the game argument is an atomic game rather than one of shape $\alpha; L$. The base cases follow from the cases presented here by letting $L = \mathsf{skip} = ?\mathit{true}$ so that $\alpha; L = \alpha$. In every case of the induction, we assume that the top-level postcondition is a modality of shape $[\alpha; L]\phi$, rather than the shape $[\alpha]\phi$ written in rule definitions. In every case, we assume the (CdGL) proof starts by applying rule $\langle\!\langle;\rangle\!\rangle$I followed immediately by an introduction rule for α, so that the postconditions of the premises are of shape $[L]\psi$ and so that it suffices by rule $\langle\!\langle;\rangle\!\rangle$I for us to show $[\alpha][L]\phi$. By packing and unpacking compositions in this way, the bulk of the proof can operate on nested modalities $[\alpha][L]\phi$ for greater proof convenience, yet the game argument $\alpha; L$, which is a sequential composition, serves to ensure correct early termination. Technically speaking, our induction principle is not strict structural induction on derivations, but structural induction on derivations modulo ; introduction and elimination, because we implicitly introduce and eliminate sequential compositions when applying the IH.

Let (SC) denote the side condition that α and Γ are system-test. The ODE cases rely on a notion of normal ODE proof (Bohrer & Platzer, 2019) where rule DI are only used to prove cuts DC, since this normal form (which always exists) obviates the need to present a refinement rule for DI.

Case hyp: Let p denote the (variable) proof term for hyp. Because the conclusion is

[2]No generality is lost because non-composition game α is equivalent to the composition $\alpha; \mathsf{skip} \equiv \alpha; ?\mathit{true}$.

$[\alpha]\phi$, then $p : [\alpha]\phi$. By the system-test assumption, α is a system. To complete the case, note $\alpha \bmod p \equiv \alpha$ and $\alpha \leq_{[]} \alpha$ reflexively by rule refl.

Case $\langle\cup\rangle$E: Recall the definition of system-test (Def. 6.2 in Section 6.5.2) as applied to rule $\langle\cup\rangle$E: the system-test assumption requires that the conclusion of A is a formula of shape $\langle ?\phi \cup ?\psi\rangle\rho$ where ϕ, ψ, and ρ are first-order. While this restriction is highly specific, it owes to the fact that, in practice, irreducible cases are usually case analyses on term comparisons, which are first-order. Note that user-level proofs may use *reducible* cases on other formulas, since this proof assumes the input has been normalized. Even in normal-form proofs, the general-purpose choice game *introduction* rule may be used freely; this case concerns the elimination rule. Because this case has already used variable name ϕ in the scrutinee, we write uppercase Φ in this case specifically for the formula usually called ϕ in other cases. For example, the conclusion of this case is $\Gamma \vdash [\alpha; L \bmod M]\Phi$.

By the case assumption that M has form $\langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle$ and by inversion, assume A is a proof of (D1) $\Gamma \vdash \langle ?\phi \cup ?\psi\rangle\rho$, B is a proof of $\Gamma, \langle ?\phi\rangle\rho \vdash [\alpha][L]\Phi$, and C is a proof of $\Gamma, \langle ?\psi\rangle\rho \vdash [\alpha][L]\Phi$. where the latter two are definitionally equivalent to (D2) $\Gamma, \phi \wedge \rho \vdash [\alpha][L]\Phi$ and (D3) $\Gamma, \psi \wedge \rho \vdash [\alpha][L]\Phi$. In this case:

$$\alpha; L \bmod \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C\rangle \equiv \{?\phi \wedge \rho; \{\alpha; L \bmod B\}\} \cup \{?\psi \wedge \rho; \{\alpha; L \bmod C\}\}$$

so it suffices to show $[\{?\phi \wedge \rho; \{\alpha; L \bmod B\}\} \cup \{?\psi \wedge \rho; \{\alpha; L \bmod C\}\}]\Phi$.

By applying the inductive hypotheses on (D2) and (D3) we have (B2) $\Gamma, \phi \wedge \rho \vdash [\alpha; L \bmod B]\Phi$ (B3) $\Gamma, \psi \wedge \rho \vdash [\alpha; L \bmod C]\Phi$ so by rule [?]I and rule $\langle;\rangle$I have (2) $\Gamma \vdash [?\phi \wedge \rho; \{\alpha; L \bmod B\}]\Phi$ (3) $\Gamma \vdash [?\psi \wedge \rho; \{\alpha; L \bmod C\}]\Phi$ and by rule $[\cup]$I have $\Gamma \vdash [\{?\phi \wedge \rho; \{\alpha; L \bmod B\}\} \cup \{?\psi \wedge \rho; \{\alpha; L \bmod C\}\}]\Phi$ as desired.

Case $\langle*\rangle$C: This case holds vacuously because a proof term of $\langle*\rangle$C is never system-test: it always introduces a diamond formula to the context. The fact that $\langle*\rangle$C proofs are non-system-test should not pose any practical limitations to completeness: case analysis in normal forms is predominantly used to inspect the state. If a proof of $\langle\alpha^*\rangle\phi$ depends on the state, it does so through some termination metric $\mathcal{M}$, which can be expressed with a disjunctive case $\langle\cup\rangle$E.

Case $[\cup]$I: In this case $\{\{\alpha \cup \beta\}; L\} \bmod (M, N) \equiv \{\{\alpha; L\} \bmod M\} \cup \{\{\beta; L\} \bmod N\}$.

Assume (D) $\Gamma \vdash [\alpha \cup \beta][L]\phi$ so (D1) $\Gamma \vdash [\alpha][L]\phi$ and (D2) $\Gamma \vdash [\beta][L]\phi$.

By the IHs have (B1) $\Gamma \vdash [\{\alpha; L\} \bmod M]\phi$ (B2) $\Gamma \vdash [\{\beta; L\} \bmod N]\phi$ so by rule $[\cup]$I have $\Gamma \vdash [\{\{\alpha; L\} \bmod M\} \cup \{\{\beta; L\} \bmod N\}]\phi$.

Case $\langle ?\rangle$I: In this case $\{?\psi^d; L\} \bmod (M, N) \equiv L \bmod N$. Assume (D) $\Gamma \vdash \langle ?\psi\rangle[L]\phi$ so (D1) $\Gamma \vdash \psi$ and (D2) $\Gamma \vdash [L]\phi$. By the IH on (D2) have $\Gamma \vdash [L \bmod N]\phi$ as desired.

Case [?]I: In this case $\{?\psi; L\} \bmod (\lambda q : \psi.\ M) \equiv \{?\psi; \{L \bmod M\}\}$. Assume (D) $\Gamma \vdash [?\psi][L]\phi$ so that (D1) $\Gamma, \psi \vdash [L]\phi$. Since $?\psi$ and Γ are system-test, (Γ, ψ) is too and the IH applies giving $\Gamma, \psi \vdash [L \bmod M]\phi$. Then by rules [?]I and $\langle;\rangle$I have $\Gamma \vdash [?\psi][L \bmod M]\phi$ and $\Gamma \vdash [?\psi; \{L \bmod M\}]\phi$ by rule $\langle;\rangle$I as desired.

Case $\langle\cup\rangle$I1: In this case $\{\{\alpha \cup \beta\}^d; L\} \bmod \langle \ell \cdot M\rangle \equiv \{\alpha^d; L\} \bmod M$. Assume (D) $\Gamma \vdash \langle\alpha \cup \beta\rangle[L]\phi$, with premise (D1) $\Gamma \vdash \langle\alpha\rangle[L]\phi = \Gamma \vdash [\alpha^d; L]\phi$. By the IH on (D1) have $\Gamma \vdash [\{\alpha^d; L\} \bmod M]\phi$ as desired.

Case $\langle\cup\rangle$I2: In this case $\{\{\alpha \cup \beta\}^d; L\} \bmod r \cdot M \equiv \{\beta^d; L\} \bmod M$. Assume (D) Γ

$\vdash \langle\alpha \cup \beta\rangle[L]\phi$, with premise (D1) $\Gamma \vdash \langle\beta\rangle[L]\phi = \Gamma \vdash [\beta^d; L]\phi$. By the IH on (D1) have $\Gamma \vdash [\{\beta^d; L\} \mathsf{mod}\ M]\phi$ as desired.

Case $\langle*\rangle$S: In this case $\{\{\alpha^*\}^d; L\} \mathsf{mod} \langle \ell \cdot M\rangle \equiv L \mathsf{mod} M$. Assume (D) $\Gamma \vdash \langle\alpha^*\rangle[L]\phi$ with premise (D1) $\Gamma \vdash [L]\phi$. By the IH on (D1) have $\Gamma \vdash [L \mathsf{mod} M]\phi$ as desired.

Case $\langle*\rangle$G: In this case $\{\{\alpha^*\}^d; L\} \mathsf{mod}\ r \cdot M \equiv \{\{\alpha; \alpha^*\}^d; L\} \mathsf{mod}\ M$. Assume (D) $\Gamma \vdash \langle\alpha^*\rangle[L]\phi$, with premise (D1) $\Gamma \vdash \langle\alpha\rangle\langle\alpha^*\rangle[L]\phi = \Gamma \vdash [\{\alpha; \alpha^*\}^d; L]\phi$. By the IH on (D1) have $\Gamma \vdash [\{\{\alpha; \alpha^*\}^d; L\} \mathsf{mod}\ M]\phi$ as desired. The well-foundedness (corr. termination) argument for this case is trivial: this is one case of a proof by induction on the structure of the CdGL derivation, so this case applies an IH to a subderivation from which the result follows after a finite number of algebraic equivalences described immediately above. Since every CdGL derivation has finitely many proof steps, the induction is well-founded.

Case [:*]I: In this case $\{x := *; L\} \mathsf{mod}\ (\lambda x : \mathbb{R}.\ M) \equiv x := *; \{L \mathsf{mod} M\}$.

Assume (D) $\Gamma \vdash [x := *][L]\phi$ with premise (D1) $\Gamma\frac{y}{x} \vdash [L]\phi$. Since $\Gamma\frac{y}{x}$ is system-test, the IH applies and $\Gamma\frac{y}{x} \vdash [L \mathsf{mod} M]\phi$ so by rules [:*]I and $\langle\!\langle;\rangle\!\rangle$I have $\Gamma \vdash [x := *; \{L \mathsf{mod} M\}]\phi$ as desired.

Case $\langle:*\rangle$I: In this case $\{\{x := *\}^d; L\} \mathsf{mod} \langle f\frac{y}{x} :* p.\ M\rangle \equiv x := f; \{L \mathsf{mod} M\}$.

Assume (D) $\Gamma \vdash \langle x := *\rangle[L]\phi$ with premise (D1) $\Gamma\frac{y}{x}, x = f\frac{y}{x} \vdash [L]\phi$. Note $(\Gamma\frac{y}{x}, x = f\frac{y}{x})$ is system-test. By the IH on (D1) have (B1) $\Gamma\frac{y}{x}, x = f\frac{y}{x} \vdash [L \mathsf{mod} M]\phi$ so by rules $\langle\!\langle:=\rangle\!\rangle$I and $\langle\!\langle;\rangle\!\rangle$I have $\Gamma \vdash [x := f; \{L \mathsf{mod} M\}]\phi$ as desired.

Case $\langle\!\langle;\rangle\!\rangle$I: We prove the case for Demon. The case for Angel is symmetric because ; is self-dual, i.e., because $\{\alpha; \beta\}^d \cong \alpha^d; \beta^d$ for all α and β as expressed by rule $;^d$. In reification result is $\{\{\alpha; \beta\}; L\} \mathsf{mod} \langle\!\langle \iota\ M\rangle\!\rangle \equiv \{\alpha; \{\beta; L\}\} \mathsf{mod}\ M$ for this case. Assume (D) $\Gamma \vdash [\alpha; \beta][L]\phi$ with premise (D1) $\Gamma \vdash [\{\alpha; \beta\}; L]\phi = \Gamma \vdash [\alpha; \{\beta; L\}]\phi$ so by IH have $\Gamma \vdash [\{\alpha; \{\beta; L\}\} \mathsf{mod}\ M]\phi$ as desired.

Case $\langle\!\langle:=\rangle\!\rangle$I: We prove the case for Demon. The Angelic case is symmetric since assignment is self-dual, i.e., since $\{x := f\}^d \cong x := f$ for all x and f as expressed by rule $:=^d$. In this case $\{x := f; L\} \mathsf{mod} \langle\!\langle x := f\frac{y}{x} \text{ in } p.\ M\rangle\!\rangle \equiv x := f; \{L \mathsf{mod} M\}$. Assume (D) $\Gamma \vdash [x := f][L]\phi$ with premise (D1) $\Gamma\frac{y}{x}, x = f\frac{y}{x} \vdash [L]\phi$. Note context $(\Gamma\frac{y}{x}, x = f\frac{y}{x})$ is system-test. By the IH on (D1) have (B1) $\Gamma\frac{y}{x}, x = f\frac{y}{x} \vdash [L \mathsf{mod} M]\phi$ so by rules $\langle\!\langle:=\rangle\!\rangle$I and $\langle\!\langle;\rangle\!\rangle$I have $\Gamma \vdash [x := f; \{L \mathsf{mod} M\}]\phi$ as desired.

Case [*]I: In this case $\{\alpha^*; L\} \mathsf{mod}\ (M \text{ rep } \psi.\ N \text{ in } O) \equiv \{\alpha \mathsf{mod} N\}^*; \{L \mathsf{mod} O\}$. Assume (D) $\Gamma \vdash [\alpha^*; L]\phi$ with premises (D1) $\Gamma \vdash \psi$ (D2) $\psi \vdash [\alpha]\psi$ (D3) $\psi \vdash [L]\phi$. The system-test condition says that ψ does not contain game or diamond modalities.

By IH on (D2) have (B2) $\psi \vdash [\alpha \mathsf{mod} N]\psi$ and by IH on (D3) have (B3) $\psi \vdash [L \mathsf{mod} O]\phi$ so by rule [*]I on (D1) (B2) (B3) have $\Gamma \vdash [\{\alpha \mathsf{mod} N\}^*][L \mathsf{mod} O]\phi$ so that we can apply rule $\langle\!\langle;\rangle\!\rangle$I to get $\Gamma \vdash [\{\alpha \mathsf{mod} N\}^*; \{L \mathsf{mod} O\}]\phi$.

The well-foundedness (corr. termination) argument for this case is trivial: this is one case of a proof by induction on the structure of the CdGL derivation, so this case applies an IH to subderivations from which the result follows after a finite number of rule applications described immediately above. Because every CdGL derivation contains finitely many proof steps, the induction is well-founded.

Case $\langle*\rangle$I: In this case:

$$\begin{aligned}&\{\{\alpha^*\}^d; L\} \mathsf{mod}\, \mathrm{for}(p:\varphi(\mathcal{M})=M; q; N)\,\{\alpha\}\,O\\ \equiv\ &\{?\mathcal{M} \succ \mathbf{0}; \{\alpha^d \mathsf{mod}\, N\}\}^*; ?\mathbf{0} \succcurlyeq \mathcal{M}; \{L \mathsf{mod}\, O\}\end{aligned}$$

Assume (D) $\Gamma \vdash \langle\alpha^*\rangle[L]\phi$ with premises (D1) $\Gamma \vdash \varphi$ (D2) $\varphi, \mathcal{M}_0 = \mathcal{M} \succ \mathbf{0} \vdash \langle\alpha\rangle(\varphi \wedge \mathcal{M}_0 \succ \mathcal{M})$ (D3) $\varphi, \mathbf{0} \succcurlyeq \mathcal{M} \vdash [L]\phi$. The system-test condition says that φ does not contain game or diamond modalities. By the IH on (D2) have (B2) $\varphi, \mathcal{M}_0 = \mathcal{M} \succ \mathbf{0} \vdash [\alpha \mathsf{mod}\, N](\varphi \wedge \mathcal{M}_0 \succ \mathcal{M})$. By the IH on (D3) have (B3) $\varphi, \mathbf{0} \succcurlyeq \mathcal{M} \vdash [L \mathsf{mod}\, O]\phi$. In this case it suffices to show $\Gamma \vdash [\{?\mathcal{M} \succ \mathbf{0}; \{\alpha \mathsf{mod}\, N\}\}^*][?\mathbf{0} \succcurlyeq \mathcal{M}][L \mathsf{mod}\, O]\phi$, using rule $[*]$I with loop invariant φ by showing premises (P1) $\Gamma \vdash \varphi$, (P2) $\varphi \vdash [?\mathcal{M} \succ \mathbf{0}; \{\alpha \mathsf{mod}\, N\}]\varphi$, and (P3) $\varphi \vdash [?\mathbf{0} \succcurlyeq \mathcal{M}][L \mathsf{mod}\, O]\phi$.

Fact (P1) holds immediately by (D1). To show (P2) it suffices by rules $\langle;\rangle$I and [?]I to show $\varphi, \mathcal{M} \succ \mathbf{0} \vdash [\alpha \mathsf{mod}\, N]\varphi$, then since $\mathcal{M}_0$ is fresh it suffices by discrete ghost iG to show $\varphi, \mathcal{M}_0 = \mathcal{M} \succ \mathbf{0} \vdash [\alpha \mathsf{mod}\, N]\varphi$, which follows from (B2) by monotonicity rule M and projection. To show (P3) it suffices by rule [?]I to show $\varphi, \mathbf{0} \succcurlyeq \mathcal{M} \vdash [L \mathsf{mod}\, O]\phi$, which is (B3).

This completes case $\langle*\rangle$I.

The well-foundedness (corr. termination) argument for this case is trivial: this is one case of a proof by induction on the structure of the CdGL derivation, so this case applies an IH to subderivations from which the result follows after a finite number of rule applications described immediately above. Because every CdGL derivation contains finitely many proof steps, the induction is well-founded. Notably, no inner meta-logical induction is performed in this case, we simply apply a CdGL proof rule ($\langle*\rangle$I) which happens to express induction arguments for loops in the object language.

Case DI: In a normal-form proof, DI only occurs on the left premise of DC, and the DC proof does not apply the IH on the left premise, thus no case for DI is needed.

Case DV: As with the DI case, the DV case never arises for normal-form proofs.

Case DC: Have $\{x' = f \,\&\, \psi; L\} \mathsf{mod}\, DC(M, N) \equiv \{x' = f \,\&\, \psi \wedge \rho; L\} \mathsf{mod}\, N$ where ρ is the differential cut formula for $x' = f \,\&\, \psi$ proved by M.

Assume (D) $\Gamma \vdash [x' = f \,\&\, \psi][L]\phi$ with premises (D1) $\Gamma \vdash [x' = f \,\&\, \psi]\rho$ and (D2) $(\Gamma \vdash [x' = f \,\&\, \psi \wedge \rho][L]\phi) \leftrightarrow (\Gamma \vdash [x' = f \,\&\, \psi \wedge \rho; L]\phi)$. By IH on (D2) have, as desired, (B2)

$$\Gamma \vdash [\{x' = f \,\&\, \psi \wedge \rho; L\} \mathsf{mod}\, N]\phi$$

Case DW: Have $\{x' = f \,\&\, \psi; L\} \mathsf{mod}\, DW(M) \equiv \{x := *; x' := f; ?\psi; \{L \mathsf{mod}\, M\}\}$ as the case assumption. Assume (D) $\Gamma \vdash [x' = f \,\&\, \psi][L]\phi$ with premise (D1) $\Gamma^y_x, \psi \vdash [L]\phi$. Recall that as always Γ^y_x is a uniform renaming which renames x to some fresh y. In DW in particular, fresh renaming of x to y is important because the test of ψ should occur in the *final* state of the ODE, whose value of x generally differs from the initial value. By the IH, (B1) $\Gamma^y_x, \psi \vdash [L \mathsf{mod}\, M]\phi$. By weakening, (B2) $\Gamma^y_x, x' = f, \psi \vdash [L \mathsf{mod}\, M]\phi$. By [?]I, $\langle:=\rangle$I, and $[:*]$I, have $\Gamma \vdash [x := *][x' := f][?\psi][L \mathsf{mod}\, M]\phi$. Then $\Gamma \vdash [x := *; x' := f; ?\psi; \{L \mathsf{mod}\, M\}]\phi$ follows by $\langle;\rangle$I as desired.

Case DG: In this case:

$$\begin{aligned} & \{x' = f \,\&\, \psi; \{y := *; y' := *\}^d; L\} \bmod DG(f_0, a, b, M) \\ \equiv\ & y := f_0; \{\{x' = f, y' = a(x)y + b(x) \,\&\, \psi; L\} \bmod M\} \end{aligned}$$

Note that when the game $x' = f \,\&\, \psi; \{y := *; y' := *\}^d; L$ is played using the strategy generated by rule DG, the strategy plays the Angelic assignments to y and y' by assigning y and y' the final values of y and y' from the extended ODE $x' = f, y' = a(x)y + b(x) \,\&\, \psi$. See the soundness proof of rule DG in CdGL (Appendix B.2) for details.

Assume (D) $\Gamma \vdash [x' = f \,\&\, \psi; \{y := *; y' := *\}^d; L]\phi$ with premise (D1)

$$\Gamma, y = f_0 \vdash [x'{=}f, y' = a(x)y + b(x) \,\&\, \psi; L]\phi$$

By the IH, have (B1)

$$\Gamma, y = f_0 \vdash [\{x' = f, y' = a(x)y + b(x) \,\&\, \psi; L\} \bmod M]\phi$$

and by rules ⟦:=⟧I and ⟦;⟧I have

$$\Gamma \vdash [y := f_0; \{\{x' = f, y' = a(x)y + b(x) \,\&\, \psi; L\} \bmod M\}]\phi$$

as desired.

Case dsolve: In this case, have

$$\begin{aligned} & \{t := 0; \{t' = 1, x' = f \,\&\, \psi\}^d; L\} \bmod AS(d, sln, M, N) \\ \equiv\ & \{t := d; x := sln; x' := f\}; \{L \bmod M\} \end{aligned}$$

Note that because rule dsolve is the solution rule for an *Angelic* ODE, the duration of the ODE is specified in proof term $AS(d, sln, M, N)$ by the CdGL term d and the reification result chooses d for the duration.

Assume (D) $\Gamma \vdash \langle t := 0; x' = f \,\&\, \psi\rangle[L]\phi$ with premises (1) $\Gamma \vdash d \geq 0$, (2) $\Gamma^y_x, 0 \leq t \leq d, x = sln^y_x, x' = f \vdash \psi$, and (3) $\Gamma^y_x, 0 \leq t = d, x = sln^y_x, x' = f \vdash [L]\phi$.

The IH on (3) gives (4) $\Gamma^y_x, 0 \leq t = d, x = sln^y_x, x' = f \vdash [L \bmod M]\phi$. Satisfy the assumption $0 \leq d$ by taking (1) and applying Lemma 5.4 using the freshness side condition on d to get (5) $\Gamma^y_x, t = d, x = sln^y_x, x' = f \vdash [L \bmod M]\phi$.

By ⟦:=⟧I and ⟦;⟧I on (4) (and the remaining variable occurrence assumptions) we have that $\Gamma \vdash [t := d; x := sln; x' := f; \{L \bmod M\}]\phi$. By ;A, one my step gives, as desired,

$$\Gamma \vdash [\{t := d; x := sln; x' := f\}; \{L \bmod M\}]\phi$$

Case bsolve: In this case, have:

$$\{t := 0; \{t' = 1, x' = f \,\&\, \psi; L\}\} \bmod DS(sol, M) \equiv t := 0; \{t' = 1, x' = f \,\&\, \psi\}; \{L \bmod M\}$$

Note that because rule bsolve is the solution rule for a *Demonic* ODE, the duration of the ODE is chosen by Demon, so the reification result must be a nondeterministic system which allows Demon to choose the ODE duration. One approach for the definition of reification

would be to generate a discrete system similar to the dsolve case except that a Demonic nondeterministic assignment determines the duration. Instead, we output a (Demonic) ODE to emphasize the idea that reification makes Angel commit to the strategy specified by the proof, but allows Demon to play an arbitrary strategy.

Assume (D) $\Gamma \vdash [t := 0; \{t' = 1, x' = f \,\&\, \psi\}][L]\phi$ with side condition (SC) $\{t, t'\} \cap \mathrm{FV}(\Gamma) = \emptyset$. The premise is (D1) $\Gamma\frac{y}{x}, t \geq 0, \hat{\psi}, x = sln\frac{y}{x}, x' = f \vdash [L]\phi$ where $\hat{\psi} \equiv \forall 0 \leq s \leq t\, [t := s; x := sln]\psi$.

By the IH on (D1) have (B1) $\Gamma\frac{y}{x}, t \geq 0, \hat{\psi}, x = sln\frac{y}{x}, x' = f \vdash [L \bmod M]\phi$.

Apply bsolve on (B1) with (SC) to get $\Gamma \vdash [t := 0; \{t' = 1, x' = f \,\&\, \psi\}][L \bmod M]\phi$ which by $\langle\!\langle;\rangle\!\rangle$I gives $\Gamma \vdash [t := 0; \{t' = 1, x' = f \,\&\, \psi\}; \{L \bmod M\}]\phi$ as desired. □

Theorem 6.4 (Reification refinement)**.** *If $\Gamma \vdash \mathcal{A} : [\alpha]\phi$ in CdGL without refinement for system-test $\Gamma, \mathcal{A}$, and hybrid game α and $\mathcal{A} \rightsquigarrow_\alpha \boldsymbol{\alpha}$ then $\Gamma \vdash \boldsymbol{\alpha} \leq_{[]} \alpha$ is provable in CdGL with refinement.*

Proof. Recall that the notations $M \rightsquigarrow_\alpha \boldsymbol{\alpha}$ and $\alpha \bmod M = \boldsymbol{\alpha}$ are equivalent for $\Gamma \vdash M : [\alpha]\phi$. The proof is by induction on the normal natural deduction proof M, with the same induction technique as in Theorem 6.3.

Case $\langle * \rangle$C: In this case have $\Gamma \vdash A : (\phi \vee \psi)$ and

$$\alpha \bmod \langle \text{case } A \text{ of } \ell \Rightarrow\ B \mid r \Rightarrow\ C \rangle \equiv \{?\phi; \{L \bmod B\}\} \cup \{?\psi; \{L \bmod C\}\}$$

and want to show $\{?\phi; \{L \bmod B\}\} \cup \{?\psi; \{L \bmod C\}\} \leq_{[]} L$. By the IHs on B and C have: (1) $\Gamma, \phi \vdash L \bmod B \leq_{[]} L$ and (2) $\Gamma, \psi \vdash L \bmod C \leq_{[]} L$. Note rule ;S applies for systems $?\phi$ and $?\psi$. By rules ;S and [∪]R it suffices to show $\{?\phi; L\} \cup \{?\psi; L\} \leq_{[]} L$, which we show:

$$\begin{array}{lll} & \{?\phi; L\} \cup \{?\psi; L\} & \\ \leq_{[]} & \{?\phi \cup ?\psi\}; L & \text{by } ;\mathrm{d}_r \\ \leq_{[]} & ?(\phi \vee \psi); L & \text{def. of } \vee \\ \leq_{[]} & \mathsf{skip}; L & \text{by (A) and [?]} \\ \leq_{[]} & L & \text{by } ;\mathrm{id}_l \end{array}$$

This completes the case for $\langle * \rangle$C.

The next two cases $\langle\!\langle;\rangle\!\rangle$I and $\langle\!\langle:=\rangle\!\rangle$I hold symmetrically for both Angel and Demon.

Case $\langle\!\langle;\rangle\!\rangle$I: In this case, have $\{\{\alpha; \beta\}; L\} \bmod M \equiv \{\alpha; \{\beta; L\}\} \bmod M$. The IH is applicable with proof M because $\{\alpha; \beta\}; L \cong \alpha; \{\beta; L\}$ by rule ;A. By the IH we have $\{\alpha; \{\beta; L\}\} \bmod M \leq_{[]} \alpha; \{\beta; L\}$. By rule ;A have $\{\alpha; \beta\}; L \cong_{[.]} \alpha; \{\beta; L\}$ so by rule trans have $\{\alpha; \{\beta; L\}\} \bmod M \leq_{[]} \{\alpha; \beta\}; L$ as desired.

Case $\langle\!\langle:=\rangle\!\rangle$I: Have $\{x := f; L\} \bmod \langle\!\langle x := f\frac{y}{x} \text{ in } p.\ M\rangle\!\rangle \equiv \{x := f; \{L \bmod M\}\}$ as the case assumption. The proof follows by rule ;S on system $x := f$, a rule with two premises. The first premise of rule ;S is $\Gamma \vdash x := f \leq_{[]} x := f$ is immediate by rule refl. We show the second premise $\Gamma \vdash [x := f](L \bmod M \leq_{[]} L)$. By inversion on M have $(\Gamma\frac{y}{x}, x = f\frac{y}{x}) \vdash M : [L]\phi$, for fresh y, so the IH gives $(\Gamma\frac{y}{x}, x = f\frac{y}{x}) \vdash L \bmod M \leq_{[]} L$, then by rule $\langle\!\langle:=\rangle\!\rangle$I have $\Gamma \vdash [x := f](L \bmod M \leq_{[]} L)$, which suffices.

Angel cases:

Case $\langle:*\rangle$I: In this case, have $\{\{x := *\}^d; L\} \mathsf{mod} \langle f\frac{y}{x} :* p.\ M\rangle \equiv \{x := f; \{L \mathsf{mod} M\}\}$. By rule ;S on system $x := f$, suffices to show $\Gamma \vdash x := f \leq_{[]} x := *^d$ (which holds immediately by rules $\langle:*\rangle$, trans, and $:=^d$) and $\Gamma \vdash [x := f](L \mathsf{mod} M \leq_{[]} L)$ which follows by rule $\langle:*\rangle$I from the IH on M, i.e., from $\Gamma\frac{y}{x}, x = f\frac{y}{x} \vdash L \mathsf{mod} M \leq_{[]} M$ for fresh y.

