Gradual Verification of Recursive Heap Data Structures

Jenna Wise DiVincenzo

CMU-S3D-23-109

December 2023

Software and Societal Systems Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jonathan Aldrich, Co-Chair
Joshua Sunshine, Co-Chair
Bryan Parno
Éric Tanter (University of Chile)
Peter Müller (ETH Zurich)

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

Copyright © 2023 **Jenna Wise DiVincenzo**

This work was sponsored by the National Science Foundation (NSF) under Grant Nos. CCF1901033, DGE1745016, and DGE2140739, the Defense Advanced Research Projects Agency (DARPA) under Agreement No. FA8750-16-2-0042, the Algorand Foundation, and the Google PhD Fellowship program. The views and conclusions contained here in are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the NSF, DARPA, Algorand Foundation, Google, or the U.S. Government.

Keywords: gradual verification, implicit dynamic frames, recursive predicates, weakest preconditions, symbolic execution, gradual typing

Dedicated to my mother, inspiration, and strongest supporter, Marlene Wise, in loving memory.

Abstract

Current static verification techniques do not provide good support for incrementality, making it difficult for developers to focus on specifying and verifying the properties and components that are most important. Dynamic verification approaches support incrementality, but cannot provide static guarantees. To bridge this gap, prior work proposed gradual verification, which supports incrementality by allowing every assertion to be complete, partial, or omitted, and provides sound verification that smoothly scales from dynamic to static checking. While promising, the prior approach to gradual verification is merely an indication of feasibility in a very simple setting and is limited to programs without recursive heap data structures.

This dissertation extends gradual verification to programs that manipulate recursive, mutable data structures on the heap. It lays the formal foundations for such gradual verification systems, addressing several technical challenges, such as semantically connecting iso- and equi-recursive interpretations of abstract predicates, and supporting gradual verification of heap ownership.

This work demonstrates the practicality of these foundations by first presenting Gradual C0, the first working gradual verifier for recursive heap data structures. Gradual C0 targets C0, a safe subset of C designed for education. During Gradual C0's development, technical challenges related to symbolic execution with imprecise specifications, minimizing insertion of dynamic checks, and extensibility to other programming languages beyond C0 were addressed. Next, I present an empirical study of gradual verification technology, which explores how specification precision correlates with run-time checking in Gradual C0. The results show that on average, Gradual C0 decreases run-time overhead between 11-34% compared to dynamic verification alone. Further, run-time performance increases in Gradual C0 as more specifications are written—and proof obligations are introduced but not statically verified—until reaching a critical mass where afterwards performance decreases correspondingly—as more and more proof obligations are proved statically. Finally, I also present a case study exploring Gradual C0's practicability and scalability by using the tool to verify a 3k lines of code C parser for loop termination. I found that Gradual C0's strong adherence to the gradual guarantee—which ensures Gradual C0 does not produce static or dynamic verification errors resulting from missing specifications—was necessary for assuring real software with many modules and functions and allowed me to find bugs in code and specifications far earlier than static verification alone. However, this property at times hindered efforts to reduce run-time checking through writing additional specifications. I propose a new specification construct (inspired by prior work in gradual typing) that is designed to facilitate this process in the presence of the gradual guarantee.

This dissertation thus makes significant contributions to realizing the promising idea of gradual verification, and lays a solid foundation for future gradual verification technology and tools that work at scale.

Acknowledgments

My PhD journey was a marathon of highs and lows and everything in between. But, I would always, without hesitation, choose to do it all over again thanks to my wonderful support network.

First and foremost, I thank my advisors Dr. Joshua Sunshine and Dr. Jonathan Aldrich for their dedication to my success as well as my research's success. I am deeply grateful for the many hours they spent helping me improve my presentation, writing, teaching, mentoring, and research problem formulation skills—as well as the many hours spent advising me on technical work. They helped me network with other academics, helped me advertise my research more broadly, and encouraged me to participate in a number of unique career building opportunities. Therefore, both this document and my appointment as an Assistant Professor at Purdue University would not be possible without them. Huge thanks to them for making my dream of being a professor come true!

This thesis work would also not be possible without my research collaborators. Thanks to Dr. Éric Tanter, Johannes Bader, MSc, Cameron Wong, Ian McCormack, Hemant Gouni, Jacob Gorenburg, Mona Zhang, Conrad Zimmerman, and Jan-Paul Ramos-Dávila for making significant, high quality contributions to the research presented in this document. I also appreciate Samuel Estep's work on gradual program analysis, which influenced some of my gradual verification work. Finally, the thorough writing feedback Dr. Jeremy Lacomis, Dr. Fraser Brown, and my committee members provided to me on this document was invaluable.

My family and friends' unconditional love and support helped me through the hardest times during my PhD. I will forever be grateful to my husband, Michael DiVincenzo, for the sacrifices he made to support my education and career. He is my rock and safe haven. Thanks to my adorable cats, Cosmo and Zuko, for their daily snuggles and general cuteness. I appreciate my sister, Sarah Wise, for helping me stay productive on bad days and listening to my venting sessions. Thanks to my father, Lee Wise, for inspiring my love of mathematics and computer science; and, for helping me describe my research in layman's terms. My aunt, Kathleen Lesnak, stepped up as a mother figure in my life after my mother's passing early in my PhD; she helped me get through such a dark time in my life and finish my PhD. Words cannot express how grateful I am for her presence in my life. I have the best of friends: Dr. Samantha Stoner, Nicole Moore, Morgan Evans, and Kush Jain, who always lift me up and inspire me to be a better person and researcher. I cannot thank my best friends, family, and other friends (especially my gaming/FFXIV friends) enough for helping me maintain a healthy work-life balance during my PhD.

The Software and Societal Systems Department (S3D) in the School of Computer Science at Carnegie Mellon University (CMU) is a wonderful place to do a PhD. Many faculty and students gave me helpful feedback on the research in this dissertation, including new ideas to pursue, for which I am immensely grateful. I also thank them for their strong commitment to my success. Over the years, many staff members in S3D helped me with various things, like travel reimbursements, booking rooms, scheduling meetings, taking professional headshots, advertising my success, helping my undergraduate mentees and I get compensation, answering PhD program questions, and so much more. I could focus the majority of my time on research thanks to their efforts! I also appreciate the many normal conversations I had with staff members. I am much more than my research, with many hobbies and interests, and I thank the S3D staff members for frequently reminding me of this. I was able to complete this long PhD journey because of the care and kindness exhibited by every member of S3D. Thank you so much!

Finally, I am forever grateful to my undergraduate advisor, Dr. Bonita Sharif, for mentoring me in research and preparing me for a PhD. Her continued support and check-ins during my PhD were strong motivators to see my PhD to the end. I hope I have made her proud and that I continue to make her proud! I also thank Matt Weyant, MSc, my supervisor at the MIT Lincoln Laboratory, for connecting me with Dr. Jonathan Aldrich. This connection led to me joining S3D at CMU under Jonathan and Josh's advisement. Many thanks to my supervisors, Dr. Manoj Kumar and Dr. Pratap Pattnaik, at IBM Research for allowing me to explore my own research agenda in their group and helping me publish my first paper during my PhD. Lastly, I appreciate Drs. Roopsha Samanta, John Boyland, Peter Müller, Ronald Garcia, and Jeremy Siek for appreciating this work in its early stages and encouraging me to continue pursing it.

Contents

1 Introduction **1**
1.1 Thesis Statement . . . 2
1.2 Thesis Contributions . . . 2
1.2.1 Document Roadmap . . . 4
1.3 Challenges of Recursive Heap Data Structures . . . 4
1.3.1 Gradual Verification of Heap Ownership . . . 4
1.3.2 Gradual Verification of Recursive Predicates . . . 5
1.4 The Burden of Static Verifiers . . . 6
1.4.1 Auxiliary Specifications . . . 7
1.4.2 Lack of Early Specification Feedback . . . 9
1.5 How Gradual Verification Can Help . . . 11
1.5.1 Ignore Auxiliary Specifications with Gradual Verification . . . 11
1.5.2 Gradual Verification Gives Early Feedback with Run-time Checks . . . 12
1.6 Related Work . . . 12
1.6.1 Gradual Typing . . . 12
1.6.2 Static Verification . . . 13
1.6.3 Dynamic Verification . . . 14
1.6.4 Hybrid Verification . . . 14

2 Formal Foundations **17**
2.1 SVL_{RP} . . . 18
2.1.1 Syntax & Static Semantics . . . 18
2.1.2 Formula Semantics . . . 19
2.1.3 Static Verification . . . 22
2.1.4 Dynamic Semantics . . . 23
2.1.5 Soundness . . . 24
2.2 GVL_{RP}: Static Semantics . . . 25
2.2.1 Syntax . . . 25
2.2.2 Framing . . . 25
2.2.3 Interpretation of Gradual Formulas . . . 26
2.2.4 Lifting Predicates . . . 28
2.2.5 Lifting Functions . . . 29
2.2.6 Lifting the Verification Judgment . . . 31
2.3 GVL_{RP}: Dynamic Semantics . . . 31