Case $\langle?\rangle$I: In this case, have $\{?\psi^d; L\} \mathsf{mod} (M, N) \equiv L \mathsf{mod} N$. By rules trans and $;\mathrm{id}_l$ it suffices to show $\Gamma \vdash \mathsf{skip}; \{L \mathsf{mod} N\} \leq_{[]} ?\psi^d; L$. Then by rule ;S on system skip it suffices to show (L) $\Gamma \vdash \mathsf{skip} \leq_{[]} ?\psi^d$ and (R) $\Gamma \vdash [?\psi^d](L \mathsf{mod} N \leq_{[]} L)$. To show (L) it suffices by rule skip^d to show $\Gamma \vdash \mathsf{skip}^d \leq_{[]} ?\psi^d$, which holds by rule $\langle?\rangle$ because $\Gamma \vdash (\mathit{true} \to \psi)$ follows propositionally from $\Gamma \vdash M : \psi$. To show (R), it suffices to apply the IH on L giving $\Gamma \vdash L \mathsf{mod} N \leq_{[]} N$, then by rule $\langle?\rangle$I and $M : \psi$ have $\Gamma \vdash [?\psi^d](L \mathsf{mod} N \leq_{[]} N)$.

Case $\langle\cup\rangle$I1: In this case, have $\{\{\alpha \cup \beta\}^d; L\} \mathsf{mod} \langle \ell \cdot M\rangle \equiv \{\alpha^d; L\} \mathsf{mod} M$. By the IH, $\Gamma \vdash \{\alpha^d; L\} \mathsf{mod} M \leq_{[]} \alpha^d; L$, so by rule trans it suffices to show $\Gamma \vdash \alpha^d; L \leq_{[]} \{\alpha \cup \beta\}^d; L$. This follows by rule ;G because (L) $\Gamma \vdash \alpha^d \leq_{[]} \{\alpha \cup \beta\}^d$ and (R) $\cdot \vdash L \leq_{[]} L$. Premise (L) holds by rule $\langle\cup\rangle$R1. Premise (R) holds by rule refl, i.e., $L \leq_{[]} L$ holds.

Case $\langle\cup\rangle$I2: In this case, have $\{\{\alpha \cup \beta\}^d; L\} \mathsf{mod} r \cdot M \equiv \{\beta^d; L\} \mathsf{mod} M$. By the IH, $\Gamma \vdash \{\beta^d; L\} \mathsf{mod} M \leq_{[]} \beta^d; L$, so by rule trans it suffices to show $\Gamma \vdash \beta^d; L \leq_{[]} \{\alpha \cup \beta\}^d; L$. This follows by rule ;G because (L) $\Gamma \vdash \beta^d \leq_{[]} \{\alpha \cup \beta\}^d$ and (R) $\cdot \vdash L \leq_{[]} L$. Premise (L) holds by rule $\langle\cup\rangle$R2. Premise (R) holds by rule refl, i.e., $L \leq_{[]} L$ holds.

Case $\langle*\rangle$S: In this case, have $\{\{\alpha^*\}^d; L\} \mathsf{mod} \langle \ell \cdot M\rangle \equiv L \mathsf{mod} M$. Symmetric with case $\langle\cup\rangle$I1. By the IH, $\Gamma \vdash L \mathsf{mod} M \leq_{[]} L$ so by rule trans it suffices to show $\Gamma \vdash L \leq_{[]} \{\alpha^*\}^d; L$. This follows by rule ;S on system skip because (L) $\Gamma \vdash \mathsf{skip} \leq_{[]} \{\alpha^*\}^d$ and (R) $\Gamma \vdash [\{\alpha^*\}^d](L \leq_{[]} L)$. Premise (L) holds by rules roll_l and $\langle\cup\rangle$R1. Premise (R) holds by monotonicity rule M on $M : \langle\{\alpha^*\}^d; L\rangle\phi$ which by rule $\langle[;]\rangle$I equivalently proves $\langle\{\alpha^*\}^d\rangle\langle L\rangle\phi$ since have $\langle L\rangle\phi \vdash L \leq_{[]} L$ by rule refl ($L \leq_{[]} L$) and by weakening.

Case $\langle*\rangle$G: In this case, have $\{\{\alpha^*\}^d; L\} \mathsf{mod} r \cdot M \equiv \{\alpha^d; \{\alpha^*\}^d; L\} \mathsf{mod} M$. Symmetric with case $\langle\cup\rangle$I2. By the IH, $\Gamma \vdash \{\alpha^d; \{\alpha^*\}^d; L\} \mathsf{mod} M \leq_{[]} \alpha^d; \{\alpha^*\}^d; L$ so by rule trans it suffices to show $\Gamma \vdash \alpha^d; \{\alpha^*\}^d; L \leq_{[]} L$, more simply $\Gamma \vdash \{\alpha^d; \{\alpha^*\}^d\}; L \leq_{[]} L$ by rule ;A.

This follows by rule ;G because (L) $\Gamma \vdash \alpha^d; \{\alpha^*\}^d \leq_{[]} \{\alpha^*\}^d$ and (R) $\cdot \vdash L \leq_{[]} L$. Premise (L) holds by rules roll_l and $\langle\cup\rangle$R2. Premise (R) holds by rule refl ($L \leq_{[]} L$).

Case $\langle*\rangle$I: In this case, have:

$$\begin{aligned}&\{\{\alpha^*\}^d; L\} \mathsf{mod} \mathrm{for}(p : \varphi(\mathcal{M}) = M; q; N)\, \{\alpha\}\, O\\ \equiv\ &\{?\mathcal{M} \succ 0; \{\{\alpha^d\} \mathsf{mod} N\}\}^*; ?\mathbf{0} \succcurlyeq \mathcal{M}; \{L \mathsf{mod} O\}\end{aligned}$$

In this case we use the following abbreviations:

$$\hat{\alpha} \equiv\ ?\mathcal{M} \succ 0; \{\alpha^d \mathsf{mod} N\} \qquad \beta \equiv\ ?\mathbf{0} \succcurlyeq \mathcal{M} \qquad \gamma \equiv\ \{\alpha^*\}^d$$

We will rely on the following refinement rule. Its soundness proof for this rule is presented as a case of the main soundness proof, but is technically part of the present simultaneous induction. We present this rule only in the appendix because it is much more special-case than the other refinement rules.

$$(\mathrm{R}\langle*\rangle)\quad \frac{\Gamma\vdash\varphi \quad \varphi,\mathcal{M}_0=\mathcal{M}\succ\mathbf{0}\vdash\langle\alpha\rangle(\varphi\wedge\mathcal{M}_0\succ\mathcal{M})}{\Gamma\vdash\hat{\alpha}^*;\beta\leq_{[]}\gamma}$$

Where φ and $\mathcal{M}$ are per the proof (A) $\langle\alpha^*\rangle\phi$. From (A) we have (A1) $\Gamma\vdash\varphi$ and (A2) $\varphi,\mathcal{M}_0=\mathcal{M}\succ\mathbf{0}\vdash\langle\alpha\rangle(\varphi\wedge\mathcal{M}_0\succ\mathcal{M})$ and (A3) $\varphi,\mathbf{0}\succcurlyeq\mathcal{M}\vdash[L]\phi$. We repack (B) $\langle\alpha^*\rangle(\varphi\wedge\mathbf{0}\succcurlyeq\mathcal{M})$ by rule $\langle*\rangle$I on (A1) and (A2) proving the postcondition by rule hyp. The proof starts with rule ;S on system $\hat{\alpha}^*;\beta$, with premises (L) $\Gamma\vdash\hat{\alpha}^*;\beta\leq_{[]}\{\alpha^*\}^d$ and (R) $\Gamma\vdash[\hat{\alpha}^*;\beta](L\bmod O\leq_{[]}L)$.

Premise (L) is by rule $\mathrm{R}\langle*\rangle$ on (A1) and (A2). We show (R). By Theorem 6.3 on (B) then $\Gamma\vdash[\{\alpha^*\}^d\bmod B](\varphi\wedge\mathbf{0}\succcurlyeq\mathcal{M})$ and by inspection $\{\alpha^*\}^d\bmod B=\{\alpha^*\}^d\bmod A$ so (C) $\Gamma\vdash[\hat{\alpha}^*;\beta](\varphi\wedge\mathbf{0}\succcurlyeq\mathcal{M})$. By IH on (A3) have $\varphi,\mathbf{0}\succcurlyeq\mathcal{M}\vdash L\bmod O\leq_{[]}L$. Then by monotonicity rule M on (C) and (A3) have $\Gamma\vdash[\hat{\alpha}^*;\beta](L\bmod O\leq_{[]}L)$, satisfying the premise and completing the case.

Case dsolve: In this case the refinement i:

$$\begin{aligned}&\{t:=0;\{t'=1,x'=f\,\&\,\psi\}^d;L\}\bmod AS(d,sol,dom,M)\\ \equiv\ &\{t:=d;x:=sln;x':=f;t':=1\};\{L\bmod M\}\end{aligned}$$

By premises of dsolve have (D1) $\Gamma\vdash d\geq 0$, (D2) $\Gamma\frac{y}{x},0\leq t\leq d,x=sln\frac{y}{x},x'=f\vdash\psi$, and (D3) $\Gamma\frac{y}{x},0\leq t=d,x=sln\frac{y}{x},x'=f\vdash[L]\phi$ and by its side condition have (SC) $\{t,t'\}\cap\mathrm{FV}(\Gamma)=\emptyset$. By rule :=:= suffices to show $\Gamma\vdash\{t:=0;t:=d;x:=sln;x':=f\};\{L\bmod M\}\leq_{[]}t:=0;\{t'=1,x'=f\,\&\,\psi\}^d;L$ The proof continues with applying rule ;S twice on systems $t:=0$ and $t:=d;x:=sln;x':=f$. The first premise is $t:=0\leq_{[]}t:=0$ which proves trivially by rule refl. The second premise is $\Gamma\vdash[t:=0](t:=d;x:=sln;x':=f;t':=1\leq_{[]}\{t'=1,x'=f\,\&\,\psi\}^d)$ and by rule $\langle\!|:=|\!\rangle$I and $t\notin\mathrm{FV}(d)$ suffices to show (2) $\Gamma,t=0\vdash t:=d;x:=sln;x':=f\leq_{[]}\{t'=1,x'=f\,\&\,\psi\}^d$. By cutting in (D1) it suffices to show $\Gamma,t=0,d\geq 0\vdash t:=d;x:=sln;x':=f;t':=1\leq_{[]}\{t'=1,x'=f\,\&\,\psi\}^d$. By rule solve it suffices to show $\Gamma\vdash[t:=*;?0\leq t\leq d;x:=sln]\psi$ which follows from (D2) in several steps. In (D2), the assumption $x'=f$ can be strengthened away because x' is fresh in ψ and the context, giving (D2A) $\Gamma\frac{y}{x},0\leq t\leq d,x=sln\frac{y}{x}\vdash\psi$. Then by rules $\langle\!|:=|\!\rangle$I and [?]I have $\Gamma\vdash[?0\leq t\leq d][x:=sln]\psi$. By (SC) have $\{t,t'\}\cap\mathrm{FV}(\Gamma)=\emptyset$ and by rule [:*]I have $\Gamma\vdash[t:=*][?0\leq t\leq d][x:=sln]\psi$ from which goal $\Gamma\vdash[t:=*;?0\leq t\leq d;x:=sln]\psi$ follows immediately by rule $\langle\!|;|\!\rangle$I. This completes the second premise.

The third premise is $\Gamma\vdash[t:=0][\{t'=1,x'=f\,\&\,\psi\}^d](L\bmod M\leq_{[]}L)$, so by the semantics of ; it suffices to show $\Gamma\vdash[t:=0;\{t'=1,x'=f\,\&\,\psi\}^d](L\bmod M\leq_{[]}L)$. By rule bsolve and by reusing (D1) and (D2) it suffices to show $\Gamma\frac{y}{x},0\leq t=d,x=sln\frac{y}{x},x'=f\vdash(L\bmod M\leq_{[]}L)$ which is exactly the IH on (D3), completing the case.

Angel cases:

Case [∪]I: In this case, the refinement is $\{\{\alpha\cup\beta\};L\}\bmod(M,N)\equiv\{\alpha;L\}\bmod M\cup\{\beta;L\}\bmod N$. By rule ;d$_r$ and rule trans it suffices to show $\{\alpha;L\}\bmod M\cup\{\beta;L\}\bmod N\leq_{[]}\{\{\alpha;L\}\cup\{\beta;L\}\}$. By rule [∪]R it suffices to show (L) $\{\alpha;L\}\bmod M\cup\{\beta;L\}\bmod N\leq_{[]}\alpha;L$ and (R) $\{\alpha;L\}\bmod M\cup\{\beta;L\}\bmod N\leq_{[]}\beta;L$.

The IH on α gives $\Gamma\vdash\{\alpha;L\}\bmod M\leq_{[]}\alpha;L$, then rule [∪]L1 gives $\{\alpha;L\}\bmod M\cup\{\beta;L\}\bmod N\leq_{[]}\{\alpha;L\}\bmod M$, proving (L) by rule trans.

The IH on β gives $\Gamma \vdash \{\beta; L\} \mathsf{mod}\ N \leq_{[]} \alpha; L$, then rule [$\cup$]L2 gives $\{\alpha; L\} \mathsf{mod}\ M \cup \{\beta; L\} \mathsf{mod}\ N \leq_{[]} \{\beta; L\} \mathsf{mod}\ N$, proving (R) by rule trans.

Case [?]I: In this case, the refinement is $\{?\psi; L\} \mathsf{mod}\ (\lambda p : \psi.\ M) \equiv ?\psi; \{L \mathsf{mod}\ M\}$. By rule ;S on system $?\psi$ it suffices to show $\Gamma \vdash ?\psi \leq_{[]} ?\psi$ (which holds immediately by rule refl) and $\Gamma \vdash [?\psi](L \mathsf{mod}\ M \leq_{[]} L)$ which follows by rule [?]I from the IH on L, i.e., from $\Gamma, \psi \vdash L \mathsf{mod}\ M \leq_{[]} M$.

Case [:*]I: In this case, $\{x := *; L\} \mathsf{mod}\ (\lambda x : \mathbb{R}.\ M) \equiv x := *; \{L \mathsf{mod}\ M\}$. By rule ;S on system $x := *$ it suffices to show $\Gamma \vdash x := * \leq_{[]} x := *$ (which holds immediately by rule refl) and $\Gamma \vdash [x := *](L \mathsf{mod}\ M \leq_{[]} L)$ which follows by rule [:*]I from the IH on L, i.e., from $\Gamma^y_x \vdash L \mathsf{mod}\ M \leq_{[]} M$.

Case [*]R: Have $\{\alpha^*; L\} \mathsf{mod} [\mathrm{roll}\ (M, N)] \equiv L \mathsf{mod} M \cup \{\alpha; \alpha^*; L\} \mathsf{mod} N$. By rules roll_l, $;\mathrm{d}_r$, and trans it suffices to show $\Gamma \vdash L \mathsf{mod} M \cup \{\alpha; \alpha^*; L\} \mathsf{mod} N \leq_{[]} ?true \cup \{\alpha; \alpha^*\}; L$. By rule [$\cup$]R it suffices to show (L) $\{L \mathsf{mod} M\} \cup \{\{\alpha; \alpha^*; L\} \mathsf{mod} N\} \leq_{[]} L$ and (R) $\{\{L \mathsf{mod} M\} \cup \{\{\alpha; \alpha^*; L\} \mathsf{mod}\ N\}\} \leq_{[]} \{\alpha; \alpha^*; L\}$. By inversion have $\Gamma \vdash [L]\phi$ and $\Gamma \vdash [\alpha; \alpha^*; L]\phi$, so by the IH have (0) $\Gamma \vdash L \mathsf{mod} M \leq_{[]} L$ and (1) $\Gamma \vdash \{\alpha; \alpha^*; L\} \mathsf{mod} N \leq_{[]} \{\alpha; \alpha^*; L\} \mathsf{mod} N$. From (0) then rule [$\cup$]L1 gives $\{L \mathsf{mod} M\} \cup \{\{\alpha; \alpha^*; L\} \mathsf{mod} N\} \leq_{[]} L \mathsf{mod} M$, proving (L) by rule trans. From (1) then rule [$\cup$]L2 gives $\{L \mathsf{mod} M\} \cup \{\{\alpha; \alpha^*; L\} \mathsf{mod} N\} \leq_{[]} \{\alpha; \alpha^*; L\} \mathsf{mod} N$, proving (R) by rule trans.

Case [*]I: Have: $\{\alpha^{*d}; L\} \mathsf{mod}\ (A \mathrm{\ rep\ } \psi.\ B \mathsf{\ in\ } C) \equiv \{\alpha \mathsf{mod} B\}^*; \{L \mathsf{mod} C\}$.

By rule ;S on system $\{\alpha \mathsf{mod} B\}^*$, suffices to show (L) $\Gamma \vdash \{\alpha \mathsf{mod} B\}^* \leq_{[]} \alpha^*$ and (R) $\Gamma \vdash [\{\alpha \mathsf{mod} B\}^*](L \mathsf{mod} C \leq_{[]} L)$. In order to show (L), we first apply rule un* and then show its premise $\Gamma \vdash [\alpha \mathsf{mod} B](\alpha \mathsf{mod} B \leq_{[]} \alpha)$. Show this by rule [*]I with invariant ψ. The base case and inductive case are respectively by (A) and Theorem 6.3 on (B), giving $\psi \vdash [\alpha \mathsf{mod} B]\psi$. The post-case $\psi \vdash \alpha \mathsf{mod} B \leq_{[]} \alpha$ is the IH on (B). To show (R), apply the IH on (C) to get $\psi \vdash L \mathsf{mod} C \leq_{[]} L$. Then apply rule [*]I with invariant ψ. The base case is (A) and the inductive step is by Theorem 6.3 on (B), giving $\psi \vdash [\alpha \mathsf{mod} B]\psi$. The post-case $\psi \vdash L \mathsf{mod} C \leq_{[]} L$ is (C).

Case DC: Have $\{x' = f \,\&\, \psi; L\} \mathsf{mod}\ DC(M, N) \equiv \{\{x' = f \,\&\, \psi \wedge \rho; L\} \mathsf{mod}\ N\}$. Let ρ be the differential cut formula proved by M. By inversion on the the non-refinement CdGL DC proof term have $\Gamma \vdash M : [x' = f \,\&\, \psi]\rho$ so by *refinement* rule DC applied to fact (0) have $\Gamma \vdash \{x' = f \,\&\, \psi\} \cong \{x' = f \,\&\, \psi \wedge \rho\}$. By IH have $\Gamma \vdash \{x' = f \,\&\, \psi \wedge \rho; L\} \mathsf{mod}\ N \leq_{[]} \{x' = f \,\&\, \psi \wedge \rho\}; L$ so by rule trans it suffices to show $\Gamma \vdash \{x' = f \,\&\, \psi \wedge \rho; L\} \leq_{[]} \{x' = f \,\&\, \psi; L\}$ and by rule ;S on system $x' = f \,\&\, \psi \wedge \rho$ it suffices to show (L) $\Gamma \vdash \{x' = f \,\&\, \psi \wedge \rho\} \leq_{[]} \{x' = f \,\&\, \psi\}$ and (R) $\Gamma \vdash [x' = f \,\&\, \psi \wedge \rho](L \leq_{[]} L)$. Premise (L) follows from rule DC applied to fact (0). Premise (R) follows by rule refl under an application of monotonicity rule M on N.

Case DW: Have $\{x' = f \,\&\, \psi; L\} \mathsf{mod}\ DW(M) \equiv \{x := *; x' := f; ?\psi; \{L \mathsf{mod}\ M\}\}$. From DW proof have (0) $\Gamma^y_x, \psi \vdash M : [L]\phi$. By IH have (1) $\Gamma^y_x, \psi \vdash L \mathsf{mod}\ M \leq_{[]} L$. By rule [?]I have (1A) $\Gamma^y_x \vdash [?\psi](L \mathsf{mod}\ M \leq_{[]} L)$. By weakening have (1B) $\Gamma^y_x, x' = f \vdash [?\psi](L \mathsf{mod}\ M \leq_{[]} L)$. Note for fresh z have $(\Gamma^y_x)^z_x$ since x is fresh in Γ^y_x. Thus, (1C) $(\Gamma^y_x)^z_x, x' = f \vdash [?\psi](L \mathsf{mod}\ M \leq_{[]} L)$. By applying rules $\langle\![:=]\!\rangle$I and $\langle\![;]\!\rangle$I we have (1D) $\Gamma^y_x \vdash [x' := f; ?\psi](L \mathsf{mod}\ M \leq_{[]} L)$. By rules [:*]I and $\langle\![;]\!\rangle$I have (1E) $\Gamma \vdash [x := *; x' := f; ?\psi](L \mathsf{mod} M \leq_{[]} L)$. Note to prove the case it suffices by rule trans to show

(L) $\Gamma \vdash x := *; x' := f; ?\psi \leq_{[]} x' = f \,\&\, \psi$ and (R) $\Gamma \vdash [x := *; x' := f; ?\psi](L \bmod M \leq_{[]} L)$. Premise (L) is by rule DW. Premise (R) is (1D).

Case DG: In this case, the refinement is:

$$\begin{aligned} & \{x' = f \,\&\, \psi; \{y := *; y' := *\}^d; L\} \bmod DG(f_0, a, b, M) \\ \equiv\ & y := f_0; \{\{x' = f, y' = a(x)y + b(x) \,\&\, \psi; L\} \bmod M\} \end{aligned}$$

We prove the case by transitivity rule trans:

$$\begin{aligned} & y := f_0; \{\{x' = f, y' = a(x)y + b(x) \,\&\, \psi; L\} \bmod M\} && \text{(C.1)} \\ \leq_{[]}\ & y := f_0; \{x' = f, y' = a(x)y + b(x) \,\&\, \psi; L\} && \text{(C.2)} \\ \leq_{[]} & \{y := f_0; x' = f, y' = a(x)y + b(x) \,\&\, \psi\}; L && \text{(C.3)} \\ \leq_{[]} & \{x' = f \,\&\, \psi; \{y := *; y' := *\}^d\}; L && \text{(C.4)} \\ \leq_{[]}\ & x' = f \,\&\, \psi; \{y := *; y' := *\}^d; L && \text{(C.5)} \end{aligned}$$

Step (C.2) follows from rule ;S on system $y := f_0$ and the IH on M. By inversion $\Gamma, y = f_0 \vdash M : [x' = f, y' = a(x)y + b(x) \,\&\, \psi; L]\phi$ with side condition that $y \notin \mathrm{FV}(\Gamma) \cup \mathrm{FV}(f_0) \cup \mathrm{FV}(f) \cup \mathrm{FV}(a) \cup \mathrm{FV}(b) \cup \mathrm{FV}(\psi) \cup \{x\}$ (but y need not be free in ϕ), so by IH have $\Gamma, y = f_0 \vdash \{x' = f, y' = a(x)y + b(x) \,\&\, \psi; L\} \bmod M \leq_{[]} x' = f, y' = a(x)y + b(x) \,\&\, \psi; L$. The first premise of rule ;S is $\Gamma \vdash y := f_0 \leq_{[]} y := f_0$ which holds by rule refl. The second premise is

$$\Gamma \vdash [y := f_0](\{x' = f, y' = a(x)y + b(x) \,\&\, \psi; L\} \bmod M \leq_{[]} x' = f, y' = a(x)y + b(x) \,\&\, \psi; L)$$

which follows from the IH because when y is fresh in Γ and f_0, the context resulting from application of rule $\langle\!:=\!\rangle$I is $\Gamma, y = f_0$. Steps (C.3) and (C.5) hold by rule ;A. Step (C.4) holds by rule trans. The first premise $\Gamma \vdash \{y := f_0; \{x' = f, y' = a(x)y + b(x) \,\&\, \psi\}\} \leq_{[]} \{x' = f \,\&\, \psi; y := *; y' := *^d\}$ holds by rule DG. By rules $\langle\!;\!\rangle$I, $\langle\!:=\!\rangle$I, DW, and refl, the second premise $\Gamma \vdash [y := f_0; x' = f, y' = a(x)y + b(x) \,\&\, \psi](L \leq_{[]} L)$ holds.

Case bsolve: In this case the refinement is

$$\{t := 0; t' = 1, x' = f \,\&\, \psi\}; L \bmod DS(sol, M) \equiv t := 0; t' = 1, x' = f \,\&\, \psi; \{L \bmod M\}$$

The proof starts by applying rule ;S on $\{t := 0; t' = 1, x' = f \,\&\, \psi\}; L$. The first premise $\{t := 0; t' = 1, x' = f \,\&\, \psi\} \leq_{[]} \{t := 0; t' = 1, x' = f \,\&\, \psi\}$ holds by rule refl. The second premise is $\Gamma \vdash [t := 0; t' = 1, x' = f \,\&\, \psi](L \bmod M \leq_{[]} L)$. The IH gives (IH) $\Gamma\frac{y}{x}, t \geq 0, \hat{\psi}, x = sln\frac{y}{x}, x' = f \vdash (L \bmod M \leq_{[]} L)$. Then, by applying the rule bsolve, we have $\Gamma \vdash [t := 0; \{t' = 1, x' = f \,\&\, \psi\}](L \bmod M \leq_{[]} L)$, using the same solution sol. Thus, the application of rule ;S is complete, yielding:

$$\Gamma \vdash \{t := 0; t' = 1, x' = f \,\&\, \psi; L\} \leq_{[]} t := 0; t' = 1, x' = f \,\&\, \psi; \{L \bmod M\}$$

from which the conclusion follows immediately by rule ;A. □

Appendix D

Appendices to Chapter 7

D.1 Kaisar and Bellerophon Case Studies

For the sake of being self-contained, we list the Bellerophon and Kaisar proofs used in the Kaisar evaluation. These listings include Bellerophon proofs from prior work, such as the IJRR (Mitsch et al., 2017) and RA-L (Bohrer, Tan, et al., 2019) proofs. To access these proofs in a more convenient format, see the archive file which accompanies this thesis. The RA-L proof was checked in KeYmaera X 4.7, the others in 4.9.2.

Before we give the source listings of models and proofs, we repeat the line-counting rules from the thesis with added detail. We count only non-blank, non-punctuation, non-comment lines. Debugging statements which print information to the user *are* counted, specifically as proof lines. The Demonic looping operator `*`, choice operator ∪, and the duality operator ^{d} are considered punctuation. In order to avoid unfairly overcounting the proof length of our competitor Bellerophon, we do not count its variable *declarations* (e.g. `R V().` for real-valued constant `V`) because they have no counterpart in Kaisar. Because Bellerophon's auxiliary *definitions* (e.g. `B safe(R d) <-> (d>=0).`) do have a counterpart (`let`) in Kaisar, we count (all) lines from a Bellerophon definition. The total line count can be less than the sum of modeling and proof lines because the same line may contribute to both the model and proof. Assertions in Kaisar and `@invariant` annotations in Bellerophon are counted as proof but not model. Kaisar's `note` statement is treated likewise. In Kaisar, we count line labels, `let`, `switch`, and `case` as model. The (Assump) column counts all lines which contain `using` clauses in Kaisar and `hide` (weakening) steps in Bellerophon. The `hide` steps specify that a given assumption should be *ignored* by proof automation.

D.1.1 PLDI

We give the models and proofs for the Demonic Control model (PLDI-DC):

```
Bellerophon model and proof:
/* Exported from KeYmaera X v4.9.2 */
```

```
SharedDefinitions.
/* definitions shared among all lemmas and theorems */
  R V().          /* maximum velocity */
  R eps().         /* reaction time */

  B bounds() <-> ( V()>=0 & eps()>=0 ).
  B init(R d) <-> ( d>=0 & bounds() ).
  B safe(R d) <-> ( d>=0 ).
  B loopinv(R d) <-> ( d>=0 ).

  HP ctrl ::= {
    {?d>=V*eps(); v:=*; ?0<=v&v<=V; ++ v:=0;}
    t := 0;
  }.

  HP plant ::= { {d'=-v, t'=1 & t<=eps()}@invariant(d>=V()*(eps()-t))
   }.
End.

Lemma "PLDI-DC".
ProgramVariables.
/* program variables may change their value over time */
  R d.            /* distance to stop sign */
  R v.            /* velocity of the car */
  R t.            /* clock variable */
End.

Problem.          /* differential dynamic logic formula */
    init(d)
 -> [
      ?bounds();
      {
        ctrl;
        plant;
      }*@invariant(loopinv(d))
    ] safe(d)
End.

Tactic "Velocity Car: Proof 1".
  expandAllDefs; auto
End.
End.
```

```
Kaisar model, Demonic Control (PLDI-DC):
let inv() <-> (d>=v*(eps-t) & t>=0 & t<=eps & 0<=v&v<=V);
?(d >= 0 & V >= 0 & eps >= 0 & v=0 & t=0);
```

```
!(inv());
{
 {?(d >= eps*V); v:=*; ?(0<=v & v<=V); ++ v:=0;}
 {t := 0; {d' = -v, t' = 1 & ?(t <= eps)};}
 !(inv());
}*
!(d >= 0);
```

The following additional variants of the PLDI model are used in the evaluation from Section 8.5.1. Their Kaisar versions were written first, then backported to Bellerophon. We list both the Kaisar and Bellerophon versions.

```
Bellerophon model and proof, Angelic Sandbox (PLDI-AS):
/* Exported from KeYmaera X v4.9.2 */

SharedDefinitions
R V().
R eps().
B bounds() <-> ( V()>=0 & eps()>=0 ).
B init(R d, R v, R t) <-> ( d>=0 & bounds() & v=0 & t=0).
B safe(R d) <-> ( d>=0 ).
B loopinv(R d, R v, R t) <->
(d>=v*(eps()-t) & t>=0 & t<=eps() & 0<=v&v<=V()).
HP ctrl ::= {
      vCand :=*;
     { {?(d>=eps()*V() & 0<= vCand & vCand <= V());
       v:=vCand;}
      ++{v:=0;}}^@}.
HP plant ::= { t := 0; {d'=-v, t'=1 & t<=eps()} }.
End.
Lemma "PLDI-AS".
ProgramVariables.
  /* program variables may change their value over time */
  R d.            /* distance to stop sign */
  R v.            /* velocity of the car */
  R t.            /* clock variable */
End.

Problem.          /* differential dynamic logic formula */
    init(d,v,t)
 -> [ {
        ctrl;
        plant;
      }*
    ] safe(d)
End.
```

```
Tactic "PLDI-AS: Proof"
expandAllDefs ;
implyR(1); loop("d>=v*(eps()-t)&t>=0&t<=eps()&0<=v&v<=V()", 1) ; <(
  auto,
  composeb(1) ; composeb(1) ; randomb(1) ; allR(1) ; dualDirectb(1) ;
  choiced(1) ; cut("d>=eps()*V()&0<=vCand&vCand<=V()|true") ; <(
    orR(1) ; orL(-2) ; <(
      hideR(2=="<v:=0;>[t:=0;{d'=-v,t'=1&t<=eps()}]
           (d>=v*(eps()-t)&t>=0&t<=eps()&0<=v&v<=V())") ;
      composed(1) ; testd(1) ; andR(1) ; <(
        propClose,
        assignd(1) ; composeb(1) ; assignb(1) ; dC("t>=0", 1) ; <(
          dC("d>=vCand*(eps()-t)", 1) ; <(
            dW(1) ; auto,
            dI(1)),
          dI(1))),
      hideR(1=="<?d>=eps()*V()&0<=vCand&vCand<=V();v:=vCand;>
            [t:=0;{d'=-v,t'=1&t<=eps()}]
            (d>=v*(eps()-t)&t>=0&t<=eps()&0<=v&v<=V())") ;
      assignd(1) ; composeb(1) ; assignb(1) ; dC("t>=0", 1) ; <(
        dC("d>=0*(eps()-t)", 1) ; <(
          dW(1) ; auto,
          dI(1)),dI(1))),
    hideR(1=="<?d>=eps()*V()&0<=vCand&vCand<=V();v:=vCand;>[t:=0;
      {d'=-v,t'=1&t<=eps()}](d>=v*(eps()-t)&t>=0&t<=eps()&0<=v&v<=V())
      |<v:=0;>[t:=0;{d'=-v,t'=1&t<=eps()}]
        (d>=v*(eps()-t)&t>=0&t<=eps()&0<=v&v<=V())") ;
  propClose),auto)
End.
End.
```

```
Kaisar model, Angelic Sandbox (PLDI-AS):
let inv() <-> (d>=v*(eps-t) & t>=0 & t<=eps & 0<=v&v<=V);
?(d >= 0 & V >= 0 & eps >= 0 & v=0 & t=0);
!(inv());
{
 vCand :=*;
 switch {
   case (d>=eps*V & 0 <=vCand & vCand <= V) => v:=vCand;
   case (true) => v:=0;
 }
 {t := 0; {d' = -v, t' = 1 & ?(t <= eps); & !(d >= v*(eps-t));};}
 !(inv());
}*
!(d >= 0);
```

```
Bellerophon model and proof, Timed Angelic Control (PLDI-TAC):
/* Exported from KeYmaera X v4.9.2 */

SharedDefinitions
R V().
R eps().
B bounds() <-> ( V()> 0 & eps()>0 ).
B init(R d, R v, R t) <-> ( d>=0 & bounds() & v=0 & t=0 ).
B safe(R d) <-> ( d>=0 ).
B loopinv(R d, R v, R t) <-> (d>=0 & t>=0 & t<=eps() & 0<=v&v<=V()).
HP ctrl ::= {
    {  {?d>=V()*eps(); v:=V; {?0<=v&v<=V;}^@ }
    ++ {v:=0;}}^@
  }.
HP plant ::= { t := 0; {d'=-v, t'=1 & t<=eps()} }.
End.
Lemma "PLDI-TAC".
ProgramVariables.
/* program variables may change their value over time */
  R d.            /* distance to stop sign */
  R v.            /* velocity of the car */
  R t.            /* clock variable */
  R pos.
  R time.
End.

Problem.          /* differential dynamic logic formula */
    init(d, v, t)
 -> [ time := 0;
      {
       {
        {?(time<= 10000);
         ctrl;
         plant;
         time := time + 600;
        }^@
       }*
      }^@
    ] safe(d)
End.

Tactic "PLDI-TAC: Proof"
expandAllDefs ; unfold ;
con("k", "(d>=0&t>=0&t<=eps()&0<=v&v<=V())&10000-time<=k*600", 1) ; <(
  auto,
  auto,
```

```
  dualDirectd(1) ; composeb(1) ; testb(1) ; implyR(1) ; composeb(1) ;
  dualDirectb(1) ; choiced(1) ; orR(1) ; cut("d>=V()*eps()|true") ; <(
    orL(-4) ; <(
      hideR(2=="<v:=0;>[{t:=0;{d'=-v,t'=1&t<=eps()}}time:=time+600;]
        ((d>=0&t>=0&t<=eps()&0<=v&v<=V())&10000-time<=(k-1)*600)") ;
  unfold ; dC("t>=0", 1) ; <(
        dC("d>=V()*(eps()-t)", 1) ; <(
          dW(1) ; auto,
          dI(1)),
        dI(1)),
      hideR(1=="<?d>=V()*eps();v:=V();{?0<=v&v<=V();}^@>
        [{t:=0;{d'=-v,t'=1&t<=eps()}}time:=time+600;]
          ((d>=0&t>=0&t<=eps()&0<=v&v<=V())&10000-time<=(k-1)*600)") ;
    assignd(1) ; composeb(1) ; composeb(1) ; assignb(1) ;
    dC("t>=0", 1) ; <(
        dC("d>=0*(eps()-t)", 1) ; <(
          dW(1) ; auto,
          dI(1)),
        dI(1))),
    propClose))
End.
End.
```

```
Kaisar model, Timed Angelic Control (PLDI-TAC):
let inv() <-> (d>=v*(eps-t) & t>=0 & t<=eps & 0<=v&v<=V);
?(d >= 0 & V >= 0 & eps >= 0 & v=0 & t=0);
for (time := 0; !(inv()); ?(time <= 10000); time := (time + 600);) {
 switch {
   case (d>=eps*V) => v:=V; ?(0<=v&v<=V);
   case (true) => v:=0;
 }
 {t := 0; {d' = -v, t' = 1 & ?(t <= eps); & !(d >= v*(eps-t));};}
 !(inv());
}
!(d >= 0);
```

```
Bellerophon model and proof, Reach-avoid (PLDI-RA):
/* Exported from KeYmaera X v4.9.2 */

SharedDefinitions
R V().
R eps().
R liveFactor() = (4) .
R liveIncr() = ( V()*eps()/(liveFactor()^2)).
B bounds() <-> ( V()> 0 & eps()>0 ).
B init(R d, R v, R t) <-> ( d>=0 & bounds() & v=0 & t=0 ).
```

```
B safe(R d) <-> ( d>=0 ).
B live(R d) <-> ( d<=(V()+1)*eps() ).
HP ctrl ::= {
    {{?d>=V()*eps(); v:=V; {?0<=v&v<=V;}^@}}^@
}.
HP plant ::= { t := 0; {d'=-v, t'=1 & t<=eps()};
               {?(t>=eps()/liveFactor());}}.
End.
Lemma "PLDI RA".
ProgramVariables.
  /* program variables may change their value over time */
  R d.            /* distance to stop sign */
  R v.            /* velocity of the car */
  R t.            /* clock variable */
  R pos.
  R dInit.
End.

Problem.          /* differential dynamic logic formula */
    init(d, v, t)
 -> [ dInit := d;
      pos := 0;
      {?(d>=0);}^@
      {
       {
        {?(pos<= dInit & d >= V()*eps());
         ctrl;
         plant;
         pos := pos + V()*eps()/liveFactor();
        }^@
       }*
      }^@
    ] (safe(d) & live(d))
End.

Tactic "PLDI RA: Proof"
expandAllDefs ; unfold ;
con("k", "(d>=0&t>=0&t<=eps()&0<=v&v<=V())&d<=k*liveIncr()", 1) ; <(
  expandAllDefs ; auto,
  expandAllDefs ; auto,
  expandAllDefs ; unfold ; dC("d>=V()*(eps()-t)", 1) ; <(
    dC("d<=old(d)-V()*t/liveFactor()", 1) ; <(
      dW(1) ; expandAllDefs ; auto,
      expandAllDefs ; ODE(1)
      ),
    ODE(1)
```

```
    )
  )
End.
End.
```

```
Kaisar model, Reach-avoid (PLDI-RA):
let inv(pos) <-> (d >= 0 & (d@init - d) >= pos);
let liveFactor() = 4;
let liveIncr() = V*eps/(liveFactor()*liveFactor());
?(d >= 0 & V > 0 & eps > 0 & v=0 & t=0);
init:
for (pos := 0; !conv:(d >= 0 & (d@init - d) >= pos);
 ?guard:(pos <= d@init & d >= V*eps); pos := pos + liveIncr()) {
body:
   v:=V;
   {t:=0; { d'=-v, t'=1 & ?(t <= eps) & !(d >= v*(eps-t))
    & !(d <= d@body - v*t/liveFactor())};}
 ?(t >= eps/liveFactor());
 !(inv(pos + liveIncr()));
}
!(pos >= d@init - eps | d <= V*eps + eps) by guard(eps);
!(d >= 0 & d <= (V+1)*eps);
```

```
Bellerophon model and proof, Reach-avoid with disturbance (PLDI-RAD):
/* Exported from KeYmaera X v4.9.2 */

SharedDefinitions
R V().
R eps().
R liveFactor() = (4) .
R liveIncr() = ( V()*eps()/(4*liveFactor()^2)).
B bounds() <-> ( V()> 0 & eps()>0 ).
B init(R d, R v, R t) <-> ( d>=0 & bounds() & v=0 & t=0 ).
B safe(R d) <-> ( d>=0 ).
B live(R d) <-> ( d<=(V()+1)*eps() ).
HP ctrl ::= {
    {{?d>=V()*eps(); v:=V; {?0<=v&v<=V;}^@
    {vd:=*;?(0.5<=vd&vd<=1); v:=v*vd;}^@
    }}^@
}.
HP plant ::= { t := 0; {d'=-v, t'=1 & t<=eps()};
               {?(t>=eps()/liveFactor());}}.
End.
Lemma "PLDI RAD".
ProgramVariables.
  /* program variables may change their value over time */
```

```
  R d.            /* distance to stop sign */
  R v.            /* velocity of the car */
  R t.            /* clock variable */
  R pos.
  R dInit.
End.

Problem.          /* differential dynamic logic formula */
    init(d, v, t)
 -> [ dInit := d;
      pos := 0;
      {?(d>=0);}^@
      {
       {
        {?(pos<= dInit & d >= V()*eps());
         ctrl;
         plant;
         pos := pos + V()*eps()/liveFactor();
        }^@
       }*
      }^@
    ] (safe(d) & live(d))
End.

Tactic "PLDI RAD: Proof"
expandAllDefs ; unfold ;
con("k","(d>=0&t>=0&t<=eps()&0<=v&v<=V())&d<=k*liveIncr()", 1); <(
  expandAllDefs ; auto,
  expandAllDefs ; auto,
  expandAllDefs ; unfold ; dC("d>=V()*(eps()-t)", 1) ; <(
    dC("d<=old(d)-V()*t/liveFactor()", 1) ; <(
      dW(1) ; expandAllDefs ; auto,
      expandAllDefs ; ODE(1)
      ),
    ODE(1)
    )
  )
End.

End.
```

```
Kaisar model, Reach-avoid with disturbance (PLDI-RAD):
let inv(pos) <-> (d >= 0 & (d@init - d) >= pos);
let liveFactor() = 4;
let liveIncr() = V*eps/(4*liveFactor()*liveFactor());
?(d >= 0 & V > 0 & eps > 0 & v=0 & t=0);
```

```
init:
for (pos := 0; !conv:(d >= 0 & (d@init - d) >= pos);
 ?guard:(pos <= d@init & d >= V*eps); pos := pos + liveIncr()) {
body:
   v:=V;
   c:=*; ?dstrb:(0.5 <= c & c <= 1.0); ?vd:(v:=v*c);
   {t:=0; { d'=-v, t'=1 & ?(t <= eps) & !(d >= v*(eps-t))
     using dstrb ... by auto;
    & !(d <= d@body - v*t/liveFactor()) using dstrb ... by auto};}
 ?(t >= eps/liveFactor());
 !(inv(pos + liveIncr())) using dstrb vd ... by auto;
}
!(pos >= d@init - eps | d <= V*eps + eps) by guard(eps);
!(d >= 0 & d <= (V+1)*eps);
```

D.1.2 IJRR: Theorem 1

In the Bellerophon proof of IJRR, the syntax `tactic <name> as <tactic>` is used to define a named, reusable piece of proof script. If every use of the reusable proof were fully expanded, the counts for Bellerophon would be higher.

```
Bellerophon model and proof, IJRR:
/* Exported from KeYmaera X v4.9.2 */

Theorem "IJRR-1".

Definitions
  Real eps;      /* time limit for control decisions */
  Real b;       /* minimum braking capability of the robot */
  Real A;       /* maximum acceleration -b <= a <= A */
  Real W;       /* maximum steering */

  /* The straight-line stopping distance
      from brake start to full stop. */
  Real stopDist(Real v) = v^2 / (2*b);
 /* Straight-line distance to compensate acceleration */
  Real accelComp(Real v) = ((A/b + 1) * (A/2 * eps()^2 + eps()*v));
  /* Separation that allows accelerating on a new curve */
  Real admissibleSeparation(Real v) = stopDist(v) + accelComp(v);

  /* The orientation of the robot is a unit vector. */
  Bool isWellformedDir(Real dx, Real dy) <-> dx^2 + dy^2 = 1;

  Bool bounds() <-> (  /* Bounds for global constants */
      A >= 0                  /* Working engine */
    & b > 0                    /* Working brakes */
```

```
    & eps() > 0                    /* Controller reaction time */
  );
  Bool initialState /* Stopped safe initially */
  (Real x, Real y, Real v, Real dx, Real dy, Real xo, Real yo)
  <-> (v = 0
      & (x-xo)^2 + (y-yo)^2 > 0
      & isWellformedDir(dx, dy)
  );

  /* Under these assumptions we guarantee safety */
  Bool assumptions
 (Real x, Real y, Real v, Real dx, Real dy, Real xo, Real yo)
   <-> bounds() & initialState(x, y, v, dx, dy, xo, yo);

  /* Conditions that are true on each loop iteration */
  Bool loopinv
  (Real x, Real y, Real v, Real dx, Real dy, Real xo, Real yo)
  <-> (v >= 0
      & isWellformedDir(dx, dy)
      & (abs(x-xo) > stopDist(v) | abs(y-yo) > stopDist(v))
  );
End.

ProgramVariables
  Real x;     /* robot position: x */
  Real y;     /* robot position: y */
  Real v;     /* robot translational velocity */
  Real a;     /* robot translational acceleration */
  Real dx;    /* robot orientation: x */
  Real dy;    /* robot orientation: y */
  Real w;     /* robot rotational velocity */
  Real r;     /* robot curve radius */
  Real xo;    /* position of closest obstacle on curve */
  Real yo;
  Real t;     /* time */
End.

Problem
assumptions(x, y, v, dx, dy, xo, yo) ->
  [
    {
      {
        {
          /* brake on current curve or remain stopped */
          { a := -b; }
          ++
```

```
          { ?v = 0; a := 0; w := 0; }
          ++
          /* or choose a new safe curve */
          { a := A;
            w := *; ?-W<=w & w<=W; /* choose steering */
            r := *;
            /* measure closest obstacle on the curve */
            xo := *; yo := *;

            /* admissible curve */
            ?r!=0 & r*w = v;

            /* use that curve, if it is a safe one
               (admissible velocities) */
            ? abs(x-xo) > admissibleSeparation(v)
            | abs(y-yo) > admissibleSeparation(v);
          }
        };
        t := 0;
      }

      /* dynamics */
      {
      /* accelerate/decelerate and move */
      { x' = v * dx, y' = v * dy, v' = a,
        /* follow curve */
        dx' = -w * dy, dy' = w * dx, w' = a/r,
        t' = 1 & t <= eps() & v >= 0
      }
      }
    }*
  ](x - xo)^2 + (y - yo)^2 > 0
End.

Tactic "Proof Theorem 1: Static safety"
  tactic diall as (
    diffInvariant("t>=0", 1);
    diffInvariant("isWellformedDir(dx, dy)", 1)
  );

  tactic dib as (
    diall;
    diffInvariant("v = old(v) - b*t", 1);
    diffInvariant("-t * (v + b/2*t) <= x - old(x)
      & x - old(x) <= t * (v + b/2*t)", 1);
    diffInvariant("-t * (v + b/2*t) <= y - old(y)
```

```
    & y - old(y) <= t * (v + b/2*t)", 1)
);

tactic di0 as (
  diall;
  diffInvariant("v = old(v)", 1);
  diffInvariant("x = old(x)", 1);
  diffInvariant("y = old(y)", 1)
);

tactic dia as (
  diall;
  diffInvariant("v = old(v) + A*t", 1);
  diffInvariant("-t * (v - A/2*t) <= x - old(x)
    & x - old(x) <= t * (v - A/2*t)", 1);
  diffInvariant("-t * (v - A/2*t) <= y - old(y)
    & y - old(y) <= t * (v - A/2*t)", 1)
);

tactic dw as (andL('L)*; dW(1));

tactic xAccArith as (
  andL('L)*;
  print("Transforming...");
  transform("abs(x_0-xo)>v_0^2/(2*b)+(A/b+1)*(A/2*t^2+t*v_0)",
     'L=="abs(x_0-xo)>admissibleSeparation(v_0)");
  hideR('R=="abs(y-yo)>stopDist(v)");
  smartQE;
  print("Proved acc arithmetic")
);

tactic yAccArith as (
  andL('L)*;
  print("Transforming...");
  transform("abs(y_0-yo)>v_0^2/(2*b)+(A/b+1)*(A/2*t^2+t*v_0)",
    'L=="abs(y_0-yo)>admissibleSeparation(v_0)");
  hideR('R=="abs(x-xo)>stopDist(v)");
  smartQE;
  print("Proved acc arithmetic")
);

implyR(1); andL('L)*;
loop("loopinv(x, y, v, dx, dy, xo, yo)", 1); <(
  print("Base case..."); smartQE; print("Base case done")
  ,
  print("Use case..."); smartQE; print("Use case done")
```

```
    ,
    print("Induction step"); unfold; <(
      print("Braking branch"); dib; dw; prop; doall(smartQE);
      print("Braking branch done")
      ,
      print("Stopped branch"); di0; dw; prop; doall(smartQE);
      print("Stopped branch done")
      ,
      print("Acceleration branch");
      hideL('L == "abs(x-xo_0)>stopDist(v)
                  |abs(y-yo_0)>stopDist(v)");
      dia; dw;
      prop; <(
        xAccArith,
        yAccArith
      );
      print("Acceleration branch done")
    );
    print("Induction step done")
  );
  done;
  print("Proof done")
End.
End.
```

In the Kaisar version of the IJRR model, the `pragma` statements are implementation-specific commons which tell the Kaisar checker to report timing and debugging information throughout the proofchecking process.

```
Kaisar model, IJRR:
pragma option "time=true";
pragma option "trace=true";
pragma option "debugArith=true";
let stopDist(v) = (v^2 / (2*b));
let accelComp(v,a) = ((a/b + 1) * (a/2 * T^2 + T*v));
let admissibleSeparation(v,a) = (stopDist(v) + accelComp(v,a));
let bounds() <-> A >= 0 & b > 0 & T > 0;
let norm(x, y) = (x^2 + y^2)^(1/2);
let dist(xl, yl, xr, yr) = norm (xl - xr, yl - yr);
let initialState() <->
  (v = 0 & dist(x,y,xo,yo) > 1 & norm(dx, dy) = 1);
let infdistGr(x1, y1, x2, y2, z) <->
  (x1-x2 > z | x2 - x1 > z | y1 - y2 > z | y2 - y1 > z);
/* Disjuncts of infdistGr predicate, useful in case analysis */
let goal() <-> (dist(x,y,xo,yo) > 0);
?(bnds, st):(bounds() & initialState());
/* Prove infdist > 0 in base case by case-analyzing x,y and
```

```
   using assumption on euclidean distance > 1*/
!splitX:(x-xo >= 0.25 | x-xo <= 0.35) by exhaustion;
!splitXO:(xo-x >= 0.25 | xo-x <= 0.35) by exhaustion;
!splitY:(y-yo >= 0.25 | y-yo <= 0.35) by exhaustion;
!sd:(infdistGr(x, y, xo, yo, stopDist(v)))
  using splitX splitXO splitY bnds st by auto;
/* Base case invariant */
!(vSign, dxyNorm, safeDist):
  (  v >= 0
   & norm(dx, dy) = 1
   & infdistGr(x, y, xo, yo, stopDist(v)))
   using bnds st sd by auto;
{body:
  {
    {
     let monCond() <->
       ( (a = -b
       | (a = 0 & v = 0)) & infdistGr(x, y, xo, yo,  stopDist(v))
       | a = A & infdistGr(x, y, xo, yo,  admissibleSeparation(v,A)));
      { {  {?aB:(a := -b);
           xo := xo; yo := yo;
           !admiss:(monCond())
             using safeDist vSign bnds aB by auto; }
        ++ {?vZ:(v = 0); ?aZ:(a := 0); w := 0;
           xo := xo; yo := yo;
           !admiss:(monCond())
             using safeDist vSign bnds aZ vZ by auto; }
        ++ {?aA:(a := A); w := *; ?(-W<=w & w<=W);
            r := *;
            xo := *; yo := *;
            ?(r!=0 & r*w = v);
            ?admiss:(monCond());}
        }
  monitor:
        ?tZ:(t := 0);
        { x' = v * dx, y' = v * dy, v' = a,
          dx' = -w * dy, dy' = w * dx, w' = a/r,
          t' = 1 & ?dom:(t <= T & v >= 0)
         & !tSign:(t >= 0) using tZ by induction
         & !dir:(norm(dx, dy) =  1) using dxyNorm by induction
         & !vSol:(v = v@body + a*t) using tZ by induction
         & !xBound:(-t * (v@body + a/2*t) <= x - x@body
             & x - x@body <= t * (v@body + a/2*t))
             using bnds vSol dir tSign dom tZ by induction
         & !yBound:(-t * (v@body + a/2*t) <= y - y@body
            & y - y@body <= t * (v@body + a/2*t))
```

```
            using bnds vSol dir tSign dom tZ by induction
        };
        !infInd:(infdistGr(x,y,xo,yo, stopDist(v)))
        using admiss xBound yBound vSol dom bnds tSign by auto;
      }
    }
  }
  !vInv: (v >= 0) using dom by auto;
  note indStep = andI(vInv, andI(dir, infInd));
}*
!(goal()) using safeDist bnds by auto;
```