2.3.1 Footprint Splitting . . . 31
2.3.2 Small-Step Semantics . . . 32
2.4 Properties of GVL_{RP} . . . 32

3 Gradual C0: The First Gradual Verifier **35**
3.1 Gradual C0's Overall Design . . . 36
3.2 Translating C0 Source Code to Gradual Viper Source Code for Verification . . . 37
3.3 Gradual Viper: Symbolic Execution for Gradual Verification . . . 39
3.3.1 **Optimistic static verification in Gradual Viper by example** . . . 40
3.4 Gradual Viper: Implemented Algorithm . . . 48
3.4.1 **Run-time checks** . . . 49
3.4.2 **Symbolic State** . . . 49
3.4.3 **Preliminaries** . . . 50
3.4.4 **Symbolic execution of expressions** . . . 50
3.4.5 **Symbolic production of formulas** . . . 51
3.4.6 **Symbolic consumption of formulas** . . . 52
3.4.7 **Symbolic execution of statements** . . . 55
3.4.8 Valid Gradual Viper programs . . . 56
3.5 Dynamic Verification: Encoding Run-time Checks into C0 Source Code . . . 57
3.5.1 **Encoding branch conditions** . . . 58
3.5.2 **Encoding predicates** . . . 59
3.5.3 **Encoding accessibility predicates and separating conjunctions** . . . 59

4 Performance Evaluation of Gradual C0 **61**
4.1 Creating Performance Lattices . . . 62
4.2 Data Structures . . . 62
4.3 Experimental Setup . . . 64
4.4 Evaluation . . . 64
4.5 Threats to Validity . . . 68
4.6 Qualitative Experience with AVL Tree . . . 69

5 Case Study: Gradual Verification of a C Parser **71**
5.1 The Case Study: a C Parser . . . 72
5.1.1 Translating Chibicc Parser Code into C0 Code . . . 73
5.1.2 Testing TinierCP . . . 77
5.1.3 Takeaways . . . 77
5.2 Methodology . . . 78
5.2.1 Data & Materials . . . 79
5.2.2 Data Analysis Procedure . . . 80
5.3 Results . . . 81
5.3.1 The author verified loop termination in TinierCP in a top-down manner focusing only on relevant functions and properties . . . 82
5.3.2 Gradual C0 makes selective and top-down verification workflows possible 84

5.3.3 Selective and top-down verification workflows backed by Gradual C0 are useful for assuring loop termination in TinierCP . . . 85
5.3.4 Gradual C0 needs further improvements to support selective and top-down verification workflows . . . 88
5.3.5 General Gradual C0 limitations and improvements . . . 91
5.4 Discussion . . . 92
5.5 Threats to Validity . . . 94

6 Conclusion 95

A Appendix 97
A.1 Chpt. 2's Appendix . . . 97
A.1.1 SVL_{RP} . . . 97
A.1.2 GVL_{RP} . . . 104
A.2 Chpt. 3's Appendix . . . 108
A.2.1 Diff and Translate . . . 109
A.2.2 Symbolic production of formulas . . . 111
A.2.3 **Symbolic consumption of formulas** . . . 112
A.2.4 **Symbolic execution of statements** . . . 115
A.2.5 **Valid program** . . . 120

Bibliography 121

List of Figures

1.1 Non-empty linked list insertion in C0 6
1.2 The static verification of `insertLast` from Fig. 1.1 8
1.3 The incremental verification of `insertLast` from Fig. 1.1 9

2.1 SVL_{RP}: Syntax 18
2.2 SVL_{RP}: Formula evaluation (select rules) 20
2.3 SVL_{RP}: Iso-recursive formula evaluation (select rule) 20
2.4 SVL_{RP}: Framing (select rules) 21
2.5 SVL_{RP}: Weakest liberal precondition calculus (select rules) 23
2.6 SVL_{RP}: Small-step semantics (select rules) 24
2.7 GVL_{RP}: Weakest liberal precondition calculus (select rules). 30
2.8 GVL_{RP}: Small-step semantics adjusted from Fig. 2.6 (select rules) 32

3.1 System design of Gradual C0 36
3.2 Shared abstract syntax definitions 38
3.3 GVC0 abstract syntax 38
3.4 Gradual Viper abstract syntax 38
3.5 Abstract syntax comparison for GVC0 and Gradual Viper 38
3.6 A simple bank withdraw example written in the Gradual C0 language 40
3.7 A simple bank withdraw program written in the Gradual Viper language 40
3.8 A gradually verified, bank withdraw program that is contrived to illustrate how Gradual Viper works 40
3.9 Contents of the symbolic state at each program point during Gradual Viper's static verification of `withdraw` in Fig. 3.7 41
3.10 Rules for symbolically executing expressions 51
3.11 Rules for symbolically producing formulas 52
3.12 Formally defining the branch function 53
3.13 *Select rules* for symbolically consuming formulas 54
3.14 *Select rules* for symbolically executing program statements 55
3.15 Rules defining a valid Gradual Viper program 57
3.16 `insertLastWrapper` program with run-time checks from Gradual Viper . . . 58
3.17 `insertLastWrapper` program with run-time checks generated by GVC0 . . . 58

4.1 For each example, the average quantity of verification conditions and the subset that were eliminated statically at each level of specification completeness across all paths sampled. Shading indicates the standard deviation. 65
4.2 The mean time elapsed at each step over the 16 paths sampled. Shading indicates the confidence interval of the mean for each verification type. 65
4.3 The quantity of specification elements, grouped by type and location, that caused the highest ($P_{99\%}$) increases and decreases in time elapsed out of every path sampled 67

5.1 C Grammar parsed by TinierCP 73
5.2 C program used to test TinierCP 75
5.3 Questions asked in the Google Form used to capture qualitative data from the author. All questions were required 79
5.4 Partial specification summaries for select partial specifications of TinierCP. For each partial specification, the table gives the ratio of lines of specification code to lines of program code and the distribution of specification elements for the partial specification. Element counts are formatted as "*Accessibility Predicate/Predicate Instance/Boolean Expression/Imprecision/Total*" 82
5.5 Gradual verifier results for select partials specifications of TinierCP. For each partial specification, the table gives the percentage of verification conditions statically verified, the time Gradual C0 spent statically verifying the partial specification in minutes and seconds, and the time Gradual C0 spent dynamically verifying the partial specification in seconds for the test case from §5.1.2 82
5.6 Graph of function calls for TinierCP functions partially specified by the author . . 83
5.7 skip code snippet 88
5.8 declspec code snippet 88
5.9 skip and declspec code snippets from TinierCP with the author's specifications . 88

A.1 SVL_{RP}: Expression dynamic semantics 97
A.2 SVL_{RP}: Formula evaluation 98
A.3 SVL_{RP}: Framing 99
A.4 $\mathsf{acc}(e) : \textsc{Expr} \rightarrow \textsc{Formula}$ 100
A.5 SVL_{RP}: Weakest liberal precondition calculus 101
A.6 SVL_{RP}: Small-step semantics 102
A.6 SVL_{RP}: Small-step semantics (continued) 103
A.7 Heap aware weakest liberal precondition across multiple stack frames 104
A.8 Definition of the TotalFP function. 104
A.9 GVL_{RP}: Weakest liberal precondition calculus. 105
A.10 GVL_{RP}: Small-step semantics adjusted from Fig. A.6 for gradual formulas 106
A.10 GVL_{RP}: Small-step semantics adjusted from Fig. A.6 for gradual formulas (continued) 107
A.11 Path condition helper functions 108
A.12 Rules for symbolically executing expressions without introducing run-time checks (except for a special case for unfold) 108

A.13 Rules for symbolically executing expressions without modifying the optimistic heap and path condition . . . 109
A.14 Algorithm for computing the diff between two symbolic values . . . 109
A.15 TRANSLATE's procedure for resolving variables . . . 110
A.16 Rules for symbolically consuming formulas (1/3) . . . 112
A.16 Rules for symbolically consuming formulas (2/3) . . . 113
A.16 Rules for symbolically consuming formulas (3/3) . . . 114
A.17 Heap remove function definitions . . . 116
A.18 Check and assert function definitions . . . 116
A.19 Rules for symbolically executing program statements (1/2) . . . 117
A.19 Rules for symbolically executing program statements (2/2) . . . 118
A.20 Boolean function determining if a gradual formula is equi-recursively imprecise or not . . . 119
A.21 Well-formed formula function definition . . . 119

List of Tables

4.1 Description of benchmark examples. For each example, the table shows the complexity of the test program without verification, the number of sampled partial specifications, and the distribution of specification elements for the complete specification. Element counts are formatted as “*Accessibility Predicate/Predicate Instance/Boolean Expression/Imprecision*" 63
4.2 Summary statistics for the performance of each example over 16 paths at selected workloads (ω), comparing gradual verification (GV) against dynamic verification (DV). The grouped column “% in Δt, GV. vs. DV." displays summary statistics for the percent decrease in time elapsed for each step when using GV versus DV. The column “% Steps GV < DV for Paths DV < GV" shows the distribution of steps that performed best under GV that were part of paths containing steps that performed better under DV. The final column shows the percentage of paths in which every step performed better under GV. 66

5.1 C features used by Chibicc that are unsupported by C0 and Gradual C0, and how they are avoided in the verifiable C0 version of Chibicc's parser 74

Chapter 1

Introduction

Automated deductive verification tools (static verifiers), such as Viper [33], Dafny [26], and VeriFast [22], are powerful in that they prove the absence of bugs in software with respect to a user provided specification and do so automatically. Therefore, since the first introduction of the technology supporting such tools—Hoare logic [21]—in 1969, researchers have continued making advancements to support more interesting programs and properties. *Weakest liberal preconditions*, introduced by Dijkstra [11], made implementing Hoare logic with *satisfiability modulo theories (SMT) solvers* much easier. Then, thanks to improved capabilities of SMT solvers—from advancements in SAT solving [31]—proofs of many low-level properties have been automated. More recently, *separation logic* [38] enabled the modular static verification of heap-manipulating programs. Its variant *Implicit dynamic frames* (IDF) [44] and extension with *recursive abstract predicates* [36, 44] further support verifying recursive heap data structures, such as trees, lists, and graphs.

While these techniques allow users to specify and verify more code than ever before, static verifiers implementing them (*e.g.* Viper [33], VeriFast [22], and Dafny [26]) are still largely unused in practice due to the burden they place on their users. Such tools poorly support partial specifications, and thus require users to provide a number of auxiliary specifications (such as folds, unfolds, loop invariants, and inductive lemmas) in an all or nothing fashion to discharge proofs of code correctness. Additionally, the tools also require many of these auxiliary specifications to be written before they can provide feedback on the correctness of wanted specifications (such as pre- and postconditions specifying the behavior of functions). For example, to prove that a simple list insertion function preserves list acyclicity, static verifers need 1.5 times as many lines of auxiliary specifications to program code (§1.4). They also need a significant number of these auxiliary specifications to uncover problems with specifications of the acyclic property (§1.4). I discuss this example and the burden of static verifiers in more detail in §1.4.

To address this issue, Bader et al. [3] proposed *gradual verification*, which builds on prior research on gradual typing [41, 42, 43]—in particular the Abstracting Gradual Typing methodology [19]—to support the incremental specification and verification of software. Bader et al. [3] extend a simple Hoare logic static verifier with partial, *imprecise specifications* backed by run-time checking where necessary. An imprecise formula can be fully unknown, written `?`, or combine a *static part* with the unknown, as in $? * x.f == 2$. During static verification, an imprecise specification can be optimistically strengthened (in non-contradictory ways) by the

verifier to support proof goals. Wherever such strengthenings occur, dynamic checks are inserted to preserve soundness. Gradual verification smoothly supports the spectrum between static and dynamic verification. This is captured by properties adapted from gradual typing [43], namely the *gradual guarantee*, stating that the verifier will not flag static or dynamic errors for specifications that are correct but imprecise, and the fact that gradual verification *conservatively extends* static verification, *i.e.* they coincide on fully-precise programs.

While the basic principles of gradual program verification have already been laid out by Bader et al. [3], their work only accounts for pre- and postconditions that include basic logical and arithmetic formulas. Thus, this work only applies to simple programs without loops, heap values, or recursive data structures; and can be categorized as an early proof of concept of the idea of gradual verification. To move research in gradual verification forward and show that it can make significant improvements to the usability and scalability of verification technology, this dissertation advances Bader et al. [3]'s basic principles to deal with realistic programming scenarios involving recursive heap data structures.

1.1 Thesis Statement

This brings me to my thesis statement:

It is possible to build gradual verification technology that

- *supports the specification and verification of programs manipulating recursive heap data structures,*
- *is sound and adheres to gradual properties,*
- *has minimized run-time overhead, and*
- *is useful in practice.*

Note, from here on out I will use “we" to describe work that was done in collaboration with others (all of which have been acknowledged in the acknowledgements section). I am, of course, the main contributor (“first author") of the work presented in this dissertation.

1.2 Thesis Contributions

To support this thesis statement, this document makes the following contributions:

Theoretical Foundations of Gradual Verification for Recursive Heap Data Structures. This dissertation presents the **design, formalization, and meta-theory of the first gradual verification system for recursive heap data structures** [47]. It supports the strengthening of imprecise specifications with *accessibility predicates* (from IDF) and *recursive predicates*; and thus, also the run-time verification of these constructs. We proved that this system is sound and adheres to gradual properties—*i.e.* it is a conservative extension of the base static verifier and adheres to the gradual guarantee. In this more sophisticated setting, we addressed several technical challenges, such as semantically connecting iso- and equi-recursive interpretations of recursive predicates,

and supporting the gradual verification of heap ownership. §1.3 discusses these challenges in more detail.

Gradual C0: the First Gradual Verification Tool. We present the **design and implementation of *Gradual C0*—the first gradual verifier that both can be used on real programs containing recursive heap data structures and minimizes run-time checks with statically available information** [14]. Gradual C0's design is inspired by our foundational theory and is sound and adheres to gradual properties. Technically, Gradual C0 is built on top of the Viper static verifier [33], which supports IDF and recursive abstract predicates; and programs verified by Gradual C0 are written in the C0 programming language, which is a simpler and safer subset of the C language targeted at education [2]. Gradual C0 leverages Viper's infrastructure to simplify the implementation of gradual verifiers for other programming languages, and we demonstrate how this is done for C0. In this work, we addressed new technical challenges from realizing our foundational theory in a working tool. We handle challenges from using *symbolic execution* for reasoning, such as relating symbolic values to their source code counterparts for run-time checking, and minimizing run-time checks and their overhead, such as tracking a set of optimistically assumed accessibility predicates during static verification.

Performance Evaluation of Gradual C0. We evaluate how precision correlates with run-time checking in Gradual C0 with a **first of its kind empirical evaluation of a gradual verifier** [14]. The study explores static and dynamic performance characteristics for thousands of sample partial specifications derived from four common data structures, as inspired by similar work in gradual typing [46]. In particular, we observe how adding or removing individual atomic formulas and `?` within a specification impacts the degree of static and dynamic verification and, as a result, the run-time overhead of the program. We compare run-time performance to a fully dynamic approach as a baseline. We found that Gradual C0 reduces run-time overhead by an average of 11-34% compared to dynamic verification and respects the gradual guarantee empirically across the sampled specifications. Further, Gradual C0 follows interesting performance trends: performance increases gradually as more proof obligations are specified but are not yet statically verified; and thus, must be checked at run time. Eventually, a critical mass of specifications are written that allows more and more of these proof obligations to be proven statically and run-time overhead decreases accordingly.

A Case Study with Gradual C0. We **explore how gradual verification is used to verify real application software that uses recursive heap data structures through a case study**. We used Gradual C0 to verify a subset of a 3k lines of code C parser, called TinierCP, for loop termination, and recorded the experience through journal entries. TinierCP has various structures and modules, plenty of functions (27) that call each other in intricate ways, and manipulates lists and trees with loops and recursion across these functions. We qualitatively coded the entry data in three rounds: from initial low-level codes into patterns and themes and then into a narrative. We found that Gradual C0's strong adherence to the gradual guarantee, which suppresses errors caused by missing specifications, allowed us to verify only the code and properties we cared about in a top-down workflow. Relying on Gradual C0's verification results in a top-down work-

flow also allowed us to uncover bugs in TinierCP and our specifications during the verification process—and do so far earlier than if we would have used static verification alone. But, Gradual C0 has room for improvement. Gradual C0 should provide an additional specification language construct (inspired by prior work on gradual typestate [18]) that allows users to specify—via predicates and accessibility predicates—what heap locations are not accessed by a function (this is in contrast to preconditions, which specify the heap locations accessed by a function). This construct allows users to provide additional information to Gradual C0 that it can use to discharge more proof obligations statically reducing run-time verification overhead. Additionally, we can improve Gradual C0's usability by having Gradual C0 report whether specifications marked by a user are checked statically or dynamically.

1.2.1 Document Roadmap

The rest of this document is outlined as follows. The challenges with supporting the gradual verification of recursive heap data structures are discussed in §1.3 along with our novel solutions. The burden static verifiers place on their users is demonstrated in §1.4, and we demonstrate how gradual verification can alleviate this burden in §1.5. Related work is given in 1.6. Chpt. 2 lays out the formal foundations of gradual verification for recursive heap data structures, and Chpt. 3 presents the design and implementation of Gradual C0. The performance study is discussed in Chpt. 4 and the case study in Chpt. 5. The document concludes in Chpt. 6 with future work.

1.3 Challenges of Recursive Heap Data Structures

This section explains the challenges involved in accounting for implicit dynamic frames (IDF) [44] and recursive abstract predicates [36] in the context of gradual program verification. We also informally outline our novel solutions to these challenges, which will be formally developed, implemented, and evaluated in the following chapters.

1.3.1 Gradual Verification of Heap Ownership

Adapting the Abstracting Gradual Typing approach [19] to the verification setting gives meaning to imprecise formulas such as `x > 10 ∧ ?` by considering all the *logically consistent strengthenings* of such formulas [3, 25]. For instance, `x > 10 ∧ ?` *consistently implies* `x > 20`, but not `x < 0`. In the latter case, the formula `x < 0` contradicts the static part of the imprecise formula `x > 10`. In the former case, if we definitely know that `x > 10`, then it might optimistically be the case that `x > 20` as well. Of course, in order to preserve soundness, optimistically assuming `x > 20` when one only definitely knows that `x > 10` requires a run-time check to corroborate that the value bound to `x` at run-time is indeed greater than `20`.

When dealing with heap data structures, the logic—IDF in our case—includes more than arithmetic: we need to be able to talk about heap separation (with the separating conjunction $*$) and ownership of heap cells (with accessibility predicates like **acc**`(x.f)`). How are we to extend the interpretation of imprecise formulas in such a setting, and how can we soundly track optimistic assumptions?