D.1.3 RA-L: Theorem 1

Due to the length and complexity of the following model and proof, the reader is encouraged to consult the archive file that accompanies this thesis rather than consulting the present section of the appendix. This model and proof are included in this appendix solely for the sake of making the thesis self-contained. For example, the line breaks are sometimes difficult to read because line breaks were chosen to maximize line length and thus provide consistent line counts for the sake of the evaluation. In practical use, longer lines with more natural line breaks would be used for readability. Not only is the version in the archive file most likely easier to read, but it checks successfully (in KeYmaera X 4.7, for example) while the version presented here does not parse due to widespread use of newlines inside the arguments of tactics. If we were to print a version of the tactic that parses successfully in this appendix, it would not fit on the page.

```
Bellerophon model and proof, RA-L:
SharedDefinitions
Real T;
Real eps;
Real A;
Real B;
Real sq (Real x) = x*x;
Bool onUpperHalfPlane(Real x, Real y) <-> y > 0;
Real circle(Real x, Real y, Real k) = k*(sq(x)+sq(y)) - 2*x;
Bool onCircle(Real x, Real y, Real k) <-> circle(x,y,k) = 0;
Bool isAnnulus(Real k) <-> (
    abs(k)*eps() <= 1
  );
Bool onAnnulus(Real x, Real y, Real k) <-> (
                      (k*sq(eps()) - 2*eps()) < circle(x,y,k)
    & circle(x,y,k) < (k*sq(eps()) + 2*eps())
  );
Bool controllableSpeedGoal(Real vl, Real vh) <-> (
    0 <= vl & vl < vh &
    A * T <=  (vh - vl) &
```

```
    B * T <=  (vh - vl)
  );
Bool controllableGoalDist(Real v1, Real v2, Real a, Real xg, Real yg,
   Real k)
<->(v1 <= v2 |
   (1+2*eps()*abs(k)+sq(eps())*sq(k))*(sq(v1)-sq(v2)) <= (2*a)*(yg-eps
   ()) |
   (1+2*eps()*abs(k)+sq(eps())*sq(k))*(sq(v1)-sq(v2)) <= (2*a)*(abs(xg
   )-eps())
  );
Bool J(Real xg, Real yg, Real k, Real v, Real vl, Real vh) <-> (
    v >= 0 &
    isAnnulus(k) &
    onAnnulus(xg,yg,k) &
    controllableSpeedGoal(vl,vh) &
    controllableGoalDist(v,vh,B(),xg,yg,k) &
    controllableGoalDist(vl,v,A(),xg,yg,k)
  );
Bool admissibleAcc(Real xg, Real yg, Real v, Real vl, Real vh, Real a,
    Real k)
<-> (/* (1) Chosen acceleration is in range */
    (-B() <= a & a <= A())
    &
    /* (2) Chosen (braking) accleration does not brake to a stop too
   early */
    v + a*T() >= 0
    &
    /* (3) Chosen acceleration is safe for velocity upper bound */
    (
       v <= vh & v+a*T() <= vh
     | (
         (1 + 2*eps()*abs(k) + sq(eps())*sq(k)) * (B()*(2*v*T()+a*sq(T
   ()))
    + (sq(v+a*T()) - sq(vh))) <= (2*B()) * (abs(xg)-eps())
       | (1 + 2*eps()*abs(k) + sq(eps())*sq(k)) * (B()*(2*v*T()+a*sq(T
   ()))
          + (sq(v+a*T()) - sq(vh))) <= (2*B()) * (yg-eps())
       )
    )
    &
    /* (4) Chosen acceleration is safe for velocity lower bound */
    (
       vl <= v & v+a*T() >= vl
     | (
         (1 + 2*eps()*abs(k) + sq(eps())*sq(k)) * (A()*(2*v*T()+a*sq(T
   ()))
```

```
    + (sq(vl) - sq(v+a*T()))) <= (2*A()) * (abs(xg)-eps())
       | (1 + 2*eps()*abs(k) + sq(eps())*sq(k)) * (A()*(2*v*T()+a*sq(T
   ()))
          + (sq(vl) - sq(v+a*T()))) <= (2*A()) * (yg-eps())
       )
    )
  );
End.

Lemma "RALLemma"

ProgramVariables
  Real xg,yg; /* goal position: (dm,dm) */
  Real k;     /* curvature:     centi-(m^-1) */

  Real vl,vh; /* goal velocities (upper and lower): dm/s */

  Real t;     /* time trigger:  ds */
  Real v,a;   /* linear velocity/acceleration:  dm/s,  dm/s^2 */
End.

Problem
  T > 0 &
  eps() > 0 &
  A > 0 &
  B > 0 &
  J(xg, yg, k, v, vl, vh)
  ->
  [{
  /* Control */
  {
    {
      {
        /* Turning from left or right quadrant towards origin
     (k <= 0, xg <= 0) or (k >= 0, xg >= 0) */
        xg:=*; yg:=*; vl:=*; vh:=*; k:=*; a:=*;
        ? xg >= 0 & k >= 0 | xg <= 0 & k <= 0;

        /* Standard sanity conditions for the planner inputs:
           1) Goal (xg,yg) is in upper half plane
           2) Curvature k and bloating eps define an annulus
           3) Goal is within safe annular region
           4) Target speed limit interval (vl,vh) is wide enough
              (and >= 0)
        */
```

```
        ? onUpperHalfPlane(xg,yg) &
          isAnnulus(k) &
          onAnnulus(xg,yg,k) &
          controllableSpeedGoal(vl,vh);

        ? admissibleAcc(xg, yg, v, vl, vh, a, k);
      }
    }
    t:=0;
  }
  /* Plant */
  {
    { xg'=-v*k*yg, yg'=v*(k*xg -1), v'=a, t'=1 & (v >= 0 & t <= T) }
  }
  }*
  ]J(xg, yg, k, v, vl, vh)
End.
Tactic "RALLemma"
unfold ; loop({`v>=0&abs(k)*eps()<=1&(k*(eps()*eps())-2*eps() < k*(xg
*xg+yg*yg)-2*xg&k*(xg*xg+yg*yg)-2*xg < k*(eps()*eps())+2*eps())&(0<=
vl&vl < vh&A()*T()<=vh-vl&B()*T()<=vh-vl)&(v<=vh|(1+2*eps()*abs(k)+eps
()*eps()*(k*k))*(v*v-vh*vh)<=2*B()*(yg-eps())|(1+2*eps()*abs(k)+eps()*
eps()*(k*k))*(v*v-vh*vh)<=2*B()*(abs(xg)-eps()))&(vl<=v|(1+2*eps()*
abs(k)+eps()*eps()*(k*k))*(vl*vl-v*v)<=2*A()*(yg-eps())|(1+2*eps()*
abs(k)+eps()*eps()*(k*k))*(vl*vl-v*v)<=2*A()*(abs(xg)-eps()))`}, 1) ;
<(prop,
  prop,
  unfold ; dC({`t>=0&k*(eps()*eps())-2*eps() < k*(xg*xg+yg*yg)-2*xg&k*
  (xg*xg+yg*yg)-2*xg < k*(eps()*eps())+2*eps()`}, 1) ; <(
    dC({`v+a*(T()-t)>=0`}, 1) ; <(
      dC({`((a>=0&v+a*(T()-t)<=vh|a<=0&v<=vh)|(1+2*eps()*abs(k)+eps()*
      eps()*(k*k))*(B()*(2*v*(T()-t)+a*((T()-t)*(T()-t)))+((v+a*(T()-
      t))*(v+a*(T()-t))-vh*vh))<=2*B()*(yg-eps())|(1+2*eps()*abs(k)+
      eps()*eps()*(k*k))*(B()*(2*v*(T()-t)+a*((T()-t)*(T()-t)))+((v+a*
      (T()-t))*(v+a*(T()-t))-vh*vh))<=2*B()*(abs(xg)-eps()))&((a>=0&v>
      =vl|a<=0&v+a*(T()-t)>=vl)|(1+2*eps()*abs(k)+eps()*eps()*(k*k))*(
      A()*(2*v*(T()-t)+a*((T()-t)*(T()-t)))+(vl*vl-(v+a*(T()-t))*(v+a*
      (T()-t))))<=2*A()*(yg-eps())|(1+2*eps()*abs(k)+eps()*eps()*(k*k)
      )*(A()*(2*v*(T()-t)+a*((T()-t)*(T()-t)))+(vl*vl-(v+a*(T()-t))*(v
      +a*(T()-t))))<=2*A()*(abs(xg)-eps()))`}, 1) ; <(
        dW(1) ; hideL(-12=={`k*(eps()*eps())-2*eps() < k*(xg_0*xg_0+
  yg_0*yg_0)-2*xg_0`}) ; hideL(-12=={`k*(xg_0*xg_0+yg_0*yg_0)-2*
  xg_0 < k*(eps()*eps())+2*eps()`}) ; hideL(-13=={`v_0<=vh&v_0+a
  *T()<=vh|(1+2*eps()*abs(k)+eps()*eps()*(k*k))*(B()*(2*v_0*T()+
  a*(T()*T()))+((v_0+a*T())*(v_0+a*T())-vh*vh))<=2*B()*(abs(xg_0
  )-eps())|(1+2*eps()*abs(k)+eps()*eps()*(k*k))*(B()*(2*v_0*T()+
```

```
a*(T()*T()))+((v_0+a*T())*(v_0+a*T())-vh*vh))<=2*B()*(yg_0-
eps())`}) ; hideL(-13=={`vl<=v_0&v_0+a*T()>=vl|(1+2*eps()*abs(
k)+eps()*eps()*(k*k))*(A()*(2*v_0*T()+a*(T()*T()))+(vl*vl-(v_0
+a*T())*(v_0+a*T()))))<=2*A()*(abs(xg_0)-eps())|(1+2*eps()*abs(
k)+eps()*eps()*(k*k))*(A()*(2*v_0*T()+a*(T()*T()))+(vl*vl-(v_0
+a*T())*(v_0+a*T()))))<=2*A()*(yg_0-eps())`}) ; hideL(-17=={`
abs(k_0)*eps()<=1`}) ; hideL(-17=={`v_0<=vh_0|(1+2*eps()*abs(
k_0)+eps()*eps()*(k_0*k_0))*(v_0*v_0-vh_0*vh_0)<=2*B()*(yg_1-
eps())|(1+2*eps()*abs(k_0)+eps()*eps()*(k_0*k_0))*(v_0*v_0-
vh_0*vh_0)<=2*B()*(abs(xg_1)-eps())`}) ; hideL(-17=={`vl_0<=
v_0|(1+2*eps()*abs(k_0)+eps()*eps()*(k_0*k_0))*(vl_0*vl_0-v_0*
v_0)<=2*A()*(yg_1-eps())|(1+2*eps()*abs(k_0)+eps()*eps()*(k_0*
k_0))*(vl_0*vl_0-v_0*v_0)<=2*A()*(abs(xg_1)-eps())`}) ; hideL(
-17=={`0<=vl_0`}) ; hideL(-17=={`vl_0 < vh_0`}) ; hideL(-17==
{`A()*T()<=vh_0-vl_0`}) ; hideL(-17=={`B()*T()<=vh_0-vl_0`}) ;
hideL(-17=={`k_0*(eps()*eps())-2*eps() < k_0*(xg_1*xg_1+yg_1*
yg_1)-2*xg_1`}) ; hideL(-17=={`k_0*(xg_1*xg_1+yg_1*yg_1)-2*
xg_1 < k_0*(eps()*eps())+2*eps()`}) ; unfold ; andR(1) ; <(
        QE,
        andR(1) ; <(
          prop,
          andR(1) ; <(
            prop,
            andR(1) ; <(
              prop,
              hideL(-7=={`abs(k)*eps()<=1`}) ; hideL(-5=={`xg_0>=0&k
  >=0|xg_0<=0&k<=0`}) ; hideL(-5=={`yg_0>0`}) ; hideL(-7
  =={`A()*T()<=vh-vl`}) ; hideL(-7=={`B()*T()<=vh-vl`});
  hideL(-7=={`v_0+a*T()>=0`}) ;
  hideL(-15=={`k*(eps()*eps())-2*eps() <
  k*(xg*xg+yg*yg)-2*xg`}) ; hideL(-15=={`k*(xg*xg+yg*yg)
  -2*xg < k*(eps()*eps())+2*eps()`}) ; andR(1) ; <(
                hideL(-12=={`(a>=0&v>=vl|a<=0&v+a*(T()-t)>=vl)|(1+2*
    eps()*abs(k)+eps()*eps()*(k*k))*(A()*(2*v*(T()-t)+a*
    ((T()-t)*(T()-t)))+(vl*vl-(v+a*(T()-t))*(v+a*(T()-t)
    )))<=2*A()*(yg-eps())|(1+2*eps()*abs(k)+eps()*eps()*
    (k*k))*(A()*(2*v*(T()-t)+a*((T()-t)*(T()-t)))+(vl*vl
    -(v+a*(T()-t))*(v+a*(T()-t))))<=2*A()*(abs(xg)-eps()
    )`}) ; orL(-11) ; <(
                  QE,
                  orL(-11) ; <(
                    cut({`(1+2*eps()*abs(k)+eps()*eps()*(k*k))*((v*v
        -vh*vh)/(2*B()))<=yg-eps()`}) ; <(
                      hideL(-11=={`(1+2*eps()*abs(k)+eps()*eps()*(k*
    k))*(B()*(2*v*(T()-t)+a*((T()-t)*(T()-t)))+((v
    +a*(T()-t))*(v+a*(T()-t))-vh*vh))<=2*B()*(yg-
```

```
    eps())`}) ; QE,
                        hideR(1==={`v<=vh|(1+2*eps()*abs(k)+eps()*eps()
    *(k*k))*(v*v-vh*vh)<=2*B()*(yg-eps())|(1+2*
    eps()*abs(k)+eps()*eps()*(k*k))*(v*v-vh*vh)<=2
    *B()*(abs(xg)-eps())`}) ; QE),
                    cut({`(1+2*eps()*abs(k)+eps()*eps()*(k*k))*((v*v
        -vh*vh)/(2*B()))<=abs(xg)-eps()`}) ; <(
                        hideL(-11==={`(1+2*eps()*abs(k)+eps()*eps()*(k*
    k))*(B()*(2*v*(T()-t)+a*((T()-t)*(T()-t)))+((v
    +a*(T()-t))*(v+a*(T()-t))-vh*vh))<=2*B()*(abs(
    xg)-eps())`}) ; QE,
                        hideR(1==={`v<=vh|(1+2*eps()*abs(k)+eps()*eps()
    *(k*k))*(v*v-vh*vh)<=2*B()*(yg-eps())|(1+2*
    eps()*abs(k)+eps()*eps()*(k*k))*(v*v-vh*vh)<=2
    *B()*(abs(xg)-eps())`}) ; QE))),
                hideL(-11==={`(a>=0&v+a*(T()-t)<=vh|a<=0&v<=vh)|(1+2*
    eps()*abs(k)+eps()*eps()*(k*k))*(B()*(2*v*(T()-t)+a*
    ((T()-t)*(T()-t)))+((v+a*(T()-t))*(v+a*(T()-t))-vh*
    vh))<=2*B()*(yg-eps())|(1+2*eps()*abs(k)+eps()*eps()
    *(k*k))*(B()*(2*v*(T()-t)+a*((T()-t)*(T()-t)))+((v+a
    *(T()-t))*(v+a*(T()-t))-vh*vh))<=2*B()*(abs(xg)-eps
    ())`}) ; orL(-11) ; <(
                    QE,
                    orL(-11) ; <(
                      QE,
                      cut({`(1+2*eps()*abs(k)+eps()*eps()*(k*k))*((vl*
        vl-v*v)/(2*A()))<=abs(xg)-eps()`}) ; <(
                        QE,
                        hideR(1==={`vl<=v|(1+2*eps()*abs(k)+eps()*eps()
    *(k*k))*(vl*vl-v*v)<=2*A()*(yg-eps())|(1+2*
    eps()*abs(k)+eps()*eps()*(k*k))*(vl*vl-v*v)<=2
    *A()*(abs(xg)-eps())`}) ; QE)))))))),
      hideL(-21==={`abs(k_0)*eps()<=1`}) ; hideL(-21==={`v<=vh_0|(1+2*
eps()*abs(k_0)+eps()*eps()*(k_0*k_0))*(v*v-vh_0*vh_0)<=2*B()*(
yg_0-eps())|(1+2*eps()*abs(k_0)+eps()*eps()*(k_0*k_0))*(v*v-
vh_0*vh_0)<=2*B()*(abs(xg_0)-eps())`}) ; hideL(-21==={`vl_0<=v|
(1+2*eps()*abs(k_0)+eps()*eps()*(k_0*k_0))*(vl_0*vl_0-v*v)<=2*
A()*(yg_0-eps())|(1+2*eps()*abs(k_0)+eps()*eps()*(k_0*k_0))*(
vl_0*vl_0-v*v)<=2*A()*(abs(xg_0)-eps())`}) ; hideL(-21==={`0<=
vl_0`}) ; hideL(-21==={`vl_0 < vh_0`}) ; hideL(-21==={`A()*T()<=
vh_0-vl_0`}) ; hideL(-21==={`B()*T()<=vh_0-vl_0`}) ; hideL(-21
==={`k_0*(eps()*eps())-2*eps() < k_0*(xg_0*xg_0+yg_0*yg_0)-2*
xg_0`}) ; hideL(-21==={`k_0*(xg_0*xg_0+yg_0*yg_0)-2*xg_0 < k_0*
(eps()*eps())+2*eps()`}) ; boxAnd(1) ; andR(1) ; <(
        hideL(-12==={`k*(eps()*eps())-2*eps() < k*(xg*xg+yg*yg)-2*xg
  `}) ; hideL(-12==={`k*(xg*xg+yg*yg)-2*xg < k*(eps()*eps())+2*
```

```
eps()`}) ; hideL(-7=={`abs(k)*eps()<=1`}) ; hideL(-13=={`vl
<=v&v+a*T()>=vl|(1+2*eps()*abs(k)+eps()*eps()*(k*k))*(A()*(2
*v*T()+a*(T()*T()))+(vl*vl-(v+a*T())*(v+a*T())))<=2*A()*(
abs(xg)-eps())|(1+2*eps()*abs(k)+eps()*eps()*(k*k))*(A()*(2*
v*T()+a*(T()*T()))+(vl*vl-(v+a*T())*(v+a*T())))<=2*A()*(yg-
eps())`}) ; orL(-12) ; <(
        cut({`a>=0|a<=0`}) ; <(
          orL(-17) ; <(
            MR({`a>=0&v+a*(T()-t)<=vh`}, 1) ; <(
              dI(1),
              prop),
            MR({`a<=0&v<=vh`}, 1) ; <(
              dI(1),
              prop)),
          hideR(1=={`[{xg'=-v*k*yg,yg'=v*(k*xg-1),v'=a,t'=1&((v>=0
    &t<=T())&t>=0&k*(eps()*eps())-2*eps() < k*(xg*xg+yg*yg)-
    2*xg&k*(xg*xg+yg*yg)-2*xg < k*(eps()*eps())+2*eps())&v+a
    *(T()-t)>=0}](a>=0&v+a*(T()-t)<=vh|a<=0&v<=vh)|(1+2*
    eps()*abs(k)+eps()*eps()*(k*k))*(B()*(2*v*(T()-t)+a*((
    T()-t)*(T()-t)))+((v+a*(T()-t))*(v+a*(T()-t))-vh*vh))<=2
    *B()*(yg-eps())|(1+2*eps()*abs(k)+eps()*eps()*(k*k))*(
    B()*(2*v*(T()-t)+a*((T()-t)*(T()-t)))+((v+a*(T()-t))*(v+
    a*(T()-t))-vh*vh))<=2*B()*(abs(xg)-eps()))`}) ; QE),
        orL(-12) ; <(
          orL(-5) ; <(
            cut({`abs(k)=k&abs(xg)=xg`}) ; <(
              andL(-17) ; allL2R(-17) ; hideL(-17=={`abs(k)=k`}) ;
  allL2R(-17) ; hideL(-17=={`abs(xg)=xg`}) ; MR({`(1+2
  *eps()*k+eps()*eps()*(k*k))*(v*(T()-t)+a/2*((T()-t)*
  (T()-t))+((v+a*(T()-t))*(v+a*(T()-t))-vh*vh)/(2*B())
  )<=xg-eps()`}, 1) ; <(
                hideL(-9=={`A()*T()<=vh-vl`}) ; hideL(-9=={`B()*
  T()<=vh-vl`}) ; hideL(-9=={`v+a*T()>=0`}) ; hideL(
  -6=={`yg>0`}) ; hideL(-3=={`A()>0`}) ; hideL(-9==
  {`a<=A()`}) ; dI(1),
                QE),
              hideR(1=={`[{xg'=-v*k*yg,yg'=v*(k*xg-1),v'=a,t'=1&((
  v>=0&t<=T())&t>=0&k*(eps()*eps())-2*eps() < k*(xg*xg
  +yg*yg)-2*xg&k*(xg*xg+yg*yg)-2*xg < k*(eps()*eps())+
  2*eps())&v+a*(T()-t)>=0}](a>=0&v+a*(T()-t)<=vh|a<=0
  &v<=vh)|(1+2*eps()*abs(k)+eps()*eps()*(k*k))*(B()*(2
  *v*(T()-t)+a*((T()-t)*(T()-t)))+((v+a*(T()-t))*(v+a*
  (T()-t))-vh*vh))<=2*B()*(yg-eps())|(1+2*eps()*abs(k)
  +eps()*eps()*(k*k))*(B()*(2*v*(T()-t)+a*((T()-t)*(
  T()-t)))+((v+a*(T()-t))*(v+a*(T()-t))-vh*vh))<=2*B()
  *(abs(xg)-eps()))`}) ; QE),
```

```
            hideL(-3=={`A()>0`}) ; hideL(-8=={`A()*T()<=vh-vl`}) ;
hideL(-8=={`B()*T()<=vh-vl`}) ; hideL(-8=={`v+a*T()>=0
`}) ; cut({`abs(k)=-k&abs(xg)=-xg`}) ; <(
              andL(-13) ; allL2R(-13) ; hideL(-13=={`abs(k)=-k`})
  ; allL2R(-13) ; hideL(-13=={`abs(xg)=-xg`}) ; MR({`(
  1+2*eps()*(-k)+eps()*eps()*(k*k))*(v*(T()-t)+a/2*((
  T()-t)*(T()-t))+((v+a*(T()-t))*(v+a*(T()-t))-vh*vh)/
  (2*B()))<=-xg-eps()`}, 1) ; <(
                hideL(-5=={`yg>0`}) ; dI(1),
                QE),
              hideR(1=={`[{xg'=-v*k*yg,yg'=v*(k*xg-1),v'=a,t'=1&((
  v>=0&t<=T())&t>=0&k*(eps()*eps())-2*eps() < k*(xg*xg
  +yg*yg)-2*xg&k*(xg*xg+yg*yg)-2*xg < k*(eps()*eps())+
  2*eps())&v+a*(T()-t)>=0}](a>=0&v+a*(T()-t)<=vh|a<=0
  &v<=vh)|(1+2*eps()*abs(k)+eps()*eps()*(k*k))*(B()*(2
  *v*(T()-t)+a*((T()-t)*(T()-t)))+((v+a*(T()-t))*(v+a*
  (T()-t))-vh*vh))<=2*B()*(yg-eps())|(1+2*eps()*abs(k)
  +eps()*eps()*(k*k))*(B()*(2*v*(T()-t)+a*((T()-t)*(
  T()-t)))+((v+a*(T()-t))*(v+a*(T()-t))-vh*vh))<=2*B()
  *(abs(xg)-eps()))`}) ; QE)),
          MR({`(1+2*eps()*abs(k)+eps()*eps()*(k*k))*(v*(T()-t)+a/2
    *((T()-t)*(T()-t))+((v+a*(T()-t))*(v+a*(T()-t))-vh*vh)/(
    2*B()))<=yg-eps()`}, 1) ; <(
            orL(-5) ; <(
              cut({`abs(k)=k`}) ; <(
                allL2R(-17) ; hideL(-17=={`abs(k)=k`}) ; dI(1),
                hideR(1=={`[{xg'=-v*k*yg,yg'=v*(k*xg-1),v'=a,t'=1&
    ((v>=0&t<=T())&t>=0&k*(eps()*eps())-2*eps() < k*(
    xg*xg+yg*yg)-2*xg&k*(xg*xg+yg*yg)-2*xg < k*(eps()*
    eps())+2*eps())&v+a*(T()-t)>=0}](1+2*eps()*abs(k)+
    eps()*eps()*(k*k))*(v*(T()-t)+a/2*((T()-t)*(T()-t)
    )+((v+a*(T()-t))*(v+a*(T()-t))-vh*vh)/(2*B()))<=yg
    -eps()`}) ; QE),
              cut({`abs(k)=-k`}) ; <(
                allL2R(-17) ; hideL(-17=={`abs(k)=-k`}) ; dI(1),
                hideR(1=={`[{xg'=-v*k*yg,yg'=v*(k*xg-1),v'=a,t'=1&
    ((v>=0&t<=T())&t>=0&k*(eps()*eps())-2*eps() < k*(
    xg*xg+yg*yg)-2*xg&k*(xg*xg+yg*yg)-2*xg < k*(eps()*
    eps())+2*eps())&v+a*(T()-t)>=0}](1+2*eps()*abs(k)+
    eps()*eps()*(k*k))*(v*(T()-t)+a/2*((T()-t)*(T()-t)
    )+((v+a*(T()-t))*(v+a*(T()-t))-vh*vh)/(2*B()))<=yg
    -eps()`}) ; QE)),
            QE))),
      hideL(-15=={`v<=vh&v+a*T()<=vh|(1+2*eps()*abs(k)+eps()*eps()
*(k*k))*(B()*(2*v*T()+a*(T()*T()))+((v+a*T())*(v+a*T())-vh*
vh))<=2*B()*(abs(xg)-eps())|(1+2*eps()*abs(k)+eps()*eps()*(k
```

```
*k))*(B()*(2*v*T()+a*(T()*T()))+((v+a*T())*(v+a*T())-vh*vh))
<=2*B()*(yg-eps())`}) ; hideL(-12=={`k*(eps()*eps())-2*eps()
< k*(xg*xg+yg*yg)-2*xg`}) ; hideL(-12=={`k*(xg*xg+yg*yg)-2*
xg < k*(eps()*eps())+2*eps()`}) ; hideL(-7=={`abs(k)*eps()<=
1`}) ; hideL(-9=={`A()*T()<=vh-vl`}) ; hideL(-9=={`B()*T()<=
vh-vl`}) ; orL(-10) ; <(
        cut({`a>=0|a<=0`}) ; <(
          orL(-15) ; <(
            MR({`a>=0&v>=vl`}, 1) ; <(
              dI(1),
              QE),
            MR({`a<=0&v+a*(T()-t)>=vl`}, 1) ; <(
              dI(1),
              QE)),
          hideR(1=={`[{xg'=-v*k*yg,yg'=v*(k*xg-1),v'=a,t'=1&((v>=0
    &t<=T())&t>=0&k*(eps()*eps())-2*eps() < k*(xg*xg+yg*yg)-
    2*xg&k*(xg*xg+yg*yg)-2*xg < k*(eps()*eps())+2*eps())&v+a
    *(T()-t)>=0}](a>=0&v>=vl|a<=0&v+a*(T()-t)>=vl)|(1+2*
    eps()*abs(k)+eps()*eps()*(k*k))*(A()*(2*v*(T()-t)+a*((
    T()-t)*(T()-t)))+(vl*vl-(v+a*(T()-t))*(v+a*(T()-t))))<=2
    *A()*(yg-eps())|(1+2*eps()*abs(k)+eps()*eps()*(k*k))*(
    A()*(2*v*(T()-t)+a*((T()-t)*(T()-t)))+(vl*vl-(v+a*(T()-t
    ))*(v+a*(T()-t))))<=2*A()*(abs(xg)-eps()))`}) ; QE),
        orL(-10) ; <(
          orL(-5) ; <(
            cut({`abs(k)=k&abs(xg)=xg`}) ; <(
              andL(-15) ; allL2R(-15) ; hideL(-15=={`abs(k)=k`}) ;
  allL2R(-15) ; hideL(-15=={`abs(xg)=xg`}) ; MR({`(1+2
  *eps()*k+eps()*eps()*(k*k))*(v*(T()-t)+a/2*((T()-t)*
  (T()-t))+(vl*vl-(v+a*(T()-t))*(v+a*(T()-t)))/(2*A())
  )<=xg-eps()`}, 1) ; <(
                hideL(-6=={`yg>0`}) ; dI(1),
                QE),
              hideR(1=={`[{xg'=-v*k*yg,yg'=v*(k*xg-1),v'=a,t'=1&((
  v>=0&t<=T())&t>=0&k*(eps()*eps())-2*eps() < k*(xg*xg
  +yg*yg)-2*xg&k*(xg*xg+yg*yg)-2*xg < k*(eps()*eps())+
  2*eps())&v+a*(T()-t)>=0}](a>=0&v>=vl|a<=0&v+a*(T()-
  t)>=vl)|(1+2*eps()*abs(k)+eps()*eps()*(k*k))*(A()*(2
  *v*(T()-t)+a*((T()-t)*(T()-t)))+(vl*vl-(v+a*(T()-t))
  *(v+a*(T()-t))))<=2*A()*(yg-eps())|(1+2*eps()*abs(k)
  +eps()*eps()*(k*k))*(A()*(2*v*(T()-t)+a*((T()-t)*(
  T()-t)))+(vl*vl-(v+a*(T()-t))*(v+a*(T()-t))))<=2*A()
  *(abs(xg)-eps()))`}) ; QE),
            cut({`abs(k)=-k&abs(xg)=-xg`}) ; <(
              andL(-15) ; allL2R(-15) ; hideL(-15=={`abs(k)=-k`});
  allL2R(-15) ; hideL(-15=={`abs(xg)=-xg`}) ; MR({`(1+
```

```
2*eps()*(-k)+eps()*eps()*(k*k))*(v*(T()-t)+a/2*((T()
-t)*(T()-t))+(vl*vl-(v+a*(T()-t))*(v+a*(T()-t)))/(2*
A()))<=-xg-eps()`}, 1) ; <(
              hideL(-6=={`yg>0`}) ; dI(1),
              QE),
            hideR(1=={`[{xg'=-v*k*yg,yg'=v*(k*xg-1),v'=a,t'=1&((
v>=0&t<=T())&t>=0&k*(eps()*eps())-2*eps() < k*(xg*xg
+yg*yg)-2*xg&k*(xg*xg+yg*yg)-2*xg < k*(eps()*eps())+
2*eps())&v+a*(T()-t)>=0}](a>=0&v>=vl|a<=0&v+a*(T()-
t)>=vl)|(1+2*eps()*abs(k)+eps()*eps()*(k*k))*(A()*(2
*v*(T()-t)+a*((T()-t)*(T()-t)))+(vl*vl-(v+a*(T()-t))
*(v+a*(T()-t))))<=2*A()*(yg-eps())|(1+2*eps()*abs(k)
+eps()*eps()*(k*k))*(A()*(2*v*(T()-t)+a*((T()-t)*(T(
)-t)))+(vl*vl-(v+a*(T()-t))*(v+a*(T()-t))))<=2*A()*(
abs(xg)-eps()))`}) ; QE)),
        MR({`(1+2*eps()*abs(k)+eps()*eps()*(k*k))*(v*(T()-t)+a/2
  *((T()-t)*(T()-t))+(vl*vl-(v+a*(T()-t))*(v+a*(T()-t)))/(
  2*A()))<=yg-eps()`}, 1) ; <(
          orL(-5) ; <(
            cut({`abs(k)=k`}) ; <(
              allL2R(-15) ; hideL(-15=={`abs(k)=k`}) ; dI(1),
              hideR(1=={`[{xg'=-v*k*yg,yg'=v*(k*xg-1),v'=a,t'=1&
  ((v>=0&t<=T())&t>=0&k*(eps()*eps())-2*eps() < k*(
  xg*xg+yg*yg)-2*xg&k*(xg*xg+yg*yg)-2*xg < k*(eps()*
  eps())+2*eps())&v+a*(T()-t)>=0}](1+2*eps()*abs(k)+
  eps()*eps()*(k*k))*(v*(T()-t)+a/2*((T()-t)*(T()-t)
  )+(vl*vl-(v+a*(T()-t))*(v+a*(T()-t)))/(2*A()))<=yg
  -eps()`}) ; QE),
            cut({`abs(k)=-k`}) ; <(
              allL2R(-15) ; hideL(-15=={`abs(k)=-k`}) ; dI(1),
              hideR(1=={`[{xg'=-v*k*yg,yg'=v*(k*xg-1),v'=a,t'=1&
  ((v>=0&t<=T())&t>=0&k*(eps()*eps())-2*eps() < k*(
  xg*xg+yg*yg)-2*xg&k*(xg*xg+yg*yg)-2*xg < k*(eps()*
  eps())+2*eps())&v+a*(T()-t)>=0}](1+2*eps()*abs(k)+
  eps()*eps()*(k*k))*(v*(T()-t)+a/2*((T()-t)*(T()-t)
  )+(vl*vl-(v+a*(T()-t))*(v+a*(T()-t)))/(2*A()))<=yg
  -eps()`}) ; QE)),
           QE))))),
hideL(-5=={`xg>=0&k>=0|xg<=0&k<=0`}) ; hideL(-6=={`abs(k)*eps()
<=1`}) ; hideL(-13=={`v<=vh&v+a*T()<=vh|(1+2*eps()*abs(k)+eps()*
eps()*(k*k))*(B()*(2*v*T()+a*(T()*T()))+((v+a*T())*(v+a*T())-vh*
vh))<=2*B()*(abs(xg)-eps())|(1+2*eps()*abs(k)+eps()*eps()*(k*k))
*(B()*(2*v*T()+a*(T()*T()))+((v+a*T())*(v+a*T())-vh*vh))<=2*B()*
(yg-eps())`}) ; hideL(-13=={`vl<=v&v+a*T()>=vl|(1+2*eps()*abs(k)
+eps()*eps()*(k*k))*(A()*(2*v*T()+a*(T()*T()))+(vl*vl-(v+a*T())*
(v+a*T())))<=2*A()*(abs(xg)-eps())|(1+2*eps()*abs(k)+eps()*eps()
```

```
      *(k*k))*(A()*(2*v*T()+a*(T()*T()))+(vl*vl-(v+a*T())*(v+a*T())))
      <=2*A()*(yg-eps())`}) ; dI(1)),
    hideL(-5=={`xg>=0&k>=0|xg<=0&k<=0`}) ; hideL(-6=={`abs(k)*eps()<=1
    `}) ; hideL(-13=={`v<=vh&v+a*T()<=vh|(1+2*eps()*abs(k)+eps()*eps()
    *(k*k))*(B()*(2*v*T()+a*(T()*T()))+((v+a*T())*(v+a*T())-vh*vh))<=2
    *B()*(abs(xg)-eps())|(1+2*eps()*abs(k)+eps()*eps()*(k*k))*(B()*(2*
    v*T()+a*(T()*T()))+((v+a*T())*(v+a*T())-vh*vh))<=2*B()*(yg-eps())`
    }) ; hideL(-13=={`vl<=v&v+a*T()>=vl|(1+2*eps()*abs(k)+eps()*eps()*
    (k*k))*(A()*(2*v*T()+a*(T()*T()))+(vl*vl-(v+a*T())*(v+a*T())))<=2*
    A()*(abs(xg)-eps())|(1+2*eps()*abs(k)+eps()*eps()*(k*k))*(A()*(2*v
    *T()+a*(T()*T()))+(vl*vl-(v+a*T())*(v+a*T())))<=2*A()*(yg-eps())`}
    ) ; dI(1)))
End.
End.

Theorem "RAL"

ProgramVariables
  Real xg,yg; /* goal position: (dm,dm) */
  Real k;     /* curvature:     centi-(m^-1) */

  Real vl,vh; /* goal velocities (upper and lower): dm/s */

  Real t;     /* time trigger:  ds */
  Real v,a;   /* linear velocity/acceleration:  dm/s,  dm/s^2 */
End.

Problem
  T > 0 &
  eps() > 0 &
  A > 0 &
  B > 0 &
  J(xg, yg, k, v, vl, vh)
  ->
  [{
  /* Control */
  {
    {
      {
        /* Turning from left or right quadrant towards origin
           (k <= 0, xg <= 0) or (k >= 0, xg >= 0) */
        xg:=*; yg:=*; vl:=*; vh:=*; k:=*; a:=*;
        ? xg >= 0 & k >= 0 | xg <= 0 & k <= 0;

        /* Standard sanity conditions for the planner inputs:
           1. Goal (xg,yg) is in upper half plane
```

```
            2. Curvature k and bloating eps define an annulus
            3. Goal is within safe annular region
            4. Target speed limit interval (vl,vh) is wide enough
               (and >= 0)
         */

         ? onUpperHalfPlane(xg,yg) &
           isAnnulus(k) &
           onAnnulus(xg,yg,k) &
           controllableSpeedGoal(vl,vh);

         ? admissibleAcc(xg, yg, v, vl, vh, a, k);
       }
     }
     t:=0;
   }
   /* Plant */
   {
     { xg'=-v*k*yg, yg'=v*(k*xg -1), v'=a, t'=1 & (v >= 0 & t <= T) }
   }
   }*
   ]( sq(xg) + sq(yg) <= sq(eps()) -> vl <= v & v <= vh )
End.

Tactic "RAL"
implyR(1) ; MR({`v>=0&abs(k)*eps()<=1&(k*(eps()*eps())-2*eps() < k*(xg
*xg+yg*yg)-2*xg&k*(xg*xg+yg*yg)-2*xg < k*(eps()*eps())+2*eps())&(0<=vl
&vl < vh&A()*T()<=vh-vl&B()*T()<=vh-vl)&(v<=vh|(1+2*eps()*abs(k)+eps()
*eps()*(k*k))*(v*v-vh*vh)<=2*B()*(yg-eps())|(1+2*eps()*abs(k)+eps()*
eps()*(k*k))*(v*v-vh*vh)<=2*B()*(abs(xg)-eps()))&(vl<=v|(1+2*eps()*abs
(k)+eps()*eps()*(k*k))*(vl*vl-v*v)<=2*A()*(yg-eps())|(1+2*eps()*abs(k)
+eps()*eps()*(k*k))*(vl*vl-v*v)<=2*A()*(abs(xg)-eps()))`}, 1) ; <(
  useLemma({`RALLemma`}, {`prop`}),
  unfold ; orL(-9) ; <(
    orL(-8) ; <(
      prop,
      QE),
    orL(-8) ; <(
      QE,
      hideL(-7=={`abs(k)*eps()<=1`}) ; hideL(-13=={`k*(eps()*eps())-2*
      eps() < k*(xg*xg+yg*yg)-2*xg`}) ; hideL(-13=={`k*(xg*xg+yg*yg)-2
      *xg < k*(eps()*eps())+2*eps()`}) ; hideL(-12=={`B()*T()<=vh-vl`}
      ) ; hideL(-11=={`A()*T()<=vh-vl`}) ; andR(1) ; <(
        hideL(-7=={`(1+2*eps()*abs(k)+eps()*eps()*(k*k))*(v*v-vh*vh)<=
  2*B()*(yg-eps())|(1+2*eps()*abs(k)+eps()*eps()*(k*k))*(v*v-vh*
  vh)<=2*B()*(abs(xg)-eps())`}) ; QE,
```

```
        hideL(-8=={`(1+2*eps()*abs(k)+eps()*eps()*(k*k))*(vl*vl-v*v)<=
  2*A()*(yg-eps())|(1+2*eps()*abs(k)+eps()*eps()*(k*k))*(vl*vl-v
  *v)<=2*A()*(abs(xg)-eps())`}) ; QE))))
End.
End.
```

The debugging print statements in the following proof are not designed to be readable to the reader. They are presented in their original form in the interest of faithfully showing how the tool got used: as with any language, terse debugging statements are often used during development as a note from the proof author to themselves.

```
Kaisar model, RA-L:
let sq(x) = (x*x);
let onUpperHalfPlane(x, y) <-> (y > 0);
let circle(x, y, k) = (k*(sq(x)+sq(y)) - 2*x);
let isAnnulus(k) <-> (abs(k)*eps <= 1);
let onAnnulus(x, y, k) <-> (
                    (k*sq(eps) - 2*eps) < circle(x,y,k)
  & circle(x,y,k) < (k*sq(eps) + 2*eps));
let isIvl(vl, vh) <-> (0 <= vl & vl < vh);
let loCSG(vl, vh) <-> (A * T <=  (vh - vl));
let hiCSG(vl, vh) <-> (B * T <=  (vh - vl));
let controllableSpeedGoal(vl, vh) <-> (
  isIvl(vl, vh) &
  loCSG(vl, vh) &
  hiCSG(vl, vh));
let controllableGoalDist(v1, v2, a, xg, yg, k) <-> (
    v1 <= v2 |
    (xg>=0&k>=0 &
     ((1 + 2*eps*  k + sq(eps*k)) * (sq(v1)-sq(v2)) <= (2*a) *
     (yg-eps) |
      (1 + 2*eps*k + sq(eps*k)) * (sq(v1)-sq(v2)) <= (2*a) * (xg-eps))
   | xg<=0&k<=0 &
     ((1 + 2*eps*(-k) + sq(eps*k)) * (sq(v1)-sq(v2)) <= (2*a) *
     (yg-eps) |
      (1 + 2*eps*(-k) + sq(eps*k)) * (sq(v1)-sq(v2)) <= (2*a) *
      (-xg-eps))));
let J(xg, yg, k, v, vl, vh) <-> (
  v >= 0 &
  isAnnulus(k) &
  onAnnulus(xg,yg,k) &
  controllableSpeedGoal(vl,vh) &
  controllableGoalDist(v,vh,B,xg,yg,k) &
  controllableGoalDist(vl,v,A,xg,yg,k));
let admRange(a) <-> (-B <= a & a <= A);
let admBrake(v,a) <-> (v + a*T >= 0);
let negBounds(xg,yg,v,vl,vh,a,k) <->
```

```
   ((xg<=0&k<=0) &
     ((vl <= v & vl <= v+a*T
     |  (1 + 2*eps*(-k) + sq(eps*k)) * (A*(2*v*T+a*sq(T)) + (sq(vl) -
     sq(v+a*T))) <= (2*A) * (-xg-eps)
     |  (1 + 2*eps*(-k) + sq(eps*k)) * (A*(2*v*T+a*sq(T)) + (sq(vl) -
     sq(v+a*T))) <= (2*A) * (yg-eps))
   & (v <= vh & v+a*T <= vh
     | (1 + 2*eps*(-k) + sq(eps*k)) * (B*(2*v*T+a*sq(T)) + (sq(v+a*T)
     - sq(vh))) <= (2*B) * (-xg-eps)
     | (1 + 2*eps*(-k) + sq(eps*k)) * (B*(2*v*T+a*sq(T)) + (sq(v+a*T)
     - sq(vh))) <= (2*B) * (yg-eps))));
let posBounds(xg,yg,v,vl,vh,a,k) <->
   ((xg>=0&k>=0) &
     (( vl <= v & vl <= v+a*T
     | (1 + 2*eps*(k) + sq(eps*k)) * (A*(2*v*T+a*sq(T)) + (sq(vl) - sq
     (v+a*T))) <= (2*A) * (xg-eps)
     | (1 + 2*eps*(k) + sq(eps*k)) * (A*(2*v*T+a*sq(T)) + (sq(vl) - sq
     (v+a*T))) <= (2*A) * (yg-eps))
   & (v <= vh & v+a*T <= vh
     | (1 + 2*eps*(k) + sq(eps*k)) * (B*(2*v*T+a*sq(T)) + (sq(v+a*T)
     - sq(vh))) <= (2*B) * (xg-eps)
     | (1 + 2*eps*(k) + sq(eps*k)) * (B*(2*v*T+a*sq(T)) + (sq(v+a*T)
     - sq(vh))) <= (2*B) * (yg-eps))));
let anyBounds(xg,yg,v,vl,vh,a,k) <-> (vl <= v & v+a*T >= vl & v <= vh
& v+a*T <= vh);
let admBounds(xg,yg,v,vl,vh,a,k) <-> (
   negBounds(xg,yg,v,vl,vh,a,k)
 | posBounds(xg,yg,v,vl,vh,a,k)
 | anyBounds(xg,yg,v,vl,vh,a,k));
let admissibleAcc(xg, yg, v, vl, vh, a, k) <-> (
  admRange(a) &
  admBrake(v,a) &
  admBounds(xg,yg,v,vl,vh,a,k));
?(const, bc):((T > 0 & eps > 0 & A > 0 & B > 0) & J(xg,yg,k,v,vl,vh));
!constA:(T > 0 & eps > 0 & A > 0) using const by prop;
!constB:(T > 0 & eps > 0 & B > 0) using const by prop;
!inv:(J(xg, yg, k, v, vl, vh));
{body:
 !ihV:(v >= 0) using inv by prop;
 !ihCgdL:(controllableGoalDist(vl,v,A,xg,yg,k)) using inv by prop;
 !ihCgdU:(controllableGoalDist(v,vh,B,xg,yg,k)) using inv by prop;
 {xg:=*;yg:=*;vl:=*;vh:=*;k:=*;a:=*;
  ?dir:(xg>=0&k>=0|xg<=0&k<=0);
  ?(uhp, isAnn, onAnn, csgIvl, csgLo, csgHi):
   (onUpperHalfPlane(xg,yg) &
    isAnnulus(k) &
```

```
    onAnnulus(xg,yg,k) &
    isIvl(vl, vh) &
    loCSG(vl, vh) &
    hiCSG(vl, vh));
  note csg = andI(csgIvl, andI(csgLo, csgHi));
  adm:
  ?(accLo, accHi, noBrake, admBounds):(-B <= a & a <= A & admBrake
  (v,a) & admBounds(xg,yg,v,vl,vh,a,k));
  note accRange = andI(accLo, accHi);
  ?accSgn:(a <= 0 | a >= 0);}
 switch (admBounds) {
 case kAnyAdm:(anyBounds(xg,yg,v,vl,vh,a,k)) =>
   print("CASE sgn(k) irrelevant");
   switch (accSgn) {
     case accNeg:(a<=0) =>
      print("subcase a<=0");
       t:=0;
       { xg'=-v*k*yg, yg'=v*(k*xg-1), v'=a, t'=1
       & ?dom:(v >= 0 & t <= T);
       & !tPos:(t>=0);
       & !dLo:((k*(eps*eps)-2*eps) < k*(sq(xg)+sq(yg))-2*xg)
       using isAnn onAnn by induction;
       & !dHi:(k*(xg*xg+yg*yg)-2*xg < (k*sq(eps)+2*eps))
       using isAnn onAnn by induction;
       & !vBound:(v+a*(T-t)>=0) using tPos noBrake by induction;
       & !vLoInd:(v + a*(T-t) >= vl)
using  accSgn  kAnyAdm dLo dHi dom tPos vBound const by induction;
       & !vHiInd:(v + a*(T-t) <= vh)
using  accSgn  kAnyAdm dLo dHi dom tPos vBound const by induction;
       & !vDec:(v <= v@adm) using accNeg by induction;
       & !vDecSlo:(v >= v@adm + a*t) using accNeg by induction;};
       ?xgSgnC:(xg<=0);
       !indVPos:(v >= 0) using dom by auto;
       !indIsAnn:(isAnnulus(k)) using isAnn by auto;
       !indOnAnn:(onAnnulus(xg,yg,k)) using onAnn dLo dHi by auto;
       print("After ODE, CASE sgn(k), irrelevant, a <= 0");
       !indCGDHBndA:(v <= vh)
using vDec vHiInd kAnyAdm dom tPos accRange by auto;
       !indCGDH:(controllableGoalDist(v,vh,B,xg,yg,k))
       using indCGDHBndA by auto;
       !indCGDLBndA:(vl <= v)
using vDec vDecSlo vLoInd kAnyAdm dom tPos accRange csg by auto;
       !indCGDL:(controllableGoalDist(vl,v,A,xg,yg,k))
       using indCGDLBndA by auto;
       !indCSG:(controllableSpeedGoal(vl,vh)) using vDec csg by auto;
       note indStepA = andI(andI(andI(andI(andI(indVPos, indIsAnn),
```

```
      indOnAnn), indCSG), indCGDH), indCGDL);
      !indAssert:(J(xg, yg, k, v, vl, vh)) using indStepA by auto;
    case accPos:(a>=0) =>
      print("subcase a>=0");
      t:=0;
      { xg'=-v*k*yg, yg'=v*(k*xg-1), v'=a, t'=1
      & ?dom:(v >= 0 & t <= T);
      & !tPos:(t>=0);
      & !dLo:((k*(eps*eps)-2*eps) < k*(sq(xg)+sq(yg))-2*xg)
      using isAnn onAnn by induction;
      & !dHi:(k*(xg*xg+yg*yg)-2*xg < (k*sq(eps)+2*eps))
      using isAnn onAnn by induction;
      & !vBound:(v+a*(T-t)>=0) using tPos noBrake by induction;
      & !vLoInd:(v + a*(T-t) >= vl)
using  accSgn  kAnyAdm dLo dHi dom tPos vBound const by induction;
      & !vHiInd:(v + a*(T-t) <= vh)
using  accSgn  kAnyAdm dLo dHi dom tPos vBound const by induction;
      & !vInc:(v >= v@adm) using accPos by induction;
      & !vIncSlo:(v <= v@adm + a*t) using accPos by induction;};
      ?xgSgnC:(xg<=0);
      !indVPos:(v >= 0) using dom by auto;
      !indIsAnn:(isAnnulus(k)) using isAnn by auto;
      !indOnAnn:(onAnnulus(xg,yg,k)) using onAnn dLo dHi by auto;
      print("After ODE, CASE sgn(k), irrelevant, a >= 0");
      !indCGDHBndA:(v <= vh)
      using vInc vIncSlo vHiInd kAnyAdm dom tPos accRange by auto;
      !indCGDH:(controllableGoalDist(v,vh,B,xg,yg,k))
      using indCGDHBndA by auto;
      !indCGDLBndA:(vl <= v)
      using vInc vLoInd kAnyAdm dom tPos accRange csg by auto;
      !indCGDL:(controllableGoalDist(vl,v,A,xg,yg,k))
      using indCGDLBndA by auto;
      !indCSG:(controllableSpeedGoal(vl,vh))
      using vInc vIncSlo csg by auto;
      note indStepA = andI(andI(andI(andI(andI(indVPos, indIsAnn),
      indOnAnn), indCSG), indCGDH), indCGDL);
      !indAssert:(J(xg, yg, k, v, vl, vh)) using indStepA by auto;}
 case kNegAdm:(negBounds(xg,yg,v,vl,vh,a,k)) =>
   note kSgn = andEL(kNegAdm);
   note admL = andEL(andER(kNegAdm));
   note admH = andER(andER(kNegAdm));
   switch (accSgn) {
     case accNeg:(a<=0) =>
        print("CASE k<=0,a<=0");
        t := 0;
        { xg'=-v*k*yg, yg'=v*(k*xg-1), v'=a, t'=1
```

```
        & ?dom:(v >= 0 & t <= T);
        & !tPos:(t>=0) by induction;
        & !dLo:((k*(eps*eps)-2*eps) < k*(sq(xg)+sq(yg))-2*xg)
  using isAnn onAnn by induction;
        & !dHi:(k*(xg*xg+yg*yg)-2*xg < (k*sq(eps)+2*eps))
  using isAnn onAnn by induction;
        & !vBound:(v+a*(T-t)>=0) using tPos noBrake by induction;
        & !bigL:(((a<=0&vl<=v+a*(T-t))
                  |(1+2*eps*(-k)+sq(eps*k))*(A*(2*v*(T-t)+a*(sq(T-t)))
      +(sq(vl)-sq(v+a*(T-t))))<=2*A*(yg-eps)
                  |(1+2*eps*(-k)+sq(eps*k))*(A*(2*v*(T-t)+a*(sq(T-t)))
      +(sq(vl)-sq(v+a*(T-t))))<=2*A*((-xg)-eps)))
          using constA dom tPos dLo dHi vBound accHi noBrake csgIvl
    csgHi uhp isAnn onAnn accSgn kSgn admL accNeg by induction;
        & !bigH:(((a<=0&v<=vh)
                  |(1+2*eps*(-k)+sq(eps*k))*(B*(2*v*(T-t)+a*(sq(T-t)))
      +(sq(v+a*(T-t))-sq(vh)))<=2*B*(yg-eps)
                  |(1+2*eps*(-k)+sq(eps*k))*(B*(2*v*(T-t)+a*(sq(T-t)))
      +(sq(v+a*(T-t))-sq(vh)))<=2*B*((-xg)-eps)))
using constB dom tPos dLo dHi vBound accRange noBrake csgIvl csgLo
uhp isAnn onAnn accSgn kSgn admH accNeg by induction;};
        ?xgSgnC:(xg<=0);
        !indVPos:(v >= 0) using dom by auto;
        !indIsAnn:(isAnnulus(k)) using isAnn by auto;
        !indOnAnn:(onAnnulus(xg,yg,k)) using onAnn dLo dHi by auto;
        print("After ODE, CASE k<=0,a<=0");
        switch(bigL) {
          case loA:((a<=0&vl<=v+a*(T-t))) =>
            print("loA<<");
            !lem:(vl <= v) using loA accNeg dom tPos by auto;
            !indCGDL:(controllableGoalDist(vl,v,A,xg,yg,k))
      using lem kSgn by auto;
          case loB:((1+2*eps*(-k)+sq(eps*k))*(A*(2*v*(T-t)+a*(sq(T-t))
    )+(sq(vl)-sq(v+a*(T-t))))<=2*A*(yg-eps)) =>
            print("loB<<");
            !lem:((1 + 2*eps*(-k) + sq(eps*k)) * (sq(vl)-sq(v)) <=
      (2*A) * (yg-eps))
              using const csgLo accRange vBound dom loB by auto;
            !indCGDL:(controllableGoalDist(vl,v,A,xg,yg,k))
      using lem kSgn xgSgnC by auto;
          case loC:((1+2*eps*(-k)+sq(eps*k))*(A*(2*v*(T-t)+a*
    (sq(T-t)))+(sq(vl)-sq(v+a*(T-t))))<=2*A*((-xg)-eps)) =>
            print("loC<<");
            !lem:((1 + 2*eps*(-k) + sq(eps*k)) * (sq(vl)-sq(v)) <=
      (2*A) * (-xg-eps))
              using const csgLo accRange vBound  dom  loC by auto;
```

```
            !indCGDL:(controllableGoalDist(vl,v,A,xg,yg,k))
      using lem kSgn xgSgnC by auto;
        }
        switch(bigH) {
          case hiA:((a<=0&v<=vh)) =>
            print("hiA<<");
            !lem:(v <= vh) using hiA by prop;
            !indCGDH:(controllableGoalDist(v,vh,B,xg,yg,k))
      using lem kSgn by auto;
          case hiB:((1+2*eps*(-k)+sq(eps*k))*(B*(2*v*(T-t)+a*(sq(T-t))
    )+(sq(v+a*(T-t))-sq(vh)))<=2*B*(yg-eps)) =>
            print("hiB<<");
            !lem:((1 + 2*eps*(-k) + sq(eps*k)) * (sq(v)-sq(vh)) <=
      (2*B) * (yg-eps))
              using const csgHi accRange vBound dom  hiB by auto;
            !indCGDH:(controllableGoalDist(v,vh,B,xg,yg,k))
      using lem kSgn xgSgnC by auto;
          case hiC:((1+2*eps*(-k)+sq(eps*k))*(B*(2*v*(T-t)+a*(sq(T-t))
    )+(sq(v+a*(T-t))-sq(vh)))<=2*B*((-xg)-eps)) =>
            print("hiC<<");
            !lem:((1 + 2*eps*(-k) + sq(eps*k)) * (sq(v)-sq(vh)) <=
      (2*B) * (-xg-eps))
              using const csgHi accRange vBound dom hiC by auto;
            !indCGDH:(controllableGoalDist(v,vh,B,xg,yg,k))
      using lem kSgn xgSgnC by auto;}
        !indCSG:(controllableSpeedGoal(vl,vh)) using csg by auto;
        note indStepC = andI(andI(andI(andI(andI(indVPos, indIsAnn),
  indOnAnn), indCSG), indCGDH), indCGDL);
        !indAssert:(J(xg, yg, k, v, vl, vh)) using indStepC by auto;
     case accPos:(a>=0) =>
        print("CASE k<=0,a>=0");
        t := 0;
        { xg'=-v*k*yg, yg'=v*(k*xg-1), v'=a, t'=1
        & ?dom:(v >= 0 & t <= T);
        & !tPos:(t>=0) by induction;
        & !dLo:((k*(eps*eps)-2*eps) < k*(sq(xg)+sq(yg))-2*xg)
  using isAnn onAnn by induction;
        & !dHi:(k*(xg*xg+yg*yg)-2*xg < (k*sq(eps)+2*eps))
  using isAnn onAnn by induction;
        & !vBound:(v+a*(T-t)>=0) using tPos noBrake by induction;
        & !bigL:(((a>=0&v>=vl)
                  |(1+2*eps*(-k)+sq(eps*k))*(A*(2*v*(T-t)+a*(sq(T-t)))
      +(sq(vl)-sq(v+a*(T-t))))<=2*A*(yg-eps)
                  |(1+2*eps*(-k)+sq(eps*k))*(A*(2*v*(T-t)+a*(sq(T-t)))
      +(sq(vl)-sq(v+a*(T-t))))<=2*A*((-xg)-eps)))
using constA dom tPos dLo dHi vBound accHi noBrake csgIvl csgHi uhp
```

```
isAnn onAnn accSgn kSgn admL accPos by induction;
        & !bigH:(((a>=0&v+a*(T-t)<=vh)
                  |(1+2*eps*(-k)+sq(eps*k))*(B*(2*v*(T-t)+a*(sq(T-t)))
      +(sq(v+a*(T-t))-sq(vh)))<=2*B*(yg-eps)
                  |(1+2*eps*(-k)+sq(eps*k))*(B*(2*v*(T-t)+a*(sq(T-t)))
      +(sq(v+a*(T-t))-sq(vh)))<=2*B*((-xg)-eps)))
using constB dom tPos dLo dHi vBound accRange noBrake csgIvl csgLo
uhp isAnn onAnn accSgn kSgn admH accPos by induction;};
        ?xgSgnC:(xg<=0);
        !indVPos:(v >= 0) using dom by auto;
        !indIsAnn:(isAnnulus(k)) using isAnn by auto;
        !indOnAnn:(onAnnulus(xg,yg,k)) using onAnn dLo dHi by auto;
        print("After ODE, CASE k<=0,a>=0");
        switch(bigL) {
          case loA:(a>=0& v>= vl) =>
            print("loA<>");
            !lem:(vl <= v) using loA by auto;
            !indCGDL:(controllableGoalDist(vl,v,A,xg,yg,k))
      using lem kSgn by auto;
          case loB:((1+2*eps*(-k)+sq(eps*k))*(A*(2*v*(T-t)+a*(sq(T-t)
    ))+(sq(vl)-sq(v+a*(T-t))))<=2*A*(yg-eps)) =>
            print("loB<>");
            !lem:((1 + 2*eps*(-k) + sq(eps*k)) * (sq(vl)-sq(v)) <=
      (2*A) * (yg-eps))
              using constA csgLo accRange vBound  dom  loB by auto;
            !indCGDL:(controllableGoalDist(vl,v,A,xg,yg,k))
      using lem kSgn xgSgnC by auto;
          case loC:((1+2*eps*(-k)+sq(eps*k))*(A*(2*v*(T-t)+a*(sq(T-t)
    ))+(sq(vl)-sq(v+a*(T-t))))<=2*A*((-xg)-eps)) =>
            print("loC<>");
            !lem:((1 + 2*eps*(-k) + sq(eps*k)) * (sq(vl)-sq(v)) <=
      (2*A) * (-xg-eps))
              using constA csgLo accRange vBound  dom  loC by auto;
            !indCGDL:(controllableGoalDist(vl,v,A,xg,yg,k))
      using lem kSgn xgSgnC by auto;}
        switch(bigH) {
          case hiA:(a>=0&v+a*(T-t)<=vh) =>
            print("hiA<>");
            !lem:(v <= vh)
            using hiA accPos dom tPos by auto;
            !indCGDH:(controllableGoalDist(v,vh,B,xg,yg,k))
      using lem kSgn by auto;
          case hiB:((1+2*eps*(-k)+sq(eps*k))*(B*(2*v*(T-t)+a*(sq(T-t))
    )+(sq(v+a*(T-t))-sq(vh)))<=2*B*(yg-eps)) =>
            print("hiB<>");
            !lem:((1 + 2*eps*(-k) + sq(eps*k)) * (sq(v)-sq(vh)) <=
```

```
      (2*B) * (yg-eps))
              using constB csgHi accRange vBound dom  hiB by auto;
            !indCGDH:(controllableGoalDist(v,vh,B,xg,yg,k))
      using lem kSgn xgSgnC by auto;
          case hiC:((1+2*eps*(-k)+sq(eps*k))*(B*(2*v*(T-t)+a*(sq(T-t))
    )+(sq(v+a*(T-t))-sq(vh)))<=2*B*((-xg)-eps)) =>
           print("hiC");
           !lem:((1 + 2*eps*(-k) + sq(eps*k)) * (sq(v)-sq(vh)) <=
      (2*B) * (-xg-eps))
              using constB csgHi accRange vBound dom hiC by auto;
            !indCGDH:(controllableGoalDist(v,vh,B,xg,yg,k))
      using lem kSgn xgSgnC by auto;}
        !indCSG:(controllableSpeedGoal(vl,vh)) using csg by auto;
        note indStepC = andI(andI(andI(andI(andI(indVPos, indIsAnn),
  indOnAnn), indCSG), indCGDH), indCGDL);
        !indAssert:(J(xg, yg, k, v, vl, vh)) using indStepC by auto;}
 case kPosAdm:(posBounds(xg,yg,v,vl,vh,a,k)) =>
   note kSgn = andEL(kPosAdm);
   note admL = andEL(andER(kPosAdm));
   note admH = andER(andER(kPosAdm));
   switch (accSgn) {
     case accNeg:(a<=0) =>
        print("CASE k>=0,a<=0");
        t := 0;
        { xg'=-v*k*yg, yg'=v*(k*xg-1), v'=a, t'=1
        & ?dom:(v >= 0 & t <= T);
        & !tPos:(t>=0) by induction;
        & !dLo:((k*(eps*eps)-2*eps) < k*(sq(xg)+sq(yg))-2*xg)
  using isAnn onAnn by induction;
        & !dHi:(k*(xg*xg+yg*yg)-2*xg < (k*sq(eps)+2*eps))
  using isAnn onAnn by induction;
        & !vBound:(v+a*(T-t)>=0) using tPos noBrake by induction;
        & !bigL:(((a<=0&vl<=v+a*(T-t))
                  |(1+2*eps*(k)+sq(eps*k))*(A*(2*v*(T-t)+a*(sq(T-t)))
      +(sq(vl)-sq(v+a*(T-t))))<=2*A*(yg-eps)
                  |(1+2*eps*(k)+sq(eps*k))*(A*(2*v*(T-t)+a*(sq(T-t)))
      +(sq(vl)-sq(v+a*(T-t))))<=2*A*((xg)-eps)))
           using constA dom tPos dLo dHi vBound accHi noBrake csgIvl
    csgHi uhp isAnn onAnn accSgn kSgn admL accNeg by induction;
        & !bigH:(((a<=0&v<=vh)
                  |(1+2*eps*(k)+sq(eps*k))*(B*(2*v*(T-t)+a*(sq(T-t)))+
      (sq(v+a*(T-t))-sq(vh)))<=2*B*(yg-eps)
                  |(1+2*eps*(k)+sq(eps*k))*(B*(2*v*(T-t)+a*(sq(T-t)))+
      (sq(v+a*(T-t))-sq(vh)))<=2*B*((xg)-eps)))
           using constB dom tPos dLo dHi vBound accRange noBrake csgIvl
csgLo uhp isAnn onAnn accSgn kSgn admH accNeg by induction;};
```

```
    ?xgSgnC:(xg>=0);
    !indVPos:(v >= 0) using dom by auto;
    !indIsAnn:(isAnnulus(k)) using isAnn by auto;
    !indOnAnn:(onAnnulus(xg,yg,k)) using onAnn dLo dHi by auto;
    print("After ODE, CASE k>=0,a<=0");
    switch(bigL) {
      case loA:((a<=0&vl<=v+a*(T-t))) =>
        print("loA><");
        !lem:(vl <= v) using loA accNeg dom tPos by auto;
        !indCGDL:(controllableGoalDist(vl,v,A,xg,yg,k))
  using lem kSgn by auto;
      case loB:((1+2*eps*(k)+sq(eps*k))*(A*(2*v*(T-t)+a*(sq(T-t)))
+(sq(vl)-sq(v+a*(T-t))))<=2*A*(yg-eps)) =>
        print("loB><");
        !lem:((1 + 2*eps*(k) + sq(eps*k)) * (sq(vl)-sq(v)) <=
  (2*A) * (yg-eps))
          using constA csgLo accRange vBound  dom  loB by auto;
        !indCGDL:(controllableGoalDist(vl,v,A,xg,yg,k))
  using lem kSgn xgSgnC by auto;
      case loC:((1+2*eps*(k)+sq(eps*k))*(A*(2*v*(T-t)+a*(sq(T-t)))
+(sq(vl)-sq(v+a*(T-t))))<=2*A*((xg)-eps)) =>
        print("loC><");
        !lem:((1 + 2*eps*(k) + sq(eps*k)) * (sq(vl)-sq(v)) <=
  (2*A) * (xg-eps))
          using constA csgLo accRange vBound  dom  loC by auto;
        !indCGDL:(controllableGoalDist(vl,v,A,xg,yg,k))
  using lem kSgn xgSgnC by auto;}
    switch(bigH) {
      case hiA:((a<=0&v<=vh)) =>
        print("hiA<<");
        !lem:(v <= vh) using hiA by prop;
        !indCGDH:(controllableGoalDist(v,vh,B,xg,yg,k))
  using lem kSgn by auto;
      case hiB:((1+2*eps*(k)+sq(eps*k))*(B*(2*v*(T-t)+a*(sq(T-t)))
+(sq(v+a*(T-t))-sq(vh)))<=2*B*(yg-eps)) =>
        print("hiB<<");
        !lem:((1 + 2*eps*(k) + sq(eps*k)) * (sq(v)-sq(vh)) <=
  (2*B) * (yg-eps))
          using constB csgHi accRange vBound dom  hiB by auto;
        !indCGDH:(controllableGoalDist(v,vh,B,xg,yg,k))
  using lem kSgn xgSgnC by auto;
      case hiC:((1+2*eps*(k)+sq(eps*k))*(B*(2*v*(T-t)+a*(sq(T-t)))
+(sq(v+a*(T-t))-sq(vh)))<=2*B*((xg)-eps)) =>
        print("hiC<<");
        !lem:((1 + 2*eps*(k) + sq(eps*k)) * (sq(v)-sq(vh)) <=
  (2*B) * (xg-eps))
```

```
              using constB csgHi accRange vBound dom hiC by auto;
            !indCGDH:(controllableGoalDist(v,vh,B,xg,yg,k))
      using lem kSgn xgSgnC by auto;}
        !indCSG:(controllableSpeedGoal(vl,vh)) using csg by auto;
        note indStepC = andI(andI(andI(andI(andI(indVPos, indIsAnn),
  indOnAnn), indCSG), indCGDH), indCGDL);
        !indAssert:(J(xg, yg, k, v, vl, vh)) using indStepC by auto;
     case accPos:(a>=0) =>
        print("CASE k>=0,a>=0");
        t := 0;
        { xg'=-v*k*yg, yg'=v*(k*xg-1), v'=a, t'=1
        & ?dom:(v >= 0 & t <= T);
        & !tPos:(t>=0) by induction;
        & !dLo:((k*(eps*eps)-2*eps) < k*(sq(xg)+sq(yg))-2*xg)
  using isAnn onAnn by induction;
        & !dHi:(k*(xg*xg+yg*yg)-2*xg < (k*sq(eps)+2*eps))
  using isAnn onAnn by induction;
        & !vBound:(v+a*(T-t)>=0) using tPos noBrake by induction;
        & !bigL:(((a>=0&v>=vl)
                  |(1+2*eps*(k)+sq(eps*k))*(A*(2*v*(T-t)+a*(sq(T-t)))+
      (sq(vl)-sq(v+a*(T-t))))<=2*A*(yg-eps)
                  |(1+2*eps*(k)+sq(eps*k))*(A*(2*v*(T-t)+a*(sq(T-t)))+
      (sq(vl)-sq(v+a*(T-t))))<=2*A*((xg)-eps)))
           using constA dom tPos dLo dHi vBound accHi noBrake csgIvl
    csgHi uhp isAnn onAnn accSgn kSgn admL accPos by induction;
        & !bigH:(((a>=0&v+a*(T-t)<=vh)
                  |(1+2*eps*(k)+sq(eps*k))*(B*(2*v*(T-t)+a*(sq(T-t)))+
      (sq(v+a*(T-t))-sq(vh)))<=2*B*(yg-eps)
                  |(1+2*eps*(k)+sq(eps*k))*(B*(2*v*(T-t)+a*(sq(T-t)))+
      (sq(v+a*(T-t))-sq(vh)))<=2*B*((xg)-eps)))
           using constB dom tPos dLo dHi vBound accRange noBrake
csgIvl csgLo uhp isAnn onAnn accSgn kSgn admH accPos by induction;};
        ?xgSgnC:(xg>=0);
        !indVPos:(v >= 0) using dom by auto;
        !indIsAnn:(isAnnulus(k)) using isAnn by auto;
        !indOnAnn:(onAnnulus(xg,yg,k)) using onAnn dLo dHi by auto;
        print("After ODE, CASE k<=0,a<=0");
        switch(bigL) {
          case loA:(a>=0&v>=vl) =>
            print("loA><");
            !lem:(vl <= v) using loA by auto;
            !indCGDL:(controllableGoalDist(vl,v,A,xg,yg,k))
      using lem kSgn by auto;
           case loB:((1+2*eps*(k)+sq(eps*k))*(A*(2*v*(T-t)+a*(sq(T-t)))
    +(sq(vl)-sq(v+a*(T-t))))<=2*A*(yg-eps)) =>
            print("loB><");
```

```
            !lem:((1 + 2*eps*(k) + sq(eps*k)) * (sq(vl)-sq(v)) <=
      (2*A) * (yg-eps))
              using constA csgLo accRange vBound dom  loB by auto;
            !indCGDL:(controllableGoalDist(vl,v,A,xg,yg,k))
      using lem kSgn xgSgnC by auto;
          case loC:((1+2*eps*(k)+sq(eps*k))*(A*(2*v*(T-t)+a*(sq(T-t)))
    +(sq(vl)-sq(v+a*(T-t))))<=2*A*((xg)-eps)) =>
            print("loC><");
            !lem:((1 + 2*eps*(k) + sq(eps*k)) * (sq(vl)-sq(v)) <=
      (2*A) * (xg-eps))
              using constA csgLo accRange vBound dom  loC by auto;
            !indCGDL:(controllableGoalDist(vl,v,A,xg,yg,k))
      using lem kSgn xgSgnC by auto;}
        switch(bigH) {
          case hiA:(a>=0&v+a*(T-t)<=vh) =>
            print("hiA><");
            !lem:(v <= vh)
            using hiA accPos dom tPos by auto;
            !indCGDH:(controllableGoalDist(v,vh,B,xg,yg,k))
      using lem kSgn by auto;
          case hiB:((1+2*eps*(k)+sq(eps*k))*(B*(2*v*(T-t)+a*(sq(T-t)))
    +(sq(v+a*(T-t))-sq(vh)))<=2*B*(yg-eps)) =>
            print("hiB><");
            !lem:((1 + 2*eps*(k) + sq(eps*k)) * (sq(v)-sq(vh))
      <= (2*B) * (yg-eps))
              using constB csgHi accRange vBound dom  hiB by auto;
            !indCGDH:(controllableGoalDist(v,vh,B,xg,yg,k))
      using lem kSgn xgSgnC by auto;
          case hiC:((1+2*eps*(k)+sq(eps*k))*(B*(2*v*(T-t)+a*(sq(T-t)))
    +(sq(v+a*(T-t))-sq(vh)))<=2*B*((xg)-eps)) =>
            print("hiC><");
            !lem:((1 + 2*eps*(k) + sq(eps*k)) * (sq(v)-sq(vh))
      <= (2*B) * (xg-eps))
              using constB csgHi accRange vBound dom hiC by auto;
            !indCGDH:(controllableGoalDist(v,vh,B,xg,yg,k))
      using lem kSgn xgSgnC by auto;}
        !indCSG:(controllableSpeedGoal(vl,vh)) using csg by auto;
        note indStepC = andI(andI(andI(andI(andI(indVPos, indIsAnn),
  indOnAnn), indCSG), indCGDH), indCGDL);
        !indAssert:(J(xg, yg, k, v, vl, vh)) using indStepC by auto;}}
 !fullIndStep:(J(xg, yg, k, v, vl, vh)) using indAssert by prop;}*
!safe:(sq(xg) + sq(yg) <= sq(eps) -> vl <= v & v <= vh)
using inv const by auto;
```