Imprecise Heap Formulas. When using IDF in a static verifier, one must make sure that formulas are self-framed. For instance, `x.f >= 2` is not self-framed, because it does not explicitly mention the accessibility predicate needed to evaluate the formula. The formula `acc(x.f) * x.f >= 2` is self-framed. We want to ensure that programmers can smoothly strengthen specifications, and one logical kind of strengthening is adding accessibility predicates that were previously missing. Accordingly, in our design imprecise formulas must optimistically allow `?` to stand in for accessibility predicates that are necessary for framing. Furthermore, this is true whether the imprecise formula appears directly in an assertion or indirectly in the definition of an abstract predicate. Indeed, in IDF, framing can sometimes come from an abstract predicate. For instance, `foo(x) * unfolding foo(x) in x.f >= 2` is self-framed if the body of `foo(x)` includes `acc(x.f)`. Thus, our semantics for imprecise formulas must allow `?` to denote not only for predicates such as `foo(x)`, but also any unfoldings of them that are necessary to frame the static part of the formula.

Runtime Checking of Ownership. For a gradual verifier to be sound, optimistic assumptions made statically due to imprecision must be safeguarded dynamically through run-time checks. Extending gradual verification to IDF by allowing imprecision to account for missing accessibility predicates means that we need to keep track of ownership in the run-time system. In particular, we design a run time that tracks and updates a set of heap locations at every program point, indicating current ownership. Heap locations are added to this set when objects are created. Each time a field is accessed, the set of owned locations is looked up: if the corresponding permission is found, the check succeeds, otherwise a run-time error is raised.

At a call site, if an owned heap location is required by the precondition of the callee, then it is removed from the owned locations of the caller. When the callee finishes executing, all callee owned heap locations are passed to the caller.

The challenge here is how to deal with imprecise preconditions, either directly or via an imprecise abstract predicate. In order to maximize the ability for the callee to execute properly, an imprecise precondition has to require *all* the owned heap locations of the caller. Indeed, said imprecision might potentially denote any location owned by the caller, not already passed statically, and effectively required in the callee. Not transferring its ownership means the callee might error out at run time.

1.3.2 Gradual Verification of Recursive Predicates

Recursive predicates can be dealt with in two different manners in program verification [45]: either *iso-recursively*—in which case to be able to exploit a predicate instance, one needs to explicitly unfold it, and vice versa, to explicitly fold it back to establish it—or *equi-recursively*—in which case a predicate is deemed identical to its unfolding, which need not be specified explicitly. These two approaches have complementary strengths, which, we argue, are particularly relevant for gradual verification. The iso-recursive approach is critical for making static reasoning manageable for tools (and for humans who must deal with the error messages reported by these tools) because it breaks reasoning into small steps. In contrast, the equi-recursive approach is much more convenient in a dynamic setting, where the run-time system can automatically

unfold predicates as needed, and so the user does not have to write explicit folds and unfolds.

In this work, we propose a novel design that achieves the benefits of both approaches. Statically, the gradual verifier treats recursive predicate instances *iso-recursively*: programmers can specify folds and unfolds in the precise parts of their pre- and postconditions, as well as in program statements, just as they would with mainstream static verifiers. By exploiting syntax, verification becomes simply algorithmic for tools to implement, and visually clear for humans to keep track of the underlying activity of the verifier.

In contrast, dynamically, predicate instances are checked equi-recursively. An equi-recursive evaluation of predicate instances is the natural choice for dynamic checking, as the run-time system can simply execute the predicate as if it were a function. Crucially, an equi-recursive approach to program evaluation allows users to leave out fold and unfold statements, which one can expect to be the default for partially (or un-)verified code. Seen dually, adopting an iso-recursive run-time approach while allowing programmers to omit (un)folding statements would mean trying to automatically infer when to actually perform (un)folding. Known approaches to this are heuristic, meaning that some well-behaved code could be conservatively rejected when made imprecise enough. This would result in a violation of the dynamic gradual guarantee [43], whose motto is that *losing precision is harmless*.

Therefore we argue that combining iso- and equi-recursive treatments of recursive predicates is *required* in order to achieve a proper gradual verifier: statically, the iso-recursive approach ensures algorithmic checking, and dynamically, the equi-recursive approach allows imprecise code to run smoothly.

1.4 The Burden of Static Verifiers

```
1  struct Node { int val; struct Node *next; };
2  typedef struct Node Node;
3
4  Node* insertLast(Node *list, int val)
5  {
6      Node *y = list;
7      while (y->next != NULL)
8        { y = y->next; }
9      y->next = alloc(struct Node);
10     y->next->val = val;
11     y->next->next = NULL;
12     return list;
13 }
```

Figure 1.1: Non-empty linked list insertion in C0

Static verifiers require a number of user-provided auxiliary specifications, such as folds, unfolds, lemmas, and loop invariants, to prove properties about recursive heap data structures. Worse, they also require users to write many of these auxiliary specifications before the tools can provide useful feedback on the correctness of other specifications, including ones containing important functional properties. Therefore, users are burdened by writing many detailed and extraneous specifications with inadequate static feedback throughout the process. In this section, we illustrate this burden with a simple list insertion example and output from Viper [33]. Then, we show in §1.5 how gradual verification overcomes this burden by smoothly supporting the spectrum between static and dynamic checking. Users can avoid writing auxiliary specifications and still get sound verification of their code with increased run-time checking. Users can also receive run-time feedback on the correctness of their specifications very early in the specification process, and the resulting error

messages closely align with inherent problems in the specifications or in the program, making debugging them easier.

Fig. 1.1 implements a linked list and function that inserts a new node at the end of a given list, called `insertLast`, in C0 [2]. In this section and in §1.5, all examples are written in C0 code as that is the language supported by our gradual verifier Gradual C0. In this work, we also extended Viper to verify C0 programs. The `insertLast` function traverses the list to its end with a `while` loop starting from the root. That is, `insertLast` implicitly assumes the list is non-empty (non-null) and acyclic; and that for multiple successive calls to `insertLast` the list remains acyclic and non-empty after insertion. These facts can be proven explicitly with static verification; the complete static specification is given in Fig. 1.2, highlighted in grey. Note, both Gradual C0 and Viper denote the separating conjunction as `&&` rather than $*$, so we do the same in this section and §1.5.

List acyclicity is specified with two predicates `acyclicSeg` and `acyclic`:

```
predicate acyclicSeg(Node *s, Node *e) =
    s == e ? true : acc(s->val) && acc(s->next) && acyclicSeg(s->next,e)
predicate acyclic(Node *n) = acyclicSeg(Node *n,NULL)
```

The `acyclicSeg` predicate uses *accessibility predicates* and the *separating conjunction* from IDF. Ownership is ensured through *accessibility predicates* such as **acc**`(s->val)`; and, **acc**`(s->val) && s->val` $==2$ states that `s->val` is uniquely owned and contains the value 2. The *separating conjunction*, denoted by `&&`, ensures memory disjointness: **acc**`(s->next) &&` **acc**`(s->next->next)` states that the heap locations `s->next` and `s->next->next` are distinct (*i.e.* $s \neq$ `s->next`) and are each owned.

Thus, the abstract recursive predicate `acyclicSeg`, which can be thought of as a pure boolean function, specifies that a list segment is acyclic. That is, `acyclicSeg(s,e)` denotes that all heap locations in list `s` are distinct up to node `e` by recursively generating accessibility predicates for each node in `s` up to `e`, joined with the separating conjunction. Further, `acyclicSeg(n,NULL)` denotes that all heap locations in list `n` are distinct and so `n` is acyclic, as specified with `acyclic(n)`.

Now that we have specified `acyclic`, we use it in `insertLast`'s precondition (line 5) and postcondition (lines 6-7) to denote preservation of list acyclicity. We also specify that `insertLast` preserves list non-nullness with simple logical comparisons (*i.e.* `list != NULL` and `\result != NULL`). Ideally, we would stop here and static verifiers would be able to prove `insertLast`'s implementation correct with respect to this specification; however, as you can see in Fig. 1.2 such tools require many more specifications. In fact, there are 27 lines of auxiliary specifications (comprised of folds, unfolds, loop invariants, and inductive lemmas); in contrast to 18 lines of wanted specifications (the predicates and pre- and postconditions) and program code. Furthermore, these auxiliary specifications are complex, as discussed next.

1.4.1 Auxiliary Specifications

Static verifiers cannot reliably unroll recursive predicates during verification; so, such tools rely on explicit *fold* and *unfold* statements to control the availability of predicate information in the

```
1  /*@ predicate acyclicSeg(Node *s, Node *e) =
2      (s == e) ? true : acc(s->val) && acc(s->next) && acyclicSeg(s->next,e); @*/
3  //@ predicate acyclic(Node *n) = acyclicSeg(n,NULL);

4  Node *insertLast(Node *list, int val)
5    //@ requires acyclic(list) && list != NULL;
6    /*@ ensures acyclic(\result) &&
7                \result != NULL; @*/
8  {
9     //@ unfold acyclic(list);
10    //@ unfold acyclicSeg(list,NULL);
11    Node *y = list;
12    //@ fold acyclicSeg(list,y);
13    while (y->next != NULL)
14    /*@ loop_invariant acyclicSeg(list,y) &&
15          acc(y->next) && acc(y->val) &&
16          acyclicSeg(y->next,NULL); @*/
17    {
18      Node *tmp = y;
19      y = y->next;
20      //@ unfold acyclicSeg(y,NULL);
21      //@ fold acyclicSeg(tmp->next,y);
22      //@ fold acyclicSeg(tmp,y);
23      mergeLemma(list,tmp,y);
24    }
25
26
27
28      y->next = alloc(struct Node);
29      y->next->val = val;
30      y->next->next = NULL;
31  //@ fold acyclicSeg(y->next->next,NULL);
32  //@ fold acyclicSeg(y->next,NULL);
33  //@ fold acyclicSeg(y,NULL);
34  mergeLemma(list,y,NULL);
35  //@ fold acyclic(list);
36      return list;
37  }
38
39  void mergeLemma(Node *a,Node *b,Node *c)
40    /*@ requires acyclicSeg(a,b) &&
41                  acyclicSeg(b,c); @*/
42    //@ ensures acyclicSeg(a,c);
43   {
44       if (a == b) {
45       } else {
46           //@ unfold acyclicSeg(a,b);
47           mergeLemma(a->next,b,c);
48           //@ fold acyclicSeg(a,c);
49       }
50  }
```

☐ Program code ▒ Static specification

Figure 1.2: The static verification of `insertLast` from Fig. 1.1

verifier. This treats predicates *iso-recursively*; while an *equi-recursive* interpretation treats predicates as their complete unrolling [45]. Consequently, the `acyclic` and `acyclicSeg` predicates are unfolded and folded often in Fig. 1.2 (lines 9-10, 12, 20-22, 31-33, and 35). Looking closely, we see that `acyclic(list)`, which is assumed true from the precondition, is unfolded on line 9. This consumes `acyclic(list)` and produces its body `acyclicSeg(list,NULL)`, which is subsequently unfolded on line 10. Then, at the fold on line 35, the body of `acyclic(list)` is packed up into the predicate itself to prove the list remains acyclic after insertion.