References

Aberth, O. (1971). The failure in computable analysis of a classical existence theorem for differential equations. *Proceedings of the American Mathematical Society*, *30*(1), 151–156. pages 209

Abramsky, S., Jagadeesan, R., & Malacaria, P. (2000). Full abstraction for PCF. *Inf. Comput.*, *163*(2), 409–470. doi: 10.1006/inco.2000.2930 pages 146

Abrial, J. (2010). *Modeling in Event-B: System and software engineering*. Cambridge University Press. doi: 10.1017/CBO9781139195881 pages 10, 226

Aczel, P., & Gambino, N. (2006). The generalised type-theoretic interpretation of constructive set theory. *J. Symb. Log.*, *71*(1), 67–103. doi: 10.2178/jsl/1140641163 pages 146

Aczel, P., & Rathjen, M. (2001). *Notes on constructive set theory* (Tech. Rep. No. 40). Institut Mittag-Leffler. Retrieved from `https://events.math.unipd.it/3wftop/pdf/AczelRathjen.pdf` (Revised June 18, 2008) pages 196, 197, 201, 428

Affeldt, R., & Cohen, C. (2017). Formal foundations of 3D geometry to model robot manipulators. In Y. Bertot & V. Vafeiadis (Eds.), *Certified Programs and Proofs* (pp. 30–42). ACM. doi: 10.1145/3018610.3018629 pages 13

Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P. H., & Ulbrich, M. (Eds.). (2016). *Deductive Software Verification - The KeY Book: From Theory to Practice* (Vol. 10001). Springer. doi: 10.1007/978-3-319-49812-6 pages 44, 252, 254

Akella, R., Tang, H., & McMillin, B. M. (2010). Analysis of information flow security in cyber-physical systems. *IJCIP*, *3*(4), 157–173. doi: 10.1016/j.ijcip.2010.09.001 pages 11

Alechina, N., Mendler, M., de Paiva, V., & Ritter, E. (2001). Categorical and Kripke semantics for constructive S4 modal logic. In L. Fribourg (Ed.), *CSL* (Vol. 2142, pp. 292–307). Springer. doi: 10.1007/3-540-44802-0_21 pages 146

Allen, S. F., Bickford, M., Constable, R. L., Eaton, R., Kreitz, C., Lorigo, L., & Moran, E. (2006). Innovations in computational type theory using Nuprl. *J. Applied Logic*, *4*(4), 428-469. (`http://www.nuprl.org/`) pages 81

Altenkirch, T., Dybjer, P., Hofmann, M., & Scott, P. J. (2001). Normalization by evaluation for typed lambda calculus with coproducts. In *LICS* (pp. 303–310). IEEE. doi: 10.1109/LICS.2001.932506 pages 182, 237

Althoff, M. (2013). Reachability analysis of nonlinear systems using conservative polynomialization and non-convex sets. In C. Belta & F. Ivancic (Eds.), *HSCC* (pp.

173–182). ACM. doi: 10.1145/2461328.2461358 pages 12

Althoff, M. (2015). An introduction to CORA 2015. In G. Frehse & M. Althoff (Eds.), *ARCH@CPSWeek* (Vol. 34, pp. 120–151). EasyChair. Retrieved from `https://easychair.org/publications/paper/xMm` pages 11

Althoff, M., & Dolan, J. M. (2014). Online verification of automated road vehicles using reachability analysis. *IEEE Trans. Robotics*, *30*(4), 903–918. doi: 10.1109/TRO.2014.2312453 pages 6, 12, 13, 19, 125

Alur, R., Henzinger, T. A., & Kupferman, O. (2002). Alternating-time temporal logic. *J. ACM*, *49*(5), 672–713. doi: 10.1145/585265.585270 pages 145

Alur, R., Henzinger, T. A., Lafferriere, G., & Pappas, G. J. (2000, July). Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, *88*(7), 971-984. doi: 10.1109/5.871304 pages 11

Anand, A., Appel, A., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Belanger, O. S., ... Weaver, M. (2017). CertiCoq: A verified compiler for Coq. In *CoqPL*. pages 6

Anand, A., & Knepper, R. A. (2015). ROSCoq: Robots powered by constructive reals. In C. Urban & X. Zhang (Eds.), *ITP* (Vol. 9236, pp. 34–50). Springer. doi: 10.1007/978-3-319-22102-1_3 pages 21

Anand, A., & Rahli, V. (2014). Towards a formally verified proof assistant. In G. Klein & R. Gamboa (Eds.), *ITP* (Vol. 8558, pp. 27–44). Springer. doi: 10.1007/978-3-319-08970-6_3 pages 59, 81

Andronick, J., & Felty, A. P. (Eds.). (2018). *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*. ACM. pages

Apt, K., De Boer, F. S., & Olderog, E.-R. (2010). *Verification of sequential and concurrent programs*. Springer Science & Business Media. pages 251, 254, 270

Apt, K. R., Bergstra, J. A., & Meertens, L. G. L. T. (1979). Recursive assertions are not enough - or are they? *Theor. Comput. Sci.*, *8*, 73–87. doi: 10.1016/0304-3975(79)90058-6 pages 254

Avigad, J., & Chlipala, A. (Eds.). (2016). *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*. ACM. Retrieved from `http://dl.acm.org/citation.cfm?id=2854065` pages

Avigad, J., & Feferman, S. (1998). Gödel's functional ("dialectica") interpretation. In S. R. Buss (Ed.), *Handbook of proof theory*. Elsevier. pages 209

Babaee, R., Gurfinkel, A., & Fischmeister, S. (2018). *P*revent : A predictive run-time verification framework using statistical learning. In E. B. Johnsen & I. Schaefer (Eds.), *Software Engineering and Formal Methods (SEFM), Held as Part of STAF* (Vol. 10886, pp. 205–220). Springer. doi: 10.1007/978-3-319-92970-5_13 pages 19

Back, R., & von Wright, J. (1998). *Refinement calculus - A systematic introduction*. Springer. doi: 10.1007/978-1-4612-1674-2 pages 224, 225

Baier, C., & Tinelli, C. (Eds.). (2015). *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015,*

London, UK, April 11-18, 2015. Proceedings (Vol. 9035). Springer. doi: 10.1007/978-3-662-46681-0 pages

Bak, S., Bogomolov, S., & Johnson, T. T. (2015). HYST: a source transformation and translation tool for hybrid automaton models. In A. Girard & S. Sankaranarayanan (Eds.), *HSCC* (pp. 128–133). ACM. doi: 10.1145/2728606.2728630 pages 11

Banach, R. (2013). Pliant modalities in Hybrid Event-B. In Z. Liu, J. Woodcock, & H. Zhu (Eds.), *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday* (Vol. 8051, pp. 37–53). Springer. doi: 10.1007/978-3-642-39698-4_3 pages 11, 226

Banach, R., Butler, M. J., Qin, S., Verma, N., & Zhu, H. (2015). Core Hybrid Event-B I: Single Hybrid Event-B machines. *Sci. Comput. Program.*, *105*, 92–123. doi: 10.1016/j.scico.2015.02.003 pages 11, 226

Bancerek, G., Bylinski, C., Grabowski, A., Kornilowicz, A., Matuszewski, R., Naumowicz, A., ... Urban, J. (2015). Mizar: State-of-the-art and beyond. In M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, & V. Sorge (Eds.), *Intelligent Computer Mathematics - International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings* (Vol. 9150, p. 261-279). Springer. doi: 10.1007/978-3-319-20615-8 pages 249

Barnett, M., Leino, K. R. M., & Schulte, W. (2005). The Spec# Programming System: An Overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, & T. Muntean (Eds.), *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: International Workshop, CASSIS 2004, Marseille, France, March 10-14, 2004, Revised Selected Papers* (pp. 49–69). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-540-30569-9_3 pages 254

Barras, B. (2010). Sets in Coq, Coq in sets. *J. Formalized Reasoning*, *3*(1), 29–48. doi: 10.6092/issn.1972-5787/1695 pages 6

Barras, B., & Werner, B. (1997). *Coq in Coq* (Tech. Rep.). Rocquencourt, France: INRIA Rocquencourt. pages 81, 250

Bartocci, E., Grosu, R., Karmarkar, A., Smolka, S. A., Stoller, S. D., Zadok, E., & Seyster, J. (2012). Adaptive runtime verification. In S. Qadeer & S. Tasiran (Eds.), *RV* (Vol. 7687, pp. 168–182). Springer. doi: 10.1007/978-3-642-35632-2_18 pages 19

Becker, H., Zyuzin, N., Monat, R., Darulova, E., Myreen, M. O., & Fox, A. C. J. (2018). A verified certificate checker for finite-precision error bounds in Coq and HOL4. In N. Bjørner & A. Gurfinkel (Eds.), *FMCAD* (pp. 1–10). IEEE. Retrieved from `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=8585253` doi: 10.23919/FMCAD.2018.8603019 pages 91

Beyer, D., & Huisman, M. (Eds.). (2018). *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I* (Vol. 10805). Springer. doi: 10.1007/978-3-319-89960-2 pages 91

Bhatia, A., Kavraki, L. E., & Vardi, M. Y. (2010). Motion planning with hybrid dynamics and temporal goals. In *CDC* (pp. 1108–1115). IEEE. Retrieved from `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp`

?punumber=5707200 doi: 10.1109/CDC.2010.5717440 pages 16

Bishop, E. (1967). *Foundations of constructive analysis*. New York: McGraw Hill. pages 146, 154, 196, 197, 211

Bishop, E., & Bridges, D. (2012). *Constructive analysis* (Vol. 279). Springer Science & Business Media. pages 197

Blanc, R., Kuncak, V., Kneuss, E., & Suter, P. (2013). An overview of the Leon verification system: verification by translation to recursive functions. In *Workshop on Scala, SCALA@ECOOP* (pp. 1:1–1:10). ACM. doi: 10.1145/2489837.2489838 pages 251

Blazy, S., Paulin-Mohring, C., & Pichardie, D. (Eds.). (2013). *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings* (Vol. 7998). Springer. doi: 10.1007/978-3-642-39634-2 pages

Bohrer, R. (2017). Differential dynamic logic. *Arch. Formal Proofs, 2017*. Retrieved from https://www.isa-afp.org/entries/Differential_Dynamic_Logic.shtml pages 27, 63

Bohrer, R., Fernández, M., & Platzer, A. (2019). dL$_\iota$: Definite descriptions in differential dynamic logic. In P. Fontaine (Ed.), *CADE* (Vol. 11716, pp. 94–110). Springer. doi: 10.1007/978-3-030-29436-6_6 pages 9, 33, 100, 203

Bohrer, R., & Platzer, A. (2018). A hybrid, dynamic logic for hybrid-dynamic information flow. In A. Dawar & E. Grädel (Eds.), *LICS* (p. 115-124). New York: ACM. doi: 10.1145/3209108.3209151 pages 9, 254, 339

Bohrer, R., & Platzer, A. (2019). Toward structured proofs for dynamic logics. *CoRR, abs/1908.05535*. Retrieved from http://arxiv.org/abs/1908.05535 pages 237, 250, 490

Bohrer, R., & Platzer, A. (2020a). Constructive game logic. In P. Müller (Ed.), *ESOP* (Vol. 12075, pp. 84–111). Springer. doi: 10.1007/978-3-030-44914-8_4 pages 191

Bohrer, R., & Platzer, A. (2020b). Constructive hybrid games. In N. Peltier & V. Sofronie-Stokkermans (Eds.), *IJCAR* (Vol. 12166, pp. 454–473). Springer. doi: 10.1007/978-3-030-51074-9_26 pages 209, 214, 215, 249

Bohrer, R., Rahli, V., Vukotic, I., Völp, M., & Platzer, A. (2017). Formally verified differential dynamic logic. In Y. Bertot & V. Vafeiadis (Eds.), *Certified Programs and Proofs* (p. 208-221). ACM. doi: 10.1145/3018610.3018616 pages 26, 27, 56, 91

Bohrer, R., Tan, Y. K., Mitsch, S., Myreen, M. O., & Platzer, A. (2018). VeriPhy: Verified controller executables from verified cyber-physical system models. In D. Grossman (Ed.), *PLDI* (p. 617-630). ACM. doi: 10.1145/3192366.3192406 pages 8, 15, 27, 69, 89, 112, 196, 249, 277, 292, 294, 305

Bohrer, R., Tan, Y. K., Mitsch, S., Sogokon, A., & Platzer, A. (2019). A formal safety net for waypoint following in ground robots. *IEEE Robotics and Automation Letters, 4*(3), 2910-2917. pages 89, 116, 123, 292, 294, 501

Boldo, S., Filliâtre, J., & Melquiond, G. (n.d.). Combining Coq and Gappa for certifying floating-point programs. In J. Carette, L. Dixon, C. S. Coen, & S. M. Watt (Eds.), *Intelligent computer mathematics (MKM), held as part of CICM.* pages 91

Boldo, S., Jourdan, J., Leroy, X., & Melquiond, G. (2013). A formally-verified C compiler supporting floating-point arithmetic. In A. Nannarelli, P. Seidel, & P. T. P. Tang

(Eds.), *IEEE Symposium on Computer Arithmetic (ARITH)* (pp. 107–115). IEEE. doi: 10.1109/ARITH.2013.30 pages 89

Boldo, S., & Melquiond, G. (2011). Flocq: A unified library for proving floating-point algorithms in Coq. In E. Antelo, D. Hough, & P. Ienne (Eds.), *IEEE Symposium on Computer Arithmetic (ARITH)* (pp. 243–252). IEEE Comp. Soc. doi: 10.1109/ARITH.2011.40 pages 91

Bouissou, O., Goubault, E., Putot, S., Tekkal, K., & Védrine, F. (2009). HybridFluctuat: A static analyzer of numerical programs within a continuous environment. In A. Bouajjani & O. Maler (Eds.), *CAV* (Vol. 5643, pp. 620–626). Springer. doi: 10.1007/978-3-642-02658-4_46 pages 91

Bourke, T., Brun, L., Dagand, P., Leroy, X., Pouzet, M., & Rieg, L. (2017). A formally verified compiler for Lustre. In A. Cohen & M. T. Vechev (Eds.), *PLDI* (pp. 586–601). ACM. doi: 10.1145/3062341.3062358 pages 90

Bridges, D. S., & Vita, L. S. (2007). *Techniques of constructive analysis*. Springer Science & Business Media. pages 146, 154, 196, 197, 211

Buss, S. R. (1998). An introduction to proof theory. *Handbook of proof theory*, *137*, 1–78. pages 177

Celani, S. A. (2001). A fragment of intuitionistic dynamic logic. *Fundam. Inform.*, *46*(3), 187–197. Retrieved from `http://content.iospress.com/articles/fundamenta-informaticae/fi46-3-01` pages 146

Chan, M., Ricketts, D., Lerner, S., & Malecha, G. (2016). Formal verification of stability properties of cyber-physical systems. In *CoqPL*. pages 13, 249, 277

Chatterjee, K., Henzinger, T. A., & Piterman, N. (2007). Strategy logic. In L. Caires & V. T. Vasconcelos (Eds.), *CONCUR* (Vol. 4703, pp. 59–73). Springer. doi: 10.1007/978-3-540-74407-8_5 pages 145

Chen, F., & Rosu, G. (2007). MOP: an efficient and generic runtime verification framework. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, & G. L. S. Jr. (Eds.), *OOPSLA* (pp. 569–588). ACM. doi: 10.1145/1297027.1297069 pages 19

Chen, X., Ábrahám, E., & Sankaranarayanan, S. (2013). Flow*: An analyzer for non-linear hybrid systems. In N. Sharygina & H. Veith (Eds.), *CAV* (Vol. 8044, pp. 258–263). Springer. doi: 10.1007/978-3-642-39799-8_18 pages 11

Chen, X., Schupp, S., Makhlouf, I. B., Ábrahám, E., Frehse, G., & Kowalewski, S. (2015). A benchmark suite for hybrid systems reachability analysis. In *NFM* (Vol. 9058). pages 12

Cimatti, A., Griggio, A., Mover, S., & Tonetta, S. (2014). Verifying LTL properties of hybrid systems with k-liveness. In A. Biere & R. Bloem (Eds.), *CAV* (Vol. 8559, pp. 424–440). Springer. doi: 10.1007/978-3-319-08867-9_28 pages 13

Cimatti, A., Griggio, A., Mover, S., & Tonetta, S. (2015). HyComp: An SMT-based model checker for hybrid systems. In C. Baier & C. Tinelli (Eds.), *TACAS* (Vol. 9035, pp. 52–67). Springer. doi: 10.1007/978-3-662-46681-0_4 pages 13

Clarke, E. M. (1980). Proving correctness of coroutines without history variables. *Acta Informatica*, *13*, 169–188. doi: 10.1007/BF00263992 pages 254

Clint, M. (1973). Program proving: Coroutines. *Acta Informatica*, *2*, 50–63. doi: 10.1007/BF00571463 pages 254

Cohen, C. (2012). *Formalized algebraic numbers: construction and first-order theory. (Formalisation des nombres algébriques : construction et théorie du premier ordre)* (Doctoral dissertation, École Polytechnique, Palaiseau, France). Retrieved from `https://tel.archives-ouvertes.fr/pastel-00780446` pages 72, 82

Cohen, C., & Mahboubi, A. (2012). Formal proofs in real algebraic geometry: from ordered fields to quantifier elimination. *Logical Methods in Computer Science*, *8*(1). doi: 10.2168/LMCS-8(1:2)2012 pages 290

Cohen, C., & Rouhling, D. (2017). A formal proof in Coq of LaSalle's invariance principle. In M. Ayala-Rincón & C. A. Muñoz (Eds.), *ITP* (Vol. 10499, pp. 148–163). Springer. doi: 10.1007/978-3-319-66107-0_10 pages 13

Cok, D. R. (2011). OpenJML: JML for Java 7 by extending OpenJDK. In M. G. Bobaru, K. Havelund, G. J. Holzmann, & R. Joshi (Eds.), *NFM* (Vol. 6617, pp. 472–479). Springer. doi: 10.1007/978-3-642-20398-5_35 pages 251

Collins, G. E. (1998). Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In B. F. Caviness & J. R. Johnson (Eds.), *Quantifier elimination and cylindrical algebraic decomposition* (pp. 85–121). Vienna: Springer. pages 82

Collins, G. E., & Hong, H. (1991, September). Partial cylindrical algebraic decomposition for quantifier elimination. *J. Symb. Comput.*, *12*(3), 299–328. doi: 10.1016/S0747-7171(08)80152-6 pages 13, 82, 102, 210, 290

Collins, P., & Graça, D. S. (2008). Effective computability of solutions of ordinary differential equations the thousand monkeys approach. *ENTCS*, *221*, 103–114. pages 209

Constable, R., Allen, S., Bromley, H., Cleaveland, W., Cremer, J., Harper, R., ... Smith, S. (1986). *Implementing mathematics with the Nuprl proof development system.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc. pages 59, 81

Coq Proof Assistant. (1989). Retrieved from `http://coq.inria.fr/` (Accessed: 2016-11-28) pages 141, 202, 228, 253

Coquand, T., & Huet, G. P. (1988). The calculus of constructions. *Inf. Comput.*, *76*(2/3), 95–120. doi: 10.1016/0890-5401(88)90005-3 pages 141, 202, 228

Coquand, T., & Paulin, C. (1988). Inductively defined types. In P. Martin-Löf & G. Mints (Eds.), *International Conference on Computer Logic (COLOG)* (Vol. 417, pp. 50–66). Springer. doi: 10.1007/3-540-52335-9_47 pages 202, 228

Corbineau, P. (2007). A declarative language for the Coq proof assistant. In M. Miculan, I. Scagnetto, & F. Honsell (Eds.), *TYPES* (Vol. 4941, pp. 69–84). Springer. doi: 10.1007/978-3-540-68103-8_5 pages 250

Cordwell, K., Tan, Y. K., & Platzer, A. (2021). A verified decision procedure for univariate real arithmetic with the BKR algorithm. In L. Cohen & C. Kaliszyk (Eds.), *ITP.* Schloss Dagstuhl - Leibniz-Zentrum für Informatik. (To appear.) pages 72, 82

Cousineau, D., Doligez, D., Lamport, L., Merz, S., Ricketts, D., & Vanzetto, H. (2012). TLA$^+$ proofs. In D. Giannakopoulou & D. Méry (Eds.), *FM* (Vol. 7436, pp. 147–154). Springer. doi: 10.1007/978-3-642-32759-9_14 pages 250, 252

Cousot, P., & Cousot, R. (1979). Constructive versions of Tarskis fixed point theorems. *Pacific Journal of Mathematics*, *82*(1), 43–57. pages 166

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., & Rival, X.

(2005). The ASTREÉ analyzer. In S. Sagiv (Ed.), *ESOP* (Vol. 3444, pp. 21–30). Springer. doi: 10.1007/978-3-540-31987-0_3 pages 11

Cruz-Filipe, L., Geuvers, H., & Wiedijk, F. (2004). C-CoRN, the constructive Coq repository at Nijmegen. In A. Asperti, G. Bancerek, & A. Trybulec (Eds.), *Mathematical Knowledge Management (MKM)* (Vol. 3119). Springer. Retrieved from `https://github.com/coq-community/corn` (Accessed: commits 9c44dae and 6411967) doi: 10.1007/978-3-540-27818-4_7 pages 196, 217, 436, 438

Cuijpers, P. J. L., & Reniers, M. A. (2005). Hybrid process algebra. *J. Log. Algebr. Program.*, *62*(2), 191–245. doi: 10.1016/j.jlap.2004.02.001 pages 10, 145

Curry, H. B., & Feys, R. (1958). *Combinatory logic* (Vol. 1). North-Holland. pages 141

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., & Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, *13*(4), 451–490. doi: 10.1145/115372.115320 pages 249

Daumas, M., Rideau, L., & Théry, L. (2001). A generic library for floating-point numbers and its application to exact computing. In R. J. Boulton & P. B. Jackson (Eds.), *TPHOLs* (Vol. 2152, pp. 169–184). Springer. doi: 10.1007/3-540-44755-5_13 pages 91

Davenport, J. H., & Heintz, J. (1988). Real quantifier elimination is doubly exponential. *J. Symb. Comput.*, *5*(1-2), 29–35. doi: 10.1016/S0747-7171(88)80004-X pages 82, 251

de Dinechin, F., Lauter, C. Q., & Melquiond, G. (2011). Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Trans. Computers*, *60*(2), 242–253. doi: 10.1109/TC.2010.128 pages 251

Degen, J., & Werner, J. (2006). Towards intuitionistic dynamic logic. *Logic and Logical Philosophy*, *15*(4), 305–324. pages 147, 189

Delahaye, D. (2000). A tactic language for the system Coq. In *LPAR* (pp. 85–95). Berlin, Heidelberg: Springer-Verlag. Retrieved from `http://dl.acm.org/citation.cfm?id=1765236.1765246` pages 252

Ding, J., & Tomlin, C. J. (2010). Robust reach-avoid controller synthesis for switched nonlinear systems. In *CDC* (pp. 6481–6486). IEEE. doi: 10.1109/CDC.2010.5717115 pages 194

Doberkat, E. (2011). Towards a coalgebraic interpretation of propositional dynamic logic. *CoRR*, *abs/1109.3685*. Retrieved from `http://arxiv.org/abs/1109.3685` pages 146

Dubins, L. E. (1957, July). On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *Amer. J. Math.*, *79*(3), 497+. doi: 10.2307/2372560 pages 115

Duggirala, P. S., Mitra, S., Viswanathan, M., & Potok, M. (2015). C2E2: A verification tool for stateflow models. In C. Baier & C. Tinelli (Eds.), *TACAS* (Vol. 9035, pp. 68–82). Springer. doi: 10.1007/978-3-662-46681-0_5 pages 11

Dupont, G., Ameur, Y. A., Pantel, M., & Singh, N. K. (2018). Proof-based approach to hybrid systems development: Dynamic logic and Event-B. In M. J. Butler, A. Raschke, T. S. Hoang, & K. Reichl (Eds.), *Abstract State Machines, Alloy, B, TLA, VDM, and*

Z (Vol. 10817, pp. 155–170). Springer. doi: 10.1007/978-3-319-91271-4_11 pages 11, 226

Dybjer, P. (1994). Inductive families. *Formal Asp. Comput.*, *6*(4), 440–465. doi: 10.1007/BF01211308 pages 202

Ehlers, R., & Finkbeiner, B. (2011). Monitoring realizability. In S. Khurshid & K. Sen (Eds.), *RV* (Vol. 7186, pp. 427–441). Springer. doi: 10.1007/978-3-642-29860-8_34 pages 19

Fainekos, G. E., Girard, A., Kress-Gazit, H., & Pappas, G. J. (2009). Temporal logic motion planning for dynamic robots. *Automatica*, *45*(2). doi: 10.1016/j.automatica.2008.08.008 pages 16

Fan, C., Mathur, U., Mitra, S., & Viswanathan, M. (2018). Controller synthesis made real: Reach-avoid specifications and linear dynamics. In H. Chockler & G. Weissenbacher (Eds.), *CAV* (Vol. 10981, pp. 347–366). Springer. doi: 10.1007/978-3-319-96145-3_19 pages 194

Fernández-Duque, D. (2018). The intuitionistic temporal logic of dynamical systems. *Logical Methods in Computer Science*, *14*(3). doi: 10.23638/LMCS-14(3:3)2018 pages 146

Filippidis, I., Dathathri, S., Livingston, S. C., Ozay, N., & Murray, R. M. (2016). Control design for hybrid systems with TuLiP: The temporal logic planning toolbox. In *CCA*. IEEE. Retrieved from `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7581720` doi: 10.1109/CCA.2016.7587949 pages 16, 195

Filliâtre, J., & Paskevich, A. (2013). Why3 - Where programs meet provers. In M. Felleisen & P. Gardner (Eds.), *ESOP* (Vol. 7792, pp. 125–128). Springer. doi: 10.1007/978-3-642-37036-6_8 pages 251

Finucane, C., Jing, G., & Kress-Gazit, H. (2010). LTLMoP: Experimenting with language, temporal logic and robot control. In *IROS*. IEEE. Retrieved from `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5639431` doi: 10.1109/IROS.2010.5650371 pages 16, 195

Fisac, J. F., Chen, M., Tomlin, C. J., & Sastry, S. S. (2015). Reach-avoid problems with time-varying dynamics, targets and constraints. In A. Girard & S. Sankaranarayanan (Eds.), *HSCC* (pp. 11–20). ACM. doi: 10.1145/2728606.2728612 pages 194

Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., & Stata, R. (2002). Extended static checking for Java. In J. Knoop & L. J. Hendren (Eds.), *PLDI* (pp. 234–245). ACM. doi: 10.1145/512529.512558 pages 251

Foster, J. N. (2010, March). *Bidirectional programming languages* (Tech. Rep. No. MS-CIS-10-08). Philadelphia, PA: Department of Computer & Information Science, University of Pennsylvania. pages 202

Foster, S. (2019). Hybrid relations in Isabelle/UTP. In P. Ribeiro & A. Sampaio (Eds.), *Unifying Theories of Programming* (Vol. 11885, pp. 130–153). Springer. doi: 10.1007/978-3-030-31038-7_7 pages 13

Fox, D., Burgard, W., & Thrun, S. (1997). The dynamic window approach to collision avoidance. *IEEE Robot. Automat. Mag.*, *4*(1). doi: 10.1109/100.580977 pages 15

Franchetti, F., Low, T. M., Mitsch, S., Mendoza, J. P., Gui, L., Phaosawasdi, A., ...