Additionally, static verifiers cannot tell if or when a loop will end (in our example the verifier cannot tell when the list being iterated over ends), but must verify all paths through the program. Therefore, static verifiers reason about loops using specifications called *loop invariants*, which are properties that are preserved for each execution of the loop including at entry and exit. Further, loop invariants must also provide information necessary for proof obligations after the loop, *e.g.* that the list in `insertLast` is acyclic after insertion. In Fig. 1.2, these constraints result in the loop invariant on lines 14-16 that segments the list into three disjoint and acyclic parts: from the root up to the current node `y` (`acyclicSeg(list,y)`),

```
1  Node *insertLast(Node *list, int val)
2    //@ requires acyclic(list) && list != NULL;
3    /*@ ensures acyclic(\result) &&
4                   \result != NULL; @*/
5  {
6      //@ unfold acyclic(list);
7      //@ unfold acyclicSeg(list,NULL);
8      Node *y = list;
9      //@ fold acyclicSeg(list,y);
10     while (y->next != NULL)
11     /*@ loop_invariant acyclicSeg(list,y) &&
12            acc(y->next) && acc(y->val) &&
13            acyclicSeg(y->next,NULL) ; @*/
14     {
15       Node *tmp = y;
16       y = y->next;
17       //@ unfold acyclicSeg(y,NULL);
18       //@ fold acyclicSeg(tmp->next,y);
19       //@ fold acyclicSeg(tmp,y);
20       mergeLemma(list,tmp,y);
21     }
22
23     y->next = alloc(struct Node);
24     y->next->val = val;
25     y->next->next = NULL;
26   //@ fold acyclicSeg(y->next->next,NULL);
27   //@ fold acyclicSeg(y->next,NULL);
28   //@ fold acyclicSeg(y,NULL);
29   mergeLemma(list,y,NULL);
30   //@ fold acyclic(list);
31     return list;
32 }
```

1st increment (least precise)	2nd increment	3rd increment	4th increment
5th increment	6th increment	7th increment (full spec)	

Figure 1.3: The incremental verification of `insertLast` from Fig. 1.1

the current node `y` (**acc**`(y->val) && ` **acc**`(y->next)`), and from the node after `y` to the end (`acyclicSeg(y->next,NULL)`). Exposing `y` via its accessibility predicates provides access to `y->next` on line 19 in the loop body; and, `acyclicSeg(list,y)` helps prove `acyclic(list)` holds after the loop, as we will see next.

To prove `acyclic(list)` holds at the end of `insertLast` (line 36), it is sufficient to prove instead that `acyclicSeg(list,NULL)` holds (line 35). After inserting a new node at the end of the list (lines 28-30), we can build up an inductive proof with folds (lines 31-33) that the list is acyclic from the insertion point `y` to the new end, *i.e.* `acyclicSeg(y,NULL)` holds. We also have that the list is acyclic from the root to `y` (`acyclicSeg(list,y)`) from to the loop invariant, and so, we are done after proving transitivity of acyclic list segments, *i.e.* `acyclicSeg(list,y)` and `acyclicSeg(y,NULL)` implies `acyclicSeg(list,NULL)`. Sadly, static verifiers cannot automatically discharge such inductive proofs, and so we specify the proof steps in `mergeLemma` on lines 39-50. Then, after using the lemma on line 34, we achieve our proof goal.

As we can see, not only do users of static verifiers need to write a number of auxiliary specifications in support of proof goals, the specifications are often more complex compared to the program code itself even for simple examples like `insertLast`. Worse even, is that while users are developing these complex specifications static tools provide limited feedback on their correctness as demonstrated next (§1.4.2).

1.4.2 Lack of Early Specification Feedback

Since static verifiers, like Viper, limit themselves to reasoning about predicates iso-recursively and rely on loop invariants to prove properties about loops, feedback on the correctness of spec-

ifications early in the specification process is limited. For example, consider that a user named Daisy incorrectly specifies the body of `acyclicSeg` (our recursive predicate) as `(s == e) ? acc(s->val) && acc(s->next) && acyclicSeg(s->next,e) : true`, which swaps the branches of the ternary in the correct specification from Fig. 1.2. Let's see how Daisy comes across this error while using Viper to incrementally specify `insertLast` in Fig. 1.3. Each increment from the first to the last (seventh) is highlighted in a different color. The first increment, highlighted in green, specifies the precondition and postcondition of `insertLast` with `acyclic` and `acyclicSeg` (lines 2-4). Since predicates are black boxes in static verifiers, Viper only tells Daisy that there is insufficient permission to access `y->next` in the loop condition on line 10. So, Daisy specifies the required `acc(y->next)` permission (as highlighted in purple for the second increment on line 12) in the loop invariant, which must frame the loop condition. But, alas Viper cannot prove that `acc(y->next)` holds on entry to the loop. Without realizing the branches of `acyclicSeg` are out of order, Daisy expects `acyclicSeg(list,NULL)`'s body to provide `acc(y->next)` at loop entry as `list != NULL` and `y == list`; and so, she unfolds `acyclic(list)` and `acyclicSeg(list,NULL)` on lines 6-7 making up the third specification increment highlighted in pink. Unfortunately, Viper still reports that `acc(y->next)` does not hold on entry to the loop, which alerts Daisy to the problem with `acyclicSeg`.

With Viper, Daisy required three specification increments to detect a bug in the first unfolding of `acyclicSeg`, and this problem gets worse the deeper the bug is in the recursive predicate. For example, consider now that `acyclicSeg`'s body is incorrectly specified as `(s == e)? true : acc(s->val) && acc(s->next) && acyclicSeg(e,s->next)`, which swaps `s->next` and `e` in the recursive call to `acyclicSeg`. As a result, `acyclicSeg` asserts in lock-step that the nodes in lists `s` and `e` are accessible and separated—which is not the intended behavior of `acyclicSeg`—and `acyclicSeg` now always fails when `e` reaches its end, *i.e.* is `NULL`, as it tries to assert `acc(e->val)` and `acc(e->next)`. It takes until the fourth specification increment highlighted in blue to discover that `acyclicSeg` is incorrectly specified using Viper. As before, Daisy is led to specifying the first three increments by Viper's error messages that first require `acc(y->next)` in the loop invariant and then require `acc(y->next)` to hold on entry to the loop. This time, however, `acyclicSeg(list,NULL)`'s body contains `acc(y->next)` when `list != NULL`, so Viper can prove `acc(y->next)` holds on loop entry and instead reports that the loop invariant `acc(y->next)` might not be preserved by the loop body. Daisy recalls the loop iterates over all nodes in the list with the current node being `y`, so preserving `acc(y->next)` in the loop is the same as showing that Viper has accessibility predicates for every node in the list. As a result, she specifies the fourth increment (lines 12-13 and 17), which continuously unfolds the `acyclicSeg(list,NULL)` predicate on every iteration of the loop and captures the information in its body in the loop invariant. Alas, Viper reports that the new loop invariant does not hold on entry as it cannot prove `acyclicSeg(y->next,NULL)` holds here. Since this information should come from unfolding `acyclicSeg(list,NULL)` on line 7, Daisy takes another look at `acyclicSeg`'s body and discovers her specification error. That is, it takes four specification increments for Daisy to realize her mistake and the fourth increment required her to think deeply about her while loop.

Clearly, static verifiers burden their users, like Daisy, by requiring them to write a number of complex auxiliary specifications both for proofs and to receive useful feedback on the correctness of their specifications. Fortunately, as we will show next in §1.5, gradual verification overcomes

this burden by smoothly supporting the spectrum between static and dynamic checking.

1.5 How Gradual Verification Can Help

In this section, we show how gradual verification's ability to smoothly integrate static and dynamic checking allows users to overcome specification burdens inherent to static verification, *e.g.* that users have to write complex auxiliary specifications in support of proofs and to receive useful feedback on the correctness of their specifications.

1.5.1 Ignore Auxiliary Specifications with Gradual Verification

In §1.4.1, we saw how static verifiers force users to write complex auxiliary specifications (like folds and unfolds, loop invariants, and inductive lemmas) in support of proof goals. In contrast, gradual verification allows users to write as many or as few auxiliary specifications as they want, and instead utilizes dynamic verification to check proof obligations not discharged statically due to missing specifications. For example, consider Fig. 1.3, in which our user Daisy incrementally specifies `insertLast` for preservation of list acyclicity and non-nullness. With our gradual verifier Gradual C0, Daisy only needs to specify the first increment in green, which contains the pre- and postcondition of `insertLast` on lines 2-4. She can completely avoid specifying the auxiliary specifications in the rest of the increments by instead specifying `?` in the loop invariant on line 10. Then, Gradual C0 uses the allocation statement on line 23 to statically validate the write accesses of `y->next->val` and `y->next->next` on lines 24-25. It can also prove statically that the list after insertion is non-null. All other proof obligations, which ensure memory safety of the while loop and the list after insertion is acyclic, are checked dynamically.

Run-time checking memory safety of the while loop involves three parts: 1) verifying access to `y->next` in the loop condition (line 10), 2) verifying access to `y->next` in the loop body (line 16), and 3) verifying access to `y->next` in the alloc statement directly after the loop (line 23). That is, Gradual C0 asserts ownership of all heap locations in the given list, which are being iterated over by the loop. Consequently, the larger the list the higher the run-time cost of verification for `insertLast`. This cost is unacceptable to Daisy, so she statically specifies memory safety of the while loop in the first four increments in Fig. 1.3 (lines 2-4, 6-7, 12-13, and 17). The new loop invariant (lines 12-13) uses `acyclicSeg` to expose **`acc`**`(y->next)` for verifying access to `y->next` in the loop condition, loop body, and after the loop. The unfolds on lines 6-7 and 17 are used to prove that the loop invariant is preserved by the loop given the precondition on line 2. After all this work, Daisy is not interested in also statically specifying the list after insertion is acyclic, and so, she joins `?` with her newly specified loop invariant. As a result, Gradual C0 now additionally checks memory safety of the while loop statically reducing run-time cost, and only checks `acyclic(\result)` from the postcondition of `insertLast` (line 3) dynamically (*i.e.* the list returned after insertion is acyclic). By allowing Gradual C0 to check `acyclic(\result)` dynamically, Daisy saved herself a lot of specification effort. She avoided building up `acyclicSeg` from the previous end of the list to the new one (increment five in orange, lines 26-28), specifying a more complex loop invariant (increment six in red, lines 9, 11, and 18-19), and stating and proving transitivity of acyclic list segments (increment seven

in yellow, lines 20 and 29). Daisy is very happy to make this human-effort vs run-time cost trade-off.

1.5.2 Gradual Verification Gives Early Feedback with Run-time Checks

As we saw in §1.4.2, static verifiers struggle to provide early feedback on specification errors in predicates. Using Viper, it took until the third specification increment in Fig. 1.3 for Daisy to discover the simple error in `acyclicSeg`'s body, `(s == e)? acc(s->val) && acc(s->next) && acyclicSeg(s->next,e) : true`, which swaps the branches of the ternary in the correct specification from Fig. 1.2. In contrast, with Gradual C0, Daisy easily discovers this error on the first specification increment (in green, lines 2-4), which specifies the pre- and postcondition of `insertLast`. She additionally specifies `?` on the loop invariant and provides a simple test case that calls `insertLast` on a list with one node. Then, Gradual C0 alerts Daisy to the error in `acyclicSeg` by reporting at run time that `acc(s->val)` from `acyclicSeg` does not hold at the end of the list where `s == NULL`. Clearly, `acc(s->val)` will never hold when `s == NULL`, so Daisy realizes she swapped the branches in `acyclicSeg`. Similarly, the second example of an error in `acyclicSeg`'s specification, which swaps `s->next` and `e` in the recursive call to `acyclicSeg` (`(s == e)? true : acc(s->val) && acc(s->next) && acyclicSeg(e,s->next)`), took four specification increments in Fig. 1.3 to be exposed by Viper. Again with Gradual C0, Daisy can detect the error by the first increment as long as she specifies `?` on her loop invariant and supplies a simple test case with a list containing two elements. In this case, Gradual C0 reports at run time that `acc(s->val)` from `acyclicSeg` does not hold for `s` which is `NULL`. Since `acc(s->val)` will never hold when `s == NULL`, Daisy takes another look at `acyclicSeg`'s body and realizes her error. That is, Gradual C0's dynamic checking of partial specifications is helpful for detecting errors in recursive predicates much earlier in the specification process than static verification alone and the errors better capture the inherent problems in the specifications.

To summarize, users of gradual verification may write as many or as little auxiliary specifications as they want and still get sound verification of their code by trading off between human-effort and run-time cost. Users can also receive feedback on the correctness of their specifications much earlier in the specification process than if they used static verification alone and the run-time errors reported often closely match the inherent problems with the specification.

1.6 Related Work

We have already discussed the most-closely related research, including the underlying logics [36, 38, 44], and foundational work on gradual typing and gradual verification [3, 19, 41, 42, 43].

1.6.1 Gradual Typing

Additional related work in gradual typing includes richer type systems such as gradual refinement types [24, 25] and gradual dependent types [15, 28]. These systems focus on pure functional programming, while our gradual verification work targets imperative programs. Therefore they

do not have to consider heap ownership; and they also do not deal with recursive predicates. Combining these approaches with gradual verification in order to account for higher-order stateful programs is a challenging venue for future work. On the other hand, prior work on gradual typestate [18, 49] and gradual ownership [40] integrate static and dynamic checking of ownership of heap data structures. Neither of these efforts considered verifying logical assertions. Both predate the AGT framework that guided our design [19] and the formulation of the gradual guarantees [43]; so, it is unclear whether these guarantees hold in these proposals.

There is an extensive body of work on optimizing run-time checks in gradual type systems. Muehlboeck and Tate [32] show that in languages with nominal type systems, such as Java, gradual typing does not exhibit the usual slowdowns induced by structural types. Feltey et al. [16] reduce run-time overhead from redundant contract checking by contract wrappers. They eliminate unnecessary contract checking by determining—across multiple contract checking boundaries for some datatype or function call—whether some of the contracts being checked imply others. While the performance results in Chpt. 4 are promising, we may be able to draw from the extensive body of work in gradual typing to achieve further performance gains in future work.

1.6.2 Static Verification

Work in static verification contains approaches that try to reduce the specification burden of users—a goal of gradual verification. Furia and Meyer [17] infer loop invariants with heuristics that weaken postconditions into invariants. When that approach fails, verification also fails because invariants are missing. Similarly, several tools (Smallfoot [4], jStar [13], and Chalice [27]) use heuristics to infer fold and unfold statements for verification. In contrast, gradual verification does not fail solely because invariants, folds, or unfolds are missing; imprecision begets optimism. However, our gradual verification techniques may benefit from similar heuristic approaches by leveraging additional static information to further reduce run-time overhead. Incorporating these heuristics in our setting may be challenging due to imprecise specifications, but it is a promising direction for future work.

Additionally, developers can use Dafny's [26] **assume** and **assert** statements to debug specifications similar to how they debug programs with print statements [29]. Unlike gradual verification, this approach does not reduce specification burden and requires manual elicitation of missing specifications needed for verification. Similarly, StaDy [37] relies on a combination of static and dynamic analysis techniques to aide developers with debugging specifications. But, it does not reduce specification burden and does not support recursive data structures.

Abductive reasoning (*abductive inference*) tries to find an explanatory hypothesis for a desired outcome [12]. In static verification, the desired outcome is a proof obligation (O), facts (F) are invariants derived from the program and specifications using some analysis, and the explanatory hypothesis (E) are invariants that do not contradict the derived facts ($\text{SAT}(F \wedge E)$) and are required to discharge the proof obligation ($F \wedge E \models O$). Ideally, F should be sufficient to discharge O, but missing or insufficient specifications often results in F being too weak to prove O leading to false positives (alarms) in tools. So, work in applying abductive reasoning to static verification [5, 6, 8, 9, 12] aims to compute E in order to prioritize—with minimal human intervention—verification failures caused by bugs in a program and de-emphasize false positives (alarms) caused by missing or incomplete specifications. In angelic verification [5, 9] and

Calcagno et al. [6]'s work, entire specifications, such as preconditions, postconditions, and loop invariants are generated as explanatory hypotheses. Dillig et al. [12] instead compute smaller, intermediate formulas as explanatory hypotheses.

Similar to prior abductive reasoning work [5, 6, 9, 12], gradual verification's static system reasons around missing or incomplete specifications to compute facts F as part of imprecise formulas $? \wedge F$. At proof obligations, we compute the weakest formula that can replace $?$ in $? \wedge F \models O$ and $\mathrm{SAT}(? \wedge F)$ successfully. So, like Dillig et al. [12] we compute intermediate explanatory hypotheses rather than whole specifications like Blackshear and Lahiri [5], Calcagno et al. [6], and Das et al. [9]. But, rather than relying on users to validate generated hypotheses [5, 6, 9, 12], we check their correctness at run time. This significantly simplifies their computation—since they do not need to be human readable and can statically mark code as unreachable—and allows our techniques to be sound (prior abduction work is not). However, modifying our weakest formula to be more human readable before run-time checking it may result in easier to understand error messages.