Veloso, M. (2017). High-assurance SPIRAL: End-to-end guarantees for robot and car control. *IEEE Control Systems*, *37*(2), 82-103. doi: 10.1109/MCS.2016.2643244 pages 19, 20

Frehse, G. (2005). PHAVer: Algorithmic verification of hybrid systems past HyTech. In M. Morari & L. Thiele (Eds.), *HSCC* (Vol. 3414, pp. 258–273). Springer. doi: 10.1007/978-3-540-31954-2_17 pages 12

Frehse, G., Guernic, C. L., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., ... Maler, O. (2011). SpaceEx: Scalable verification of hybrid systems. In G. Gopalakrishnan & S. Qadeer (Eds.), *CAV* (Vol. 6806, pp. 379–395). doi: 10.1007/978-3-642-22110-1_30 pages 11

Frittella, S., Greco, G., Kurz, A., Palmigiano, A., & Sikimic, V. (2016). A proof-theoretic semantic analysis of dynamic epistemic logic. *J. Log. Comput.*, *26*(6), 1961–2015. doi: 10.1093/logcom/exu063 pages 146

Fulton, N., Mitsch, S., Bohrer, R., & Platzer, A. (2017). Bellerophon: Tactical theorem proving for hybrid systems. In M. Ayala-Rincón & C. A. Muñoz (Eds.), *ITP* (Vol. 10499, p. 207-224). Springer. doi: 10.1007/978-3-319-66107-0_14 pages 15, 54, 92, 125, 247, 253

Fulton, N., Mitsch, S., Quesel, J.-D., Völp, M., & Platzer, A. (2015). KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In A. P. Felty & A. Middeldorp (Eds.), *CADE* (Vol. 9195, pp. 527–538). Springer. Retrieved from `https://lfcps.org/pub/KeYmaeraX-slides.pdf` (We list the URL for the presentation slides, which include on slide 16 the line counts for competing tools as reported by their developers. Accessed: January 29, 2021) doi: 10.1007/978-3-319-21401-6_36 pages 1, 12, 13, 14, 79, 92, 174, 253, 254, 264

Fulton, N., & Platzer, A. (2016). A logic of proofs for differential dynamic logic: Toward independently checkable proof certificates for dynamic logics. In J. Avigad & A. Chlipala (Eds.), *Certified Programs and Proofs* (p. 110-121). ACM. doi: 10.1145/2854065.2854078 pages 147

Ghilardi, S. (1989). Presheaf semantics and independence results for some non-classical first-order logics. *Arch. Math. Log.*, *29*(2), 125–136. doi: 10.1007/BF01620621 pages 146

Ghosh, S. (2008). Strategies made explicit in dynamic game logic. *Logic and the Foundations of Game and Decision Theory*, *6006*. pages 145

Gonthier, G., & Mahboubi, A. (2010). An introduction to small scale reflection in Coq. *J. Formalized Reasoning*, *3*(2), 95–152. doi: 10.6092/issn.1972-5787/1979 pages 250

Goranko, V. (2003). The basic algebra of game equivalences. *Studia Logica*, *75*(2), 221–238. doi: 10.1023/A:1027311011342 pages 146, 226, 227, 230, 232, 258

Grebing, S. (2019). *User interaction in deductive interactive program verification* (Doctoral dissertation, Karlsruhe Institute of Technology, Germany). Retrieved from `https://nbn-resolving.org/urn:nbn:de:101:1-20191030035842277609 22` pages 253

Greengard, S. (2015). Automotive systems get smarter. *Commun. ACM*, *58*(10), 18–20. doi: 10.1145/2811286 pages 1

Griffin, T. (1990). A formulae-as-types notion of control. In F. E. Allen (Ed.), *POPL* (pp.

47–58). ACM Press. Retrieved from http://dl.acm.org/citation.cfm?id=96709 doi: 10.1145/96709.96714 pages 147

Guéneau, A., Myreen, M. O., Kumar, R., & Norrish, M. (2017). Verified characteristic formulae for CakeML. In H. Yang (Ed.), *ESOP* (Vol. 10201, pp. 584–610). Springer. doi: 10.1007/978-3-662-54434-1_22 pages 89, 105, 109

Guernic, C. L. (2009). *Reachability analysis of hybrid systems with linear continuous dynamics. (Calcul d'atteignabilité des systèmes hybrides à partie continue linéaire)* (Doctoral dissertation, Joseph Fourier University, Grenoble, France). Retrieved from https://tel.archives-ouvertes.fr/tel-00422569 pages 11

Habermalz, E. (2000). *Interactive theorem proving with schematic theory specific rules.* Retrieved from https://publikationen.bibliothek.kit.edu/2272000 pages 252

Halbwachs, N., Lagnier, F., & Ratel, C. (1992). Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Trans. Software Eng.*, *18*(9), 785–793. doi: 10.1109/32.159839 pages 90

Harel, D., Tiuryn, J., & Kozen, D. (2000). *Dynamic logic.* Cambridge, MA, USA: MIT Press. pages 9, 11, 155, 163, 202, 206, 207

Harper, R. (2011). *The holy trinity.* Retrieved from https://web.archive.org/web/20170921012554/http://existentialtype.wordpress.com/2011/03/27/the-holy-trinity/ pages 141

Harper, R., Honsell, F., & Plotkin, G. D. (1993). A framework for defining logics. *J. ACM*, *40*(1), 143–184. doi: 10.1145/138027.138060 pages 81

Harrison, J. (1996). A Mizar mode for HOL. In J. von Wright, J. Grundy, & J. Harrison (Eds.), *TPHOLs* (Vol. 1125, pp. 203–220). Springer. doi: 10.1007/BFb0105406 pages 250

Harrison, J. (2006a). Floating-point verification using theorem proving. In M. Bernardo & A. Cimatti (Eds.), *Formal methods for hardware verification, international school on formal methods for the design of computer, communication, and software systems (SFM)* (Vol. 3965, pp. 211–242). Springer. doi: 10.1007/11757283_8 pages 91

Harrison, J. (2006b). Towards self-verification of HOL Light. In U. Furbach & N. Shankar (Eds.), *IJCAR* (Vol. 4130, p. 177-191). Springer. pages 81

Harrison, J. (2007). Verifying nonlinear real formulas via sums of squares. In K. Schneider & J. Brandt (Eds.), *TPHOLs* (Vol. 4732, pp. 102–118). Springer. doi: 10.1007/978-3-540-74591-4_9 pages 72, 82

Henzinger, T. A. (1996). The theory of hybrid automata. In *LICS* (pp. 278–292). IEEE Comp. Soc. Retrieved from http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4265 doi: 10.1109/LICS.1996.561342 pages 10, 12

Henzinger, T. A., Horowitz, B., & Majumdar, R. (1999). Rectangular hybrid games. In J. C. M. Baeten & S. Mauw (Eds.), *CONCUR* (Vol. 1664, pp. 320–335). Springer. doi: 10.1007/3-540-48320-9_23 pages 6, 16, 17, 195

Henzinger, T. A., Kopke, P. W., Puri, A., & Varaiya, P. (1998). What's decidable about hybrid automata? *J. Comput. Syst. Sci.*, *57*(1), 94–124. doi: 10.1006/jcss.1998.1581 pages 10, 15

Hickman, T., Laursen, C. P., & Foster, S. (2021). Certifying differential equation solutions from computer algebra systems in isabelle/hol. *CoRR*, *abs/2102.02679*. Retrieved from https://arxiv.org/abs/2102.02679 pages 13

Hilken, B. P., & Rydeheard, D. E. (1999). A first order modal logic and its sheaf models. In M. Fairtlough, M. Mendler, & E. Moggi (Eds.), *FLoC Satellite Workshop on Intuitionistic Modal Logics and Applications*. pages 146

Ho, S., Abrahamsson, O., Kumar, R., Myreen, M. O., Tan, Y. K., & Norrish, M. (2018). Proof-producing synthesis of CakeML with I/O and local state from monadic HOL functions. In D. Galmiche, S. Schulz, & R. Sebastiani (Eds.), *IJCAR* (Vol. 10900, pp. 646–662). Springer. doi: 10.1007/978-3-319-94205-6_42 pages 86, 87, 88

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, *12*(10), 576–580. doi: 10.1145/363235.363259 pages 141

Hoare, C. A. R. (1978). Communicating sequential processes. *Commun. ACM*, *21*(8), 666–677. doi: 10.1145/359576.359585 pages 9, 11, 260

Hoare, C. A. R., & He, J. (1998). *Unifying theories of programming*. Prentice Hall. pages 13

Hofmann, M., van Oosten, J., & Streicher, T. (2006). Well-foundedness in realizability. *Arch. Math. Log.*, *45*(7), 795–805. doi: 10.1007/s00153-006-0003-5 pages 158, 159, 176, 356

Hölzl, J., Immler, F., & Huffman, B. (2013). Type classes and filters for mathematical analysis in Isabelle/HOL. In S. Blazy, C. Paulin-Mohring, & D. Pichardie (Eds.), *ITP* (Vol. 7998, pp. 279–294). Springer. doi: 10.1007/978-3-642-39634-2_21 pages 43

Holzmann, G. J. (2013). Landing a spacecraft on Mars. *IEEE Softw.*, *30*(2), 83–86. doi: 10.1109/MS.2013.32 pages 1

Howard, W. A. (1980). The formulae-as-types notion of construction. *To H.B. Curry: essays on combinatory logic, lambda calculus and formalism*, *44*, 479–490. pages 141

Huang, J., Erdogan, C., Zhang, Y., Moore, B. M., Luo, Q., Sundaresan, A., & Rosu, G. (2014). ROSRV: runtime verification for robots. In B. Bonakdarpour & S. A. Smolka (Eds.), *RV* (Vol. 8734, pp. 247–254). Springer. doi: 10.1007/978-3-319-11164-3_20 pages 19

Huang, Z., Wang, Y., Mitra, S., Dullerud, G. E., & Chaudhuri, S. (2015). Controller synthesis with inductive proofs for piecewise linear systems: An SMT-based algorithm. In *CDC* (pp. 7434–7439). IEEE. doi: 10.1109/CDC.2015.7403394 pages 194

Huerta y Munive, J. J., & Struth, G. (2019). Predicate transformer semantics for hybrid systems: Verification components for isabelle/hol. *CoRR*, *abs/1909.05618*. Retrieved from http://arxiv.org/abs/1909.05618 pages 13

Hughes, G. E., & Cresswell, M. (1996). *A new introduction to modal logic*. Routledge. pages 224, 229

Hupel, L. (2019a). Private communication. pages 6, 73, 89

Hupel, L. (2019b). *Verified code generation from Isabelle/HOL* (Doctoral dissertation, Technical University of Munich, Germany). Retrieved from https://nbn-resolving.org/urn:nbn:de:bvb:91-diss-20190711-1473785-1-3 pages 73

Hupel, L., & Nipkow, T. (2018). A verified compiler from Isabelle/HOL to CakeML. In A. Ahmed (Ed.), *ESOP*. Springer. pages 6, 89

Immler, F. (2015). Verified reachability analysis of continuous systems. In *TACAS* (pp. 37–51). doi: 10.1007/978-3-662-46681-0_3 pages 13, 91

Immler, F., Rädle, J., & Wenzel, M. (2019). Virtualization of HOL4 in Isabelle. In J. Harrison, J. O'Leary, & A. Tolmach (Eds.), *ITP* (Vol. 141, pp. 21:1–21:18). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. doi: 10.4230/LIPIcs.ITP.2019.21 pages 85, 89

Immler, F., & Traut, C. (2016). The flow of ODEs. In J. C. Blanchette & S. Merz (Eds.), *ITP* (Vol. 9807, pp. 184–199). Springer. doi: 10.1007/978-3-319-43144-4_12 pages 13, 46, 91

Ioannidis, E., Kaashoek, M. F., & Zeldovich, N. (2019). Extracting and optimizing formally verified code for systems programming. In J. M. Badger & K. Y. Rozier (Eds.), *NFM* (Vol. 11460, pp. 228–236). Springer. doi: 10.1007/978-3-030-20652-9_15 pages 6

Jeannet, B. (2003). Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems. *Formal Methods in System Design*, *23*(1), 5–37. doi: 10.1023/A:1024480913162 pages 90

Jeannin, J., Ghorbal, K., Kouskoulas, Y., Gardner, R., Schmidt, A., Zawadzki, E., & Platzer, A. (2015). Formal verification of ACAS X, an industrial airborne collision avoidance system. In A. Girault & N. Guan (Eds.), *EMSOFT* (p. 127-136). IEEE Press. doi: 10.1109/EMSOFT.2015.7318268 pages 1

Jeannin, J., Ghorbal, K., Kouskoulas, Y., Schmidt, A., Gardner, R., Mitsch, S., & Platzer, A. (2017). A formally verified hybrid system for safe advisories in the next-generation airborne collision avoidance system. *STTT*, *19*(6), 717-741. doi: 10.1007/s10009-016-0434-1 pages 1, 5

Jones, C. B. (1991). *Systematic software development using VDM (2. ed.)*. Prentice Hall. pages 254

Kaliszyk, C., Pak, K., & Urban, J. (2016). Towards a Mizar environment for Isabelle: foundations and language. In J. Avigad & A. Chlipala (Eds.), *Certified Programs and Proofs* (pp. 58–65). ACM. Retrieved from `http://dl.acm.org/citation.cfm?id=2854065` doi: 10.1145/2854065.2854070 pages 250

Kamide, N. (2010). Strong normalization of program-indexed lambda calculus. *Bull. Sect. Logic Univ. Łódź*, *39*(1-2), 65–78. pages 147, 192

King, J. C. (1971). A program verifier. In C. V. Freiman, J. E. Griffith, & J. L. Rosenfeld (Eds.), *Information Processing, Proceedings of IFIP Congress* (pp. 234–249). North-Holland. pages 251

Kleene, S. C. (1938). On notation for ordinal numbers. *J. Symb. Log.*, *3*(4), 150–155. doi: 10.2307/2267778 pages 102

Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., ... Winwood, S. (2010). seL4: Formal verification of an operating-system kernel. *Commun. ACM*, *53*(6), 107–115. doi: 10.1145/1743546.1743574 pages 91, 250

Klein, G., & Nipkow, T. (2006). A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, *28*(4), 619–695. doi: 10.1145/1146811 pages 89

Kleymann, T. (1999). Hoare logic and auxiliary variables. *Formal Asp. Comput.*, *11*(5), 541–566. doi: 10.1007/s001650050057 pages 254

Kloetzer, M., & Belta, C. (2008). A fully automated framework for control of linear systems from temporal logic specifications. *IEEE Trans. Automat. Contr.*, *53*(1), 287–297. doi: 10.1109/TAC.2007.914952 pages 6, 16, 195

Kozen, D. (1997). Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, *19*(3), 427-443. pages 224

Krebbers, R., Timany, A., & Birkedal, L. (2017). Interactive proofs in higher-order concurrent separation logic. In G. Castagna & A. D. Gordon (Eds.), *POPL* (pp. 205–217). ACM. Retrieved from `http://dl.acm.org/citation.cfm?id=3009855` doi: 10.1145/3009837 pages 250

Kripke, S. A. (1972). Naming and necessity. In *Semantics of Natural Language* (pp. 253–355). Springer. pages 39

Kumar, R., Arthan, R., Myreen, M. O., & Owens, S. (2016). Self-formalisation of higher-order logic: Semantics, soundness, and a verified implementation. *J. Autom. Reasoning*, *56*(3), 221–259. doi: 10.1007/s10817-015-9357-x pages 6, 81, 86

Kumar, R., Myreen, M. O., Norrish, M., & Owens, S. (2014). CakeML: A verified implementation of ML. In S. Jagannathan & P. Sewell (Eds.), *POPL* (pp. 179–192). ACM. doi: 10.1145/2535838.2535841 pages 18, 81

Lamport, L. (1992). Hybrid systems in TLA$^+$. In R. L. Grossman, A. Nerode, A. P. Ravn, & H. Rischel (Eds.), *Hybrid systems* (Vol. 736, pp. 77–102). Springer. doi: 10.1007/3-540-57318-6_25 pages 21, 250, 252

Lamport, L. (1995). How to write a proof. *American Mathematical Monthly*, *102*(7), 600–608. Retrieved from `http://lamport.azurewebsites.net/pubs/lamport-how-to-write.pdf` (Accessed: January 29, 2021) pages 252

Lamport, L. (2012). How to write a 21st century proof. *Journal of Fixed Point Theory and Applications*. doi: 10.1007/s11784-012-0071-6 pages 252

Leavens, G. T., Baker, A. L., & Ruby, C. (1999). JML: A notation for detailed design. In H. Kilov, B. Rumpe, & I. Simmonds (Eds.), *Behavioral Specifications of Businesses and Systems* (Vol. 523, pp. 175–188). Springer. doi: 10.1007/978-1-4615-5229-1_12 pages 254

Lecomte, T., Déharbe, D., Prun, É., & Mottin, E. (2020). Applying a formal method in industry: a 25-year trajectory. *CoRR*, *abs/2005.07190*. Retrieved from `https://arxiv.org/abs/2005.07190` pages 11

Leino, K. R. M. (1998). Extended static checking. In D. Gries & W. P. de Roever (Eds.), *Programming Concepts and Methods (PROCOMET)* (Vol. 125, pp. 1–2). Chapman & Hall. pages 251

Leino, K. R. M. (2008, jun). This is Boogie 2. Microsoft Research. Retrieved from `https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/` (Accessed: January 29, 2021) pages 254

Leino, K. R. M. (2010). Dafny: An automatic program verifier for functional correctness. In E. M. Clarke & A. Voronkov (Eds.), *LPAR* (Vol. 6355, pp. 348–370). Springer. doi: 10.1007/978-3-642-17511-4_20 pages 251, 254

Leino, K. R. M., Müller, P., & Smans, J. (2009). Verification of concurrent programs with Chalice. In A. Aldini, G. Barthe, & R. Gorrieri (Eds.), *Foundations of Security Analysis and Design (FOSAD) Tutorial Lectures* (Vol. 5705, pp. 195–222). Springer. doi: 10.1007/978-3-642-03829-7_7 pages 254

Leroy, X. (2006). Formal certification of a compiler back-end *or*: Programming a compiler with a proof assistant. In J. G. Morrisett & S. L. P. Jones (Eds.), *POPL* (pp. 42–54). ACM. Retrieved from http://dl.acm.org/citation.cfm?id=1111037 doi: 10.1145/1111037.1111042 pages 89

Li, W., Passmore, G. O., & Paulson, L. C. (2019). Deciding univariate polynomial problems using untrusted certificates in Isabelle/HOL. *J. Autom. Reason.*, *62*(1), 69–91. doi: 10.1007/s10817-017-9424-6 pages 72, 82

Liebrenz, T., Herber, P., & Glesner, S. (2018). Deductive verification of hybrid control systems modeled in Simulink with KeYmaera X. In J. Sun & M. Sun (Eds.), *ICFEM* (Vol. 11232, pp. 89–105). Springer. doi: 10.1007/978-3-030-02450-5_6 pages 90

Lin, Q., Chen, X., Khurana, A., & Dolan, J. M. (2020). Reachflow: An online safety assurance framework for waypoint-following of self-driving cars. In *IROS*. IEEE. pages 13, 19

Lipton, J. (1992). Constructive Kripke semantics and realizability. In *Logic from Computer Science* (pp. 319–357). pages 146, 154

Litvintchouk, S. D., & Pratt, V. R. (1977). A proof-checker for dynamic logic. In R. Reddy (Ed.), *IJCAI* (pp. 552–558). William Kaufmann. Retrieved from http://ijcai.org/Proceedings/77-1/Papers/098.pdf pages 250

Liu, J., Lv, J., Quan, Z., Zhan, N., Zhao, H., Zhou, C., & Zou, L. (2010). A calculus for hybrid CSP. In K. Ueda (Ed.), *APLAS* (Vol. 6461, pp. 1–15). Springer. doi: 10.1007/978-3-642-17164-2_1 pages 10, 13, 14, 145

Lochbihler, A. (2007). Jinja with threads. *Archive of Formal Proofs*, *2007*. Retrieved from https://www.isa-afp.org/entries/JinjaThreads.shtml pages 250

Lombardi, H. (2020, November 15). Théories géométriques pour lalgébre des nombres réels sans test de signe ni axiome de choix dépendant. (Accessed: January 25, 2021. Unpublished draft (in French)) pages 211, 290

Loos, S. M. (2016). *Differential refinement logic* (Doctoral dissertation, Computer Science Department, School of Computer Science, Carnegie Mellon University). Retrieved from http://reports-archive.adm.cs.cmu.edu/anon/2016/abstracts/16-111.html pages 282

Loos, S. M., & Platzer, A. (2016). Differential refinement logic. In M. Grohe, E. Koskinen, & N. Shankar (Eds.), *LICS* (p. 505-514). ACM. doi: 10.1145/2933575.2934555 pages 224, 225, 226, 227, 229, 230, 232, 243

Loos, S. M., Platzer, A., & Nistor, L. (2011). Adaptive cruise control: Hybrid, distributed, and now formally verified. In M. Butler & W. Schulte (Eds.), *FM* (Vol. 6664, p. 42-56). Springer. doi: 10.1007/978-3-642-21437-0_6 pages 1

Low, T. M., & Franchetti, F. (2017). High assurance code generation for cyber-physical systems. In *HASE* (pp. 104–111). IEEE Comp. Soc. Retrieved from http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7911400 doi: 10.1109/HASE.2017.28 pages 20

MacKenzie, D. (1994). Computer-related accidental death: an empirical exploration. *Science and Public Policy*, *21*(4), 233–248. pages 5

Mahboubi, A. (2007). Implementing the cylindrical algebraic decomposition within the Coq system. *Math. Struct. Comput. Sci.*, *17*(1), 99–127. doi: 10.1017/S096012950600586X pages 72, 82

Majumdar, R., Saha, I., & Zamani, M. (2012). Synthesis of minimal-error control software. In A. Jerraya, L. P. Carloni, F. Maraninchi, & J. Regehr (Eds.), *EMSOFT* (pp. 123–132). ACM. doi: 10.1145/2380356.2380380 pages 91

Makarov, E., & Spitters, B. (2013). The Picard algorithm for ordinary differential equations in Coq. In S. Blazy, C. Paulin-Mohring, & D. Pichardie (Eds.), *ITP* (Vol. 7998, pp. 463–468). Springer. doi: 10.1007/978-3-642-39634-2_34 pages 13, 196, 198, 205, 206, 209, 214, 217, 222

Malecha, G., & Bengtson, J. (2015, January). Rtac: A fully reflective tactic language. In *CoqPL*. pages 252

Mamouras, K. (2016). Synthesis of strategies using the Hoare logic of angelic and demonic nondeterminism. *Logical Methods in Computer Science*, *12*(3). doi: 10.2168/LMCS-12(3:6)2016 pages 146

Martens, C., & Crary, K. (2012). LF in LF: Mechanizing the metatheories of LF in Twelf. In *Workshop on Logical Frameworks and Meta-Languages, Theory and Practice* (pp. 23–32). pages 81

Martin, B., Ghorbal, K., Goubault, E., & Putot, S. (2017). Formal verification of station keeping maneuvers for a planar autonomous hybrid system. In L. Bulwahn, M. Kamali, & S. Linker (Eds.), *Workshop on Formal Verification of Autonomous Vehicles (FVAV@iFM)* (Vol. 257, pp. 91–104). doi: 10.4204/EPTCS.257.9 pages 15

Martinez, A. A., Majumdar, R., Saha, I., & Tabuada, P. (2010). Automatic verification of control system implementations. In L. P. Carloni & S. Tripakis (Eds.), *EMSOFT* (pp. 9–18). ACM. doi: 10.1145/1879021.1879024 pages 91

Martínez, G., Ahman, D., Dumitrescu, V., Giannarakis, N., Hawblitzel, C., Hritcu, C., ... Swamy, N. (2019). Meta-F* : Proof automation with SMT, tactics, and metaprograms. In L. Caires (Ed.), *ESOP* (Vol. 11423, pp. 30–59). Springer. doi: 10.1007/978-3-030-17184-1_2 pages 251

Matichuk, D., Murray, T., & Wenzel, M. (2016, March). Eisbach: A proof method language for Isabelle. *J. Autom. Reason.*, *56*(3), 261–282. doi: 10.1007/s10817-015-9360-2 pages 252

McLaughlin, S., & Harrison, J. (2005). A proof-producing decision procedure for real arithmetic. In R. Nieuwenhuis (Ed.), *CADE* (Vol. 3632, pp. 295–314). Springer. doi: 10.1007/11532231_22 pages 72, 82

Melquiond, G. (2012). Floating-point arithmetic in the Coq system. *Inf. Comput.*, *216*, 14–23. doi: 10.1016/j.ic.2011.09.005 pages 91

Meredith, P. O., & Rosu, G. (2010). Runtime verification with the RV system. In H. Barringer et al. (Eds.), *RV* (Vol. 6418, pp. 136–152). Springer. doi: 10.1007/978-3-642-16612-9_12 pages 19

Miller, D., Nadathur, G., Pfenning, F., & Scedrov, A. (1991). Uniform proofs as a foundation for logic programming. *Ann. Pure Appl. Log.*, *51*(1-2), 125–157. doi:

10.1016/0168-0072(91)90068-W pages 290, 291

Mitsch, S., Ghorbal, K., & Platzer, A. (2013). On provably safe obstacle avoidance for autonomous robotic ground vehicles. In P. Newman, D. Fox, & D. Hsu (Eds.), *Robotics: Science and Systems.* pages 15

Mitsch, S., Ghorbal, K., Vogelbacher, D., & Platzer, A. (2017). Formal verification of obstacle avoidance and navigation of ground robots. *I. J. Robotics Res.*, *36*(12), 1312-1340. doi: 10.1177/0278364917733549 pages 1, 5, 15, 76, 249, 277, 282, 292, 294, 322, 323, 501

Mitsch, S., Passmore, G. O., & Platzer, A. (2014). Collaborative verification-driven engineering of hybrid systems. *Mathematics in Computer Science*, *8*(1), 71-97. doi: 10.1007/s11786-014-0176-y pages 5

Mitsch, S., & Platzer, A. (2014). ModelPlex: Verified runtime validation of verified cyber-physical system models. In B. Bonakdarpour & S. A. Smolka (Eds.), *RV* (Vol. 8734, p. 199-214). Springer. doi: 10.1007/978-3-319-11164-3_17 pages 18

Mitsch, S., & Platzer, A. (2016a). The KeYmaera X proof IDE: Concepts on usability in hybrid systems theorem proving. In C. Dubois, D. Mery, & P. Masci (Eds.), *3rd Workshop on Formal Integrated Development Environment* (Vol. 240, p. 67-81). doi: 10.4204/EPTCS.240.5 pages 92

Mitsch, S., & Platzer, A. (2016b). ModelPlex: Verified runtime validation of verified cyber-physical system models. *Formal Methods in System Design*, *49*(1), 33-74. (Special issue of selected papers from RV'14) doi: 10.1007/s10703-016-0241-z pages 1, 2, 6, 18, 85, 93, 94, 95, 196, 241, 299

Mitsch, S., & Platzer, A. (2018). Verified runtime validation for partially observable hybrid systems. *CoRR*, *abs/1811.06502*. Retrieved from `http://arxiv.org/abs/1811.06502` pages 18, 196

Mitsch, S., & Platzer, A. (2020). A retrospective on developing hybrid systems provers in the KeYmaera family - A tale of three provers. In W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, & M. Ulbrich (Eds.), *Deductive Software Verification: Future Perspectives - Reflections on the Occasion of 20 Years of KeY* (Vol. 12345, p. 21-64). Springer. doi: 10.1007/978-3-030-64354-6_2 pages 56

Mullen, E., Pernsteiner, S., Wilcox, J. R., Tatlock, Z., & Grossman, D. (2018). Œuf: minimizing the Coq extraction TCB. In J. Andronick & A. P. Felty (Eds.), *Certified Programs and Proofs* (pp. 172–185). ACM. doi: 10.1145/3167089 pages 6

Myreen, M. O., & Davis, J. (2014). The reflective Milawa theorem prover is sound - (Down to the machine code that runs it). In (pp. 421–436). doi: 10.1007/978-3-319-08970-6_27 pages 81

Myreen, M. O., & Owens, S. (2012). Proof-producing synthesis of ML from higher-order logic. In P. Thiemann & R. B. Findler (Eds.), *ICFP* (pp. 115–126). ACM. doi: 10.1145/2364527.2364545 pages 89, 91, 105

Myreen, M. O., Owens, S., & Kumar, R. (2013). Steps towards verified implementations of HOL Light. In *ITP* (pp. 490–495). pages 81

Narkawicz, A., Muñoz, C. A., & Dutle, A. (2015). Formally-verified decision procedures for univariate polynomial computation based on Sturm's and Tarski's theorems. *J. Autom. Reason.*, *54*(4), 285–326. doi: 10.1007/s10817-015-9320-x pages 72, 82

Nerode, A., & Kohn, W. (1993). Multiple agent hybrid control architecture. In R. L. Grossman, A. Nerode, A. P. Ravn, & H. Rischel (Eds.), *Hybrid systems* (pp. 297–316). Berlin, Heidelberg: Springer Berlin Heidelberg. pages 16

Nilsson, P., Hussien, O., Balkan, A., Chen, Y., Ames, A. D., Grizzle, J. W., ... Tabuada, P. (2016). Correct-by-construction adaptive cruise control: Two approaches. *IEEE Trans. Contr. Sys. Techn.*, *24*(4). doi: 10.1109/TCST.2015.2501351 pages 6, 16

Nipkow, T. (2002). Hoare Logics in Isabelle/HOL. In H. Schwichtenberg & R. Steinbrüggen (Eds.), *Proof and System-Reliability* (pp. 341–367). Dordrecht: Springer Netherlands. doi: 10.1007/978-94-010-0413-8_11 pages 250

Nipkow, T., & Klein, G. (2014). *Concrete semantics - With Isabelle/HOL*. Springer. doi: 10.1007/978-3-319-10542-0 pages 28

Nipkow, T., Paulson, L. C., & Wenzel, M. (2002). *Isabelle/HOL: A proof assistant for higher-order logic* (Vol. 2283). Springer. doi: 10.1007/3-540-45949-9 pages 28, 253

Osheim, E., & Switzer, T. (2011–). *Powerful new number types and numeric abstractions for Scala.* Retrieved from `https://github.com/typelevel/spire` (Accessed Oct. 13, 2020) pages 309, 318

Owicki, S. (1975). *Axiomatic proof techniques for parallel programs.* Garland Publishing, New York. pages 251, 254

Owicki, S., & Gries, D. (1976, December). An axiomatic proof technique for parallel programs I. *Acta Informatica*, *6*(4), 319–340. doi: 10.1007/BF00268134 pages 254

Padon, O., McMillan, K. L., Panda, A., Sagiv, M., & Shoham, S. (2016). Ivy: safety verification by interactive generalization. In C. Krintz & E. Berger (Eds.), *PLDI* (pp. 614–630). ACM. doi: 10.1145/2908080.2908118 pages 251

Parikh, R. (1983). Propositional game logic. In *FACS* (pp. 195–200). IEEE Comp. Soc. doi: 10.1109/SFCS.1983.47 pages 140, 141, 145, 154, 163, 164, 358

Pauly, M. (2002). A modal logic for coalitional power in games. *J. Log. Comput.*, *12*(1), 149–166. doi: 10.1093/logcom/12.1.149 pages 145

Pauly, M., & Parikh, R. (2003). Game logic - An overview. *Studia Logica*, *75*(2), 165–182. doi: 10.1023/A:1027354826364 pages 153, 192

Peleg, D. (1987). Concurrent dynamic logic. *J. ACM*, *34*(2), 450–479. doi: 10.1145/23005.23008 pages 147

Pereira, A., & Althoff, M. (2015). Safety control of robots under computed torque control using reachable sets. In *ICRA* (pp. 331–338). IEEE. Retrieved from `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7128761` doi: 10.1109/ICRA.2015.7139020 pages 13

Peterson, A., & Swanson, C. (2013). Modeling WALL·E and EVE's love. Term Project, Course 15-424/624/824, Carnegie Mellon University. Retrieved from `http://symbolaris.com/course/fcps13/projects/apeterso-cswanson.pdf` (Accessed: April 30, 2021) pages 272

Pfenning, F., & Schürmann, C. (1999). System description: Twelf - A meta-logical framework for deductive systems. In H. Ganzinger (Ed.), *CADE* (Vol. 1632, pp. 202–206). Springer. doi: 10.1007/3-540-48660-7_14 pages 81

Pinisetty, S., Jéron, T., Tripakis, S., Falcone, Y., Marchand, H., & Preoteasa, V. (2017).