1.6.3 Dynamic Verification

Meyer [30] introduced the Eiffel language, which automatically performs dynamic verification of pre- and postconditions and class invariants in first order logic. Nguyen et al. [34] extended dynamic verification to support separation logic assertions. More recently, Agten et al. [1] applied dynamic checking at the boundaries between statically verified and unverified code to guarantee that no assertion failures or invalid memory accesses occur at run time in any verified code. Their approach improved on Nguyen et al. [34]'s approach in terms of performance by allowing unverified code to read arbitrary memory. Further, unlike Nguyen et al. [34], Agten et al. [1]'s approach only needs access to verified code rather than the entire codebase. As with Nguyen et al. [34]'s work, our gradual verification work supports dynamic verification of ownership and first order logic. Our techniques additionally support run-time checking of recursive predicates. Similarly to Agten et al. [1], gradual verification applies dynamic checking at the boundaries between verified and unverified code to protect verified code and does so systematically. However, in Gradual C0 unverified code must be accessible to the verifier as it is gradually verified as well. Future work in gradual verification should incorporate insights from Agten et al. [1]'s work to avoid requiring entire codebases for verification and to improve verification performance.

1.6.4 Hybrid Verification

Another closely related work is soft contract verification by Nguyen et al. [35], which verifies dynamic contracts statically where possible and dynamically where necessary by utilizing symbolic execution. This hybrid technique does not rely on a notion of *precision*, which is central to gradual approaches and their metatheory [43]—including our gradual verification techniques. Nguyen et al. [35] use symbolic execution results directly to discharge proof obligations where possible, while our gradual verifier Gradual C0 strengthens symbolic execution results to discharge proof obligations adhering to the theory of imprecise formulas we present in this dissertation. Further, Nguyen et al. [35]'s work is targeted at dynamic functional languages, while

our work focuses on imperative languages. We also build in memory safety as a default, while Nguyen et al. [35] do not.

Additionally, Nguyen et al. [34] (discussed previously in dynamic verification related work) leveraged static information to reduce the overhead of their run-time checking approach for separation logic. They do not try to report static verification failures (unlike our gradual verification work), because their technique cannot not distinguish between failures due to inconsistent specifications and failures due to incomplete specifications (unlike our gradual verification work). Also, their run-time checking approach forces developers to specify matching heap footprints in pre- and postconditions to avoid false negatives; meanwhile, with gradual verification pre- and postconditions may contain the same or different specifications for heap footprints or may omit such specifications completely.

Chapter 2

Formal Foundations

In this chapter, we lay out the formal foundations of gradual verification for recursive heap data structures; and in particular, present the design, formalization, and meta-theory of a sound gradual verifier for programs that manipulate such data structures. Our approach follows Bader et al. [3]'s methodology (inspired by the Abstracting Gradual Typing (AGT) methodology from gradual typing [19]), but starts from a static verifier with *implicit dynamic frames* (IDF) [44] and recursive abstract predicates [36]. This more sophisticated setting requires us to address the following technical challenges:

- Imprecise specifications may be strengthened not just with boolean assertions about arithmetic expressions, but also with both *abstract predicates* and *accessibility predicates*, which denote ownership of heap locations. Our strengthening definition also includes *self-framing*, a well-formedness condition required by IDF [44].
- Both accessibility predicates and abstract predicates must potentially be verified dynamically. Our system verifies accessibility predicates at run time by tracking and updating a set of owned heap locations. We verify recursive abstract predicates by executing them as recursive boolean functions. This run-time semantics corresponds to an *equi-recursive* interpretation of abstract predicates, contrasting with the *iso-recursive* interpretation used in static verifiers [45]; our theory ensures that these interpretations are consistent.

We show that the resulting gradual verifier is sound, that it is a *conservative extension* of the static verifier—meaning that both coincide on programs with fully-precise specifications—and that it adheres to the *gradual guarantee*. This guarantee, originally formulated for gradual type systems [43], captures the intuition that relaxing specifications should not introduce new (static or dynamic) verification errors.

The rest of this chapter is outlined as follows. In §2.1 we formally present a statically verified language (SVL_{RP}) supporting a propositional specification logic extended with IDF and recursive predicates. We gradualize the static semantics of this language in §2.2 and dynamic semantics in §2.3. §2.4 formally defines the aforementioned gradual properties that the resulting gradual verifier (*i.e.* the gradual language GVL_{RP}) adheres to. The handwritten proofs of all propositions and lemmas given in this chapter can be found in the extended version of a conference paper publishing the results of this chapter [48].

$$
\begin{array}{rcll}
x, y, z & \in & \mathit{VAR} & \textit{(variables)} \\
v & \in & \mathit{VAL} & \textit{(values)} \\
e & \in & \mathit{EXPR} & \textit{(expressions)} \\
s & \in & \mathit{STMT} & \textit{(statements)} \\
o & \in & \mathit{LOC} & \textit{(object Ids)} \\
\mathtt{P} & ::= & \overline{\mathtt{cls}}\ s & \\
\mathtt{cls} & ::= & \mathtt{class}\ C\ \{\ \overline{\mathtt{field}}\ \overline{\mathtt{pred}}\ \overline{\mathtt{method}}\ \} & \\
\mathtt{field} & ::= & T\ f; & \\
\mathtt{pred} & ::= & \mathtt{predicate}\ p\,(\overline{T\,x}) = \theta & \\
T & ::= & \mathtt{Int} \mid \mathtt{Bool} \mid C \mid \top & \\
\mathtt{method} & ::= & T\ m(\overline{T\,x})\ \mathtt{contract}\ \{s\} & \\
\mathtt{contract} & ::= & \mathtt{requires}\ \theta\ \mathtt{ensures}\ \theta & \\
\oplus & ::= & + \mid - \mid * \mid \backslash & \\
\odot & ::= & \neq \mid = \mid < \mid > \mid \leq \mid \geq &
\end{array}
$$

$$
\begin{array}{rcll}
f & \in & \mathit{FIELDNAME} & \textit{(field names)} \\
m & \in & \mathit{METHODNAME} & \textit{(method names)} \\
C & \in & \mathit{CLASSNAME} & \textit{(class names)} \\
p & \in & \mathit{PREDNAME} & \textit{(predicate names)} \\
s & ::= & \mathtt{skip} \mid s\mathtt{;}\ s \mid T\,x \mid x := e \mid x.f := y & \\
 & & \mid \mathtt{if}\,(e)\,\{\,s\,\}\,\mathtt{else}\,\{\,s\,\} & \\
 & & \mid \mathtt{while}\ (e)\ \mathtt{inv}\,\theta\,\{\,s\,\} \mid x := \mathtt{new}\,C & \\
 & & \mid y := z.m(\overline{x}) \mid \mathtt{assert}\ \phi \mid \mathtt{fold}\,p(\overline{e}) & \\
 & & \mid \mathtt{unfold}\,p(\overline{e}) & \\
e & ::= & v \mid x \mid e \oplus e \mid e \odot e \mid e.f & \\
x & ::= & \mathtt{result} \mid id \mid \mathbf{old}(id) \mid \mathtt{this} & \\
v & ::= & n \mid o \mid \mathtt{null} \mid \mathtt{true} \mid \mathtt{false} & \\
\phi & ::= & \mathtt{true} \mid \mathtt{false} \mid e \odot e \mid p(\overline{e}) & \\
 & & \mid \mathbf{acc}(e.f) \mid \mathtt{if}\ e\ \mathtt{then}\ \phi\ \mathtt{else}\ \phi & \\
 & & \mid \mathbf{unfolding}\ p(\overline{e})\ \mathbf{in}\ \phi \mid \phi \wedge \phi \mid \phi * \phi & \\
\theta & ::= & \text{self-framed}\ \phi &
\end{array}
$$

Figure 2.1: SVL_{RP}: Syntax

2.1 SVL_{RP}

Following the AGT methodology [19], gradual verification is built on top of static verification. Therefore, we first formally present a statically verified language supporting a propositional specification logic extended with implicit dynamic frames (IDF) and recursive abstract predicates. This language, called SVL_{RP}, is directly inspired by Summers and Drossopoulou [45]. Readers familiar with static verification might want to read through this section anyway, because it sets up notations and key concepts used in the gradualization in §2.2.

2.1.1 Syntax & Static Semantics

The complete syntax of SVL_{RP} can be found in Fig. 2.1. Programs consist of classes and statements. Classes contain publicly accessible fields, predicates, and methods. Statements include the empty statement, sequences, variable declarations, variable and field assignments, conditionals, while loops, object allocations, method calls, assertions, as well as fold and unfold statements. Expressions can appear in specifications, and therefore cannot modify the heap. They consist of literal values (integers, objects, `null`, and booleans), variables, arithmetic expressions, comparisons, and field accesses. Methods have contracts consisting of pre- and postconditions, which are formulas represented by ϕ. Formulas join boolean values, comparisons, predicate instances, accessibility predicates, conditionals, and unfoldings via the non-separating conjunction $\wedge$ or the separating conjunction $*$ (from IDF). Note that θ refers to a *self-framed* formula [44], formally defined in §2.1.2. We require pre- and postconditions, predicate definitions, and loop invariants to be self-framed.

Looking ahead to gradual verification, we would like formulas to be efficiently evaluable at run time—and in the presence of accessibility predicates, efficient evaluation requires knowing which branch of a disjunction to evaluate. Therefore, we include a conditional `if` construct in

formulas instead of disjunction $\vee$.

As the focus of this work is not on typing, we only consider well-formed and well-typed programs, which is standard and not formalized here. Additionally, variables are declared and initialized before use, and class, predicate, and method names are unique. Finally, contracts should only contain variables that are in scope: a precondition can only contain the method's parameters $\overline{x_i}$ and `this` and a postcondition can only contain the special variable `result`, `this`, and dummy variables $\overline{\texttt{old}(x_i)}$.