Predictive runtime verification of timed properties. *J. Syst. Softw.*, *132*, 353–365. doi: 10.1016/j.jss.2017.06.060 pages 19

Pinto, A., Carloni, L. P., Passerone, R., & Sangiovanni-Vincentelli, A. L. (2006). Interchange format for hybrid systems: Abstract semantics. In J. P. Hespanha & A. Tiwari (Eds.), *HSCC* (Vol. 3927, pp. 491–506). Springer. doi: 10.1007/11730637_37 pages 11

Pit-Claudel, C., Wang, P., Delaware, B., Gross, J., & Chlipala, A. (2020). Extensible extraction of efficient imperative programs with foreign functions, manually managed memory, and proofs. In N. Peltier & V. Sofronie-Stokkermans (Eds.), *IJCAR* (Vol. 12167, pp. 119–137). Springer. Retrieved from `https://doi.org/10.1007/978-3-030-51054-1_7` doi: 10.1007/978-3-030-51054-1_7 pages 90

Platzer, A. (2007a). Differential dynamic logic for verifying parametric hybrid systems. In N. Olivetti (Ed.), *TABLEAUX* (Vol. 4548, p. 216-232). Springer. doi: 10.1007/978-3-540-73099-6_17 pages 25

Platzer, A. (2007b, Jun). Towards a hybrid dynamic logic for hybrid dynamic systems. In P. Blackburn, T. Bolander, T. Braüner, V. de Paiva, & J. Villadsen (Eds.), *International Workshop on Hybrid Logic (HyLo)* (Vol. 174, p. 63-77). doi: 10.1016/j.entcs.2006.11.026 pages 254

Platzer, A. (2008a). Differential dynamic logic for hybrid systems. *J. Autom. Reas.*, *41*(2), 143-189. doi: 10.1007/s10817-008-9103-8 pages 1, 10, 13, 14, 15, 25, 39, 92, 175, 178, 181, 234, 285, 352

Platzer, A. (2008b). *Differential dynamic logics: Automated theorem proving for hybrid systems* (Unpublished doctoral dissertation). Department of Computing Science, University of Oldenburg. pages 213

Platzer, A. (2010a). Differential-algebraic dynamic logic for differential-algebraic programs. *J. Log. Comput.*, *20*(1), 309-352. doi: 10.1093/logcom/exn070 pages 213, 214

Platzer, A. (2010b). *Logical analysis of hybrid systems: Proving theorems for complex dynamics*. Heidelberg: Springer. doi: 10.1007/978-3-642-14509-4 pages 1, 228

Platzer, A. (2011, November). *The complete proof theory of hybrid systems* (Tech. Rep. No. CMU-CS-11-144). Pittsburgh, PA: School of Computer Science, Carnegie Mellon University. pages 1

Platzer, A. (2012a). A complete axiomatization of quantified differential dynamic logic for distributed hybrid systems. *Logical Methods in Computer Science*, *8*(4), 1-44. (Special issue for selected papers from CSL'10) doi: 10.2168/LMCS-8(4:17)2012 pages 249, 254, 269, 271

Platzer, A. (2012b). The complete proof theory of hybrid systems. In *LICS* (p. 541-550). IEEE. doi: 10.1109/LICS.2012.64 pages 10, 15, 25, 178

Platzer, A. (2012c). Logics of dynamical systems. In *LICS* (p. 13-24). IEEE. doi: 10.1109/LICS.2012.13 pages 1, 14, 92

Platzer, A. (2012d). The structure of differential invariants and differential cut elimination. *Logical Methods in Computer Science*, *8*(4), 1-38. doi: 10.2168/LMCS-8(4:16)2012 pages 213, 228, 234, 270

Platzer, A. (2015a). Differential game logic. *ACM Trans. Comput. Log.*, *17*(1), 1:1–1:51. doi: 10.1145/2817824 pages 3, 14, 141, 145, 147, 150, 155, 163, 164, 166, 167, 178,

180, 194, 213, 223, 225, 358, 359, 431, 472

Platzer, A. (2015b). A uniform substitution calculus for differential dynamic logic. In A. P. Felty & A. Middeldorp (Eds.), *CADE* (Vol. 9195, p. 467-481). Springer. doi: 10.1007/978-3-319-21401-6_32 pages 32, 48, 76

Platzer, A. (2016). Logic & proofs for cyber-physical systems. In N. Olivetti & A. Tiwari (Eds.), *IJCAR* (Vol. 9706, p. 15-21). Springer. doi: 10.1007/978-3-319-40229-1_3 pages 15

Platzer, A. (2017a). A complete uniform substitution calculus for differential dynamic logic. *J. Autom. Reas.*, *59*(2), 219-265. doi: 10.1007/s10817-016-9385-1 pages 1, 10, 14, 15, 25, 32, 34, 39, 41, 42, 47, 48, 52, 54, 57, 59, 68, 73, 76, 77, 78, 79, 81, 92, 127, 180, 181, 213, 228, 253, 263, 283, 387, 439, 442

Platzer, A. (2017b). Differential hybrid games. *ACM Trans. Comput. Log.*, *18*(3), 19:1-19:44. doi: 10.1145/3091123 pages 145, 163, 164, 224

Platzer, A. (2018a). *Logical foundations of cyber-physical systems*. Springer. doi: 10.1007/978-3-319-63588-0 pages 1, 10, 25, 52, 74, 227, 228, 270, 271

Platzer, A. (2018b). Uniform substitution for differential game logic. In D. Galmiche, S. Schulz, & R. Sebastiani (Eds.), *IJCAR* (Vol. 10900, p. 211-227). Springer. doi: 10.1007/978-3-319-94205-6_15 pages 215, 357, 439, 445

Platzer, A. (2019a). Differential game logic. *Archive of Formal Proofs*, *2019*. Retrieved from `https://www.isa-afp.org/entries/Differential_Game_Logic.html` pages 26, 59, 63, 69, 81

Platzer, A. (2019b). Uniform substitution at one fell swoop. In P. Fontaine (Ed.), *CADE* (Vol. 11716, p. 425-441). Springer. doi: 10.1007/978-3-030-29436-6_25 pages 439

Platzer, A., & Clarke, E. M. (2009). Formal verification of curved flight collision avoidance maneuvers: A case study. In A. Cavalcanti & D. Dams (Eds.), *FM* (Vol. 5850, p. 547-562). Springer. doi: 10.1007/978-3-642-05089-3_35 pages 278

Platzer, A., & Quesel, J.-D. (2008a). KeYmaera: A hybrid theorem prover for hybrid systems. In A. Armando, P. Baumgartner, & G. Dowek (Eds.), *IJCAR* (Vol. 5195, p. 171-178). Springer. doi: 10.1007/978-3-540-71070-7_15 pages 1, 13, 14, 253

Platzer, A., & Quesel, J.-D. (2008b). Logical verification and systematic parametric analysis in train control. In M. Egerstedt & B. Mishra (Eds.), *HSCC* (Vol. 4981, p. 646-649). Springer. doi: 10.1007/978-3-540-78929-1_55 pages 1

Platzer, A., & Quesel, J.-D. (2009). European Train Control System: A case study in formal verification. In K. Breitman & A. Cavalcanti (Eds.), *ICFEM* (Vol. 5885, p. 246-265). Springer. doi: 10.1007/978-3-642-10373-5_13 pages 110

Platzer, A., Quesel, J.-D., & Rümmer, P. (2009). Real world verification. In R. A. Schmidt (Ed.), *CADE* (Vol. 5663, p. 485-501). Springer. doi: 10.1007/978-3-642-02959-2_35 pages 72, 82

Platzer, A., & Tan, Y. K. (2018). Differential equation axiomatization: The impressive power of differential ghosts. In A. Dawar & E. Grädel (Eds.), *LICS* (p. 819-828). New York: ACM. doi: 10.1145/3209108.3209147 pages 249, 254, 269, 271

Platzer, A., & Tan, Y. K. (2020). Differential equation invariance axiomatization. *J. ACM*, *67*(1), 6:1–6:66. doi: 10.1145/3380825 pages 15, 52, 74, 92, 213, 214, 228, 234, 249, 254, 269, 271

Prabhakar, P., & Köpf, B. (2013). Verifying information flow properties of hybrid systems. In L. Bushnell, L. Rohrbough, S. Amin, & X. D. Koutsoukos (Eds.), *HiCoNS (part of CPSweek)* (pp. 77–84). ACM. Retrieved from http://dl.acm.org/citation.cfm?id=2461446 doi: 10.1145/2461446.2461458 pages 10

Pratt, V. R. (1976). Semantical considerations on Floyd-Hoare logic. In *FOCS* (pp. 109–121). IEEE Comp. Soc. doi: 10.1109/SFCS.1976.27 pages 141

Püschel, M., Moura, J. M. F., Johnson, J. R., Padua, D. A., Veloso, M. M., Singer, B., ... Rizzolo, N. (2005). SPIRAL: code generation for DSP transforms. *Proc. IEEE*, *93*(2), 232–275. doi: 10.1109/JPROC.2004.840306 pages 19

Quesel, J.-D. (2013). *Similarity, logic, and games - Bridging modeling layers of hybrid systems* (Doctoral dissertation, Universität Oldenburg). Retrieved from http://www.cs.cmu.edu/~jquesel/paper/diss.pdf (Accessed: January 24, 2021) pages 14, 145

Quesel, J.-D., Mitsch, S., Loos, S., Aréchiga, N., & Platzer, A. (2016). How to model and prove hybrid systems with KeYmaera: A tutorial on safety. *STTT*, *18*(1), 67-91. doi: 10.1007/s10009-015-0367-0 pages 110

Quesel, J.-D., & Platzer, A. (2012). Playing hybrid games with KeYmaera. In B. Gramlich, D. Miller, & U. Sattler (Eds.), *IJCAR* (Vol. 7364, p. 439-453). Springer. doi: 10.1007/978-3-642-31365-3_34 pages 194

Rabin, M. O., & Scott, D. S. (1959). Finite automata and their decision problems. *IBM J. Res. Dev.*, *3*(2), 114–125. doi: 10.1147/rd.32.0114 pages 9, 11

Rahli, V., & Bickford, M. (2016). A nominal exploration of intuitionism. In J. Avigad & A. Chlipala (Eds.), (pp. 130–141). ACM. Retrieved from http://dl.acm.org/citation.cfm?id=2854065 doi: 10.1145/2854065.2854077 pages 59, 81

Ramanujam, R., & Simon, S. E. (2008). Dynamic logic on games with structured strategies. In G. Brewka & J. Lang (Eds.), *Principles of Knowledge Representation and Reasoning: Proceedings of the Eleventh International Conference, KR 2008, Sydney, Australia, September 16-19, 2008* (pp. 49–58). AAAI Press. Retrieved from http://www.aaai.org/Library/KR/2008/kr08-006.php pages 145

Reif, W. (1995). The KIV-approach to software verification. In *KORSO: Methods, Languages, and Tools for the Construction of Correct Software* (pp. 339–368). Springer. pages 252

Ricketts, D. (2017). *Verification of sampled-data systems using Coq* (Doctoral dissertation, University of California, San Diego, USA). Retrieved from http://www.escholarship.org/uc/item/5n1899s2 pages 21

Ricketts, D., Malecha, G., Alvarez, M. M., Gowda, V., & Lerner, S. (2015). Towards verification of hybrid systems in a foundational proof assistant. In *MEMOCODE 2015* (pp. 248–257). IEEE. Retrieved from http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7329076 doi: 10.1109/MEMCOD.2015.7340492 pages 21

Ringer, T., Palmskog, K., Sergey, I., Gligoric, M., & Tatlock, Z. (2019). QED at large: A survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages*, *5*(2-3), 102–281. pages 5

Rizaldi, A., Immler, F., Schürmann, B., & Althoff, M. (2018). A formally verified motion

planner for autonomous vehicles. In *ATVA* (Vol. 11138). Springer. doi: 10.1007/978-3-030-01090-4_5 pages 13

Rizaldi, A., Keinholz, J., Huber, M., Feldle, J., Immler, F., Althoff, M., ... Nipkow, T. (2017). Formalising and monitoring traffic rules for autonomous vehicles in Isabelle/HOL. In N. Polikarpova & S. Schneider (Eds.), *IFM* (Vol. 10510, pp. 50–66). Springer. doi: 10.1007/978-3-319-66845-1_4 pages 13

Robinson, J. (1949). Definability and decision problems in arithmetic. *J. Symb. Log.*, *14*(2), 98–114. doi: 10.2307/2266510 pages 177

Roehm, H., Oehlerking, J., Woehrle, M., & Althoff, M. (2019). Model conformance for cyber-physical systems: A survey. *ACM Trans. Cyber Phys. Syst.*, *3*(3), 30:1–30:26. doi: 10.1145/3306157 pages 19

Rouhling, D. (2018). A formal proof in Coq of a control function for the inverted pendulum. In J. Andronick & A. P. Felty (Eds.), *Certified Programs and Proofs* (pp. 28–41). ACM. doi: 10.1145/3167101 pages 13

Ruchkin, I., Sunshine, J., Iraci, G., Schmerl, B. R., & Garlan, D. (2018). IPL: an integration property language for multi-model cyber-physical systems. In K. Havelund, J. Peleska, B. Roscoe, & E. P. de Vink (Eds.), *FM* (Vol. 10951, pp. 165–184). Springer. doi: 10.1007/978-3-319-95582-7_10 pages 11

Schiffelers, R. R. H., van Beek, D. A., Man, K. L., Reniers, M. A., & Rooda, J. E. (2003). Formal semantics of hybrid Chi. In K. G. Larsen & P. Niebert (Eds.), *Formal Modeling and Analysis of Timed Systems* (Vol. 2791, pp. 151–165). Springer. doi: 10.1007/978-3-540-40903-8_12 pages 10, 145

Seto, D., Krogh, B., Sha, L., & Chutinan, A. (1998, June). The SIMPLEX architecture for safe on-line control system upgrades. In *American Control Conference.* doi: 10.1109/ACC.1998.703255 pages 18, 299

Sewell, T. A. L., Myreen, M. O., & Klein, G. (2013). Translation validation for a verified OS kernel. In H. Boehm & C. Flanagan (Eds.), *PLDI* (pp. 471–482). ACM. doi: 10.1145/2462156.2462183 pages 90

Shah, S., Dey, D., Lovett, C., & Kapoor, A. (2018). AirSim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field Serv. Robot.* (Vol. 5, pp. 621–635). pages 91, 115, 122

Shakernia, O., Pappas, G. J., & Sastry, S. (2001). Semi-decidable synthesis for triangular hybrid systems. In M. D. D. Benedetto & A. L. Sangiovanni-Vincentelli (Eds.), *HSCC* (Vol. 2034, pp. 487–500). Springer. doi: 10.1007/3-540-45351-2_39 pages 17, 195

Shakernia, O., Sastry, S., & Pappas, G. J. (2000). Decidable controller synthesis for classes of linear systems. In N. A. Lynch & B. H. Krogh (Eds.), *HSCC* (Vol. 1790, pp. 407–420). Springer. doi: 10.1007/3-540-46430-1_34 pages 17, 195

Simulink documentation. (2021). Retrieved from `https://www.mathworks.com/help/simulink/` (Accessed: January 24, 2021) pages 84

Sogokon, A., Mitsch, S., Tan, Y. K., Cordwell, K., & Platzer, A. (2019). Pegasus: A framework for sound continuous invariant generation. In M. H. ter Beek, A. McIver, & J. N. Oliveira (Eds.), *FM* (Vol. 11800, pp. 138–157). Springer. doi: 10.1007/978-3-030-30942-8_10 pages 13, 54

Stampoulis, A., & Shao, Z. (2010). VeriML: typed computation of logical terms inside a

language with effects. In P. Hudak & S. Weirich (Eds.), *ICFP* (pp. 333–344). ACM. doi: 10.1145/1863543.1863591 pages 252

Stampoulis, A., & Shao, Z. (2012). Static and user-extensible proof checking. In J. Field & M. Hicks (Eds.), *POPL* (pp. 273–284). ACM. Retrieved from http://dl.acm.org/citation.cfm?id=2103656 doi: 10.1145/2103656.2103690 pages 252

Stateflow documentation. (2021). Retrieved from https://www.mathworks.com/help/stateflow/ (Accessed: January 24, 2021) pages 84

Sterling, J., & Harper, R. (2018). Guarded computational type theory. In A. Dawar & E. Grädel (Eds.), *LICS* (pp. 879–888). ACM. Retrieved from https://doi.org/10.1145/3209108.3209153 doi: 10.1145/3209108.3209153 pages 196

Strzebonski, A. W. (2006). Cylindrical algebraic decomposition using validated numerics. *J. Symb. Comput.*, *41*(9), 1021–1038. doi: 10.1016/j.jsc.2006.06.004 pages 82

Su, W., Abrial, J., & Zhu, H. (2014). Formalizing hybrid systems with Event-B and the Rodin platform. *Sci. Comput. Program.*, *94*, 164–202. doi: 10.1016/j.scico.2014.04.015 pages 11, 226

Suenaga, K., & Hasuo, I. (2011). Programming with infinitesimals: A while-language for hybrid system modeling. In L. Aceto, M. Henzinger, & J. Sgall (Eds.), *Automata, Languages and Programming - International Colloquium, (ICALP)* (Vol. 6756, pp. 392–403). Springer. doi: 10.1007/978-3-642-22012-8_31 pages 10

Swamy, N., Hritcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., ... Zanella-Béguelin, S. (2016, January). Dependent types and multi-monadic effects in F*. In *POPL* (p. 256-270). ACM. Retrieved from https://www.fstar-lang.org/papers/mumon/ doi: 10.1145/2837614.2837655 pages 251

Syme, D. (1997). *DECLARE: A prototype declarative proof system for higher order logic* (Tech. Rep. No. 416). Cambridge, UK: Computer Laboratory, University of Cambridge. pages 250

Taly, A., & Tiwari, A. (2010). Switching logic synthesis for reachability. In L. P. Carloni & S. Tripakis (Eds.), *EMSOFT* (pp. 19–28). ACM. doi: 10.1145/1879021.1879025 pages 6, 16, 195

Tan, Y. K., Myreen, M. O., Kumar, R., Fox, A. C. J., Owens, S., & Norrish, M. (2016). A new verified compiler backend for CakeML. In J. Garrigue, G. Keller, & E. Sumii (Eds.), *ICFP* (pp. 60–73). ACM. doi: 10.1145/2951913.2951924 pages 20, 88, 89, 105, 110

Tan, Y. K., & Platzer, A. (2019). An axiomatic approach to liveness for differential equations. In M. ter Beek, A. McIver, & J. N. Oliviera (Eds.), *FM* (Vol. 11800, p. 371-388). Springer. doi: 10.1007/978-3-030-30942-8_23 pages 214, 237

Tarski, A. (1951). A decision method for elementary algebra and geometry. In B. F. Caviness & J. R. Johnson (Eds.), *Quantifier elimination and cylindrical algebraic decomposition* (pp. 24–84). Vienna: Springer. pages 81, 82, 211, 251, 290

Tomlin, C. J., Lygeros, J., & Sastry, S. S. (2000). A game theoretic approach to controller design for hybrid systems. *Proc. IEEE*, *88*(7), 949–970. pages 6, 16, 17, 194, 195

Toom, A., Naks, T., Pantel, M., Gandriau, M., & Wati, I. (2008, January). Gene-Auto: an Automatic Code Generator for a safe subset of Simulink/Stateflow and Scicos. In *Embedded Real Time Software and Systems (ERTS).* Toulouse, France. Retrieved

from https://hal.archives-ouvertes.fr/hal-02270306 pages 6, 90

Tripakis, S., Sofronis, C., Caspi, P., & Curic, A. (2005). Translating discrete-time Simulink to Lustre. *ACM Trans. Embed. Comput. Syst.*, *4*(4), 779–818. doi: 10.1145/1113830.1113834 pages 90

Vallado, D. A. (1997). *Fundamentals of Astrodynamics and Applications*. McGraw-Hill Primis. pages 278

Van Benthem, J. (2001). Games in dynamic-epistemic logic. *Bulletin of Economic Research*, *53*(4), 219–248. pages 145

van Benthem, J. (2015). Logic of strategies: What and how? In J. van Benthem, S. Ghosh, & R. Verbrugge (Eds.), *Models of Strategic Reasoning - Logics, Games, and Communities* (Vol. 8972, pp. 321–332). Springer. doi: 10.1007/978-3-662-48540-8_10 pages 145

van Benthem, J., & Bezhanishvili, G. (2007). Modal logics of space. In M. Aiello, I. Pratt-Hartmann, & J. van Benthem (Eds.), *Handbook of Spatial Logics* (pp. 217–298). Springer. doi: 10.1007/978-1-4020-5587-4_5 pages 146

van Benthem, J., Bezhanishvili, N., & Enqvist, S. (2017). A Propositional Dynamic Logic for Instantial Neighborhood Models. In A. Baltag, J. Seligman, & T. Yamada (Eds.), *Logic, Rationality, and Interaction* (Vol. 10455, pp. 137–150). Springer. doi: 10.1007/978-3-662-55665-8_10 pages 146

van Benthem, J., & Pacuit, E. (2011). Dynamic logics of evidence-based beliefs. *Studia Logica*, *99*(1-3), 61–92. doi: 10.1007/s11225-011-9347-x pages 145

van Benthem, J., Pacuit, E., & Roy, O. (2011). Toward a theory of play: A logical perspective on games and interaction. *Games*, *2*(1), 52–86. doi: 10.3390/g2010052 pages 145

van der Hoek, W., Jamroga, W., & Wooldridge, M. J. (n.d.). A logic for strategic reasoning. In *Autonomous Agents and Multiagent Systems.* pages 145

Vanderwaart, J., Dreyer, D., Petersen, L., Crary, K., Harper, R., & Cheng, P. (2003). Typed compilation of recursive datatypes. In Z. Shao & P. Lee (Eds.), *Types in Language Design and Implementation* (pp. 98–108). ACM. Retrieved from http://dl.acm.org/citation.cfm?id=604174 doi: 10.1145/604174.604187 pages 173

van Oosten, J. (2002). Realizability: A historical essay. *Math. Struct. Comput. Sci.*, *12*(3), 239–263. doi: 10.1017/S0960129502003626 pages 146, 154

Wagner, M., & Norris, G. (2009). *Boeing 787 Dreamliner*. Zenith Press. pages 1

Walter, W. (1998). *Ordinary differential equations* (Vol. 182). New York: Springer. doi: 10.1007/978-1-4612-0601-9 pages 438

Weihrauch, K. (2000). *Computable analysis - an introduction*. Springer. doi: 10.1007/978-3-642-56999-9 pages 154, 197

Weispfenning, V. (1997). Quantifier elimination for real algebra - the quadratic case and beyond. *Appl. Algebra Eng. Commun. Comput.*, *8*(2), 85–101. doi: 10.1007/s002000050055 pages 251

Wenzel, M. (1999). Isar - A generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin-Mohring, & L. Théry

(Eds.), *TPHOLs* (Vol. 1690, pp. 167–184). Springer. doi: 10.1007/3-540-48256-3_12 pages 250

Wenzel, M. (2006). Structured induction proofs in Isabelle/Isar. In J. M. Borwein & W. M. Farmer (Eds.), *Mathematical Knowledge Management, International Conference, MKM 2006, Wokingham, UK, August 11-12, 2006, Proceedings* (Vol. 4108, pp. 17–30). Springer. doi: 10.1007/11812289_3 pages 250

Wenzel, M. (2007). Isabelle/Isar – a generic framework for human-readable proof documents. *From Insight to Proof—Festschrift in Honour of Andrzej Trybulec*, *10*(23), 277–298. (Special issue of Studies in Logic, Grammar, and Rhetoric) pages 250

Wenzel, M., & Wiedijk, F. (2002). A comparison of Mizar and Isar. *J. Autom. Reasoning*, *29*(3-4), 389–411. doi: 10.1023/A:1021935419355 pages 249

Wiedijk, F. (2001). Mizar light for HOL light. In R. J. Boulton & P. B. Jackson (Eds.), *TPHOLs* (Vol. 2152, pp. 378–394). Springer. doi: 10.1007/3-540-44755-5_26 pages 250

Wijesekera, D. (1990). Constructive modal logics I. *Ann. Pure Appl. Logic*, *50*(3), 271–301. doi: 10.1016/0168-0072(90)90059-B pages 146

Wijesekera, D., & Nerode, A. (2005). Tableaux for constructive concurrent dynamic logic. *Ann. Pure Appl. Logic*, *135*(1-3), 1–72. doi: 10.1016/j.apal.2004.12.001 pages 146, 147

Yan, G., Jiao, L., Wang, S., Wang, L., & Zhan, N. (2020). Automatically generating systemc code from HCSP formal models. *ACM Trans. Softw. Eng. Methodol.*, *29*(1), 4:1–4:39. doi: 10.1145/3360002 pages 90

Yu, K., Chen, Z., & Dong, W. (2014). A predictive runtime verification framework for cyber-physical systems. In *IEEE International Conference on Software Security and Reliability (SERE)* (pp. 223–227). IEEE. doi: 10.1109/SERE-C.2014.43 pages 19

Yu, L. (2013). A formal model of IEEE floating point arithmetic. *Archive of Formal Proofs*. Retrieved from `https://www.isa-afp.org/entries/IEEE_Floating_Point.shtml` pages 91, 104

Zaliva, V., & Franchetti, F. (2018). HELIX: a case study of a formal verification of high performance program generation. In K. Davis & M. Rainey (Eds.), *FHPC@ICFP* (pp. 1–9). ACM. doi: 10.1145/3264738.3264739 pages 20

Zhan, B. (2016). AUTO2, a saturation-based heuristic prover for higher-order logic. In J. C. Blanchette & S. Merz (Eds.), *ITP* (Vol. 9807, pp. 441–456). Springer. doi: 10.1007/978-3-319-43144-4_27 pages 252

Zhou, C., Wang, J., & Ravn, A. P. (1995). A formal description of hybrid systems. In R. Alur, T. A. Henzinger, & E. D. Sontag (Eds.), *Hybrid Systems III: Verification and Control, DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems* (Vol. 1066, pp. 511–530). Springer. doi: 10.1007/BFb0020972 pages 10, 145

Ziliani, B., Dreyer, D., Krishnaswami, N. R., Nanevski, A., & Vafeiadis, V. (2013, September). Mtac: A monad for typed tactic programming in Coq. *SIGPLAN Not.*, *48*(9), 87–100. doi: 10.1145/2544174.2500579 pages 252

Zou, L., Zhan, N., Wang, S., Fränzle, M., & Qin, S. (2013). Verifying Simulink diagrams via a Hybrid Hoare Logic Prover. In *EMSOFT* (pp. 9:1–9:10). IEEE. Retrieved from `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp`

?punumber=6648479 doi: 10.1109/EMSOFT.2013.6658587 pages 90