2.1.2 Formula Semantics

In this section, we give meaning to formulas in SVL$_{\text{RP}}$. We also give related definitions for formula satisfiability, implication, footprint computation, and framing. The semantics and related definitions are inspired by Summers and Drossopoulou [45] and Bader et al. [3].

Equi-Recursive Evaluation. Evaluating the truth of formulas requires a heap H, a variable environment $\rho \in \textsc{Env}$, and a dynamic footprint $\pi \in \textsc{DynFprint} = \mathcal{P}(\textsc{Loc} \times \textsc{FieldName})$. A heap H is a partial function from heap locations to a value mapping of object fields, *i.e.* $\textsc{Heap} = \textsc{Loc} \rightharpoonup (\textsc{FieldName} \rightharpoonup \textsc{Val})$. Additionally, we introduce a big-step evaluation relation for expressions $H, \rho \vdash e \Downarrow v$, which is standard. An expression e is evaluated according to $H, \rho \vdash e \Downarrow v$ yielding value v. The heap H is used to look up fields and the local variable environment ρ to look up variables.

Then, formula evaluation $\cdot \models_E \cdot \subseteq \textsc{Mem} \times \textsc{Formula}$ determines the truth of a formula using heap H, variable environment ρ, dynamic footprint π, and an equi-recursive interpretation of predicate instances. Select rules for formula evaluation are given in Fig. 2.2 (complete rules are in the Appendix Fig. A.2). EvAcc checks whether access demanded by a formula is provided by the dynamic footprint, *e.g.* `acc(l.val)` where `l` points to `o` is true when $\langle \texttt{o}, \texttt{val} \rangle \in \pi$. EvSepOp checks two separated subformulas against disjoint partitions of the dynamic footprint. This ensures that access to the same field is not granted twice; for instance, this ensures that $\texttt{acc(l}_1\texttt{.val)} * \texttt{acc(l}_2\texttt{.val)}$ references two distinct fields. In contrast, the rule for $\wedge$ (EvAndOp) checks both operands against the full dynamic footprint; therefore, $\texttt{acc(l}_1\texttt{.val)} \wedge \texttt{acc(l}_2\texttt{.val)}$ may reference the same fields. Further, EvPred checks the complete unrolling of a predicate instance using the function $\mathsf{body}_\mu : \textsc{PredName} \to \textsc{Expr}^* \to \textsc{SfrmFormula}$. Given a predicate name and arguments, this function returns the predicate's definition (from the ambient program[1]) after parameter substitution. We make sure that every argument e_i produces a value, only in order to line up with the iso-recursive semantics. But we do not need to substitute the values into $\mathsf{body}_\mu(p)(e_1, ..., e_n)$, because it already has the e_i's within it after parameter substitution. Finally, the rule for an unfolding (EvUnfolding) ignores the predicate unfolding, because it is an iso-recursive only construct. For example, `unfolding foo(x) in x.f >= 2` is true exactly when `x.f >= 2` is true. Also, all the construct does is provide access to more heap locations in the predicate. The other rules are as expected.

Iso-Recursive Evaluation. We also introduce an iso-recursive formula evaluation semantics,

[1]Many relations we define are implicitly parameterized over the ambient program.

$$\frac{H, \rho \vdash e \Downarrow o \qquad H, \rho \vdash e\,.\,f \Downarrow v \qquad \langle o, f \rangle \in \pi}{\langle H, \rho, \pi \rangle \vDash_E \texttt{acc}(e.f)} \textsc{EvAcc} \quad \frac{\langle H, \rho, \pi_1 \rangle \vDash_E \phi_1 \qquad \langle H, \rho, \pi_2 \rangle \vDash_E \phi_2}{\langle H, \rho, \pi_1 \uplus \pi_2 \rangle \vDash_E \phi_1 * \phi_2} \textsc{EvSepOp}$$

$$\frac{H, \rho \vdash e_1 \Downarrow v_1 \qquad ... \qquad H, \rho \vdash e_n \Downarrow v_n \qquad \langle H, \rho, \pi \rangle \vDash_E \mathsf{body}_\mu(p)(e_1, ..., e_n)}{\langle H, \rho, \pi \rangle \vDash_E p\,(e_1, ..., e_n)} \textsc{EvPred}$$

$$\frac{\langle H, \rho, \pi \rangle \vDash_E \phi}{\langle H, \rho, \pi \rangle \vDash_E \texttt{unfolding}\ p(e_1, ..., e_n)\ \texttt{in}\ \phi} \textsc{EvUnfolding}$$

Figure 2.2: SVL_{RP}: Formula evaluation (select rules)

$$\frac{H, \rho \vdash e_1 \Downarrow v_1 \qquad ... \qquad H, \rho \vdash e_n \Downarrow v_n \qquad \langle p, v_1, ..., v_n \rangle \in \Pi}{\langle H, \rho, \Pi \rangle \vDash_I p(e_1, ..., e_n)} \textsc{EvPred}$$

Figure 2.3: SVL_{RP}: Iso-recursive formula evaluation (select rule)

used in static verification. This semantics differs from its equi-recursive counterpart, described previously and given in Fig. 2.2, on the EvPred rule. Fig. 2.3 presents the iso-recursive version of EvPred. It treats predicate instances as opaque permissions by checking whether a predicate instance demanded by a formula is justified by a dynamic permission set $\Pi \in \textsc{Permissions} = \mathcal{P}((\textsc{Loc} \times \textsc{FieldName}) \cup (\textsc{PredName} \times \textsc{Val}^*))$. Compared to a dynamic footprint, a dynamic permission set can contain dynamic predicate instances in addition to heap locations. For example, `acyclic(l)` where `l` points to `o` is true when $\langle \texttt{acyclic}, \texttt{o} \rangle \in \Pi$. Other than EvPred, the iso-recursive semantics is simply defined by replacing π in the equi-recursive rules with Π.

Formula Satisfiability and Implication. Similar to SVL [3], formal definitions for formula satisfiability and implication rely on sets of H, ρ, and Π tuples that make formulas true. Definition 2.1.1 presents a function that produces these sets from formulas. Definitions 2.1.2 and 2.1.3 rely on Definition 2.1.1 to formally state formula satisfiability and implication respectively. Note that these definitions are iso-recursive in order to be implementable in static verification tools.

Definition 2.1.1 (Denotational Formula Semantics)**.**
$[\![\cdot]\!] : \textsc{Formula} \rightarrow \mathcal{P}(\textsc{Heap} \times \textsc{Env} \times \textsc{Permissions})$
$[\![\phi]\!] \stackrel{\mathsf{def}}{=} \{ \langle H, \rho, \Pi \rangle \in \textsc{Heap} \times \textsc{Env} \times \textsc{Permissions} \mid \langle H, \rho, \Pi \rangle \vDash_I \phi \}$

Definition 2.1.2 (Formula Satisfiability)**.** *A formula ϕ is satisfiable if and only if $[\![\phi]\!] \neq \emptyset$. Let* $\textsc{SatFormula} \subset \textsc{Formula}$ *be the set of satisfiable formulas.*
Ex. $\texttt{acc}(\texttt{l}_1.\texttt{val}) * \texttt{acc}(\texttt{l}_2.\texttt{val})$ *is satisfiable since* $\texttt{l}_1$ *may not equal* $\texttt{l}_2$.
In contrast, $\texttt{acc}(\texttt{l}_1.\texttt{val}) * \texttt{acc}(\texttt{l}_2.\texttt{val}) * \texttt{l}_1 = \texttt{l}_2$ *is unsatisfiable.*

Definition 2.1.3 (Formula Implication)**.** $\phi_1 \Rightarrow \phi_2$ *if and only if* $[\![\phi_1]\!] \subseteq [\![\phi_2]\!]$.
Ex. $\texttt{l.val} = 6 \Rightarrow \texttt{l.val} \geq 5$, *and* $\texttt{l.val} = 6 \not\Rightarrow \texttt{acc}(\texttt{l.val}) * \texttt{l.val} \geq 5$ *since* $\texttt{acc}(\texttt{l.val})$ *is missing on the left-hand side.*

$$\frac{\langle H, \rho, \Pi \rangle \vDash_I \textbf{acc}(e.f) \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e\,.\,f} \text{ FrmField}$$

$$\frac{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \textbf{acc}(e.f)} \text{ FrmAcc} \qquad \frac{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \phi_1 \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \phi_2}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \phi_1 * \phi_2} \text{ FrmSepOp}$$

$$\frac{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_1 \qquad ... \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_n}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} p(e_1, ..., e_n)} \text{ FrmPred}$$

$$\frac{\begin{array}{c}\langle H, \rho, \Pi \rangle \vDash_I p(e_1, ..., e_n) \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_1 \qquad ... \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_n \\ \langle H, \rho, \Pi' \rangle \vdash_{\texttt{frmI}} \phi \qquad \Pi' = \Pi \cup \lfloor \mathsf{body}_\mu(p)(e_1, ..., e_n) \rfloor_{H,\rho}\end{array}}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \textbf{unfolding } p(e_1, ..., e_n) \textbf{ in } \phi} \text{ FrmUnfolding}$$

Figure 2.4: SVL_{RP}: Framing (select rules)

Footprints and Framing. A statically-verified language supporting IDF requires formal definitions for the footprint and framing of a formula. These definitions are also iso-recursive.

The *footprint* of a formula ϕ, denoted $\lfloor \phi \rfloor_{H,\rho}$, is simply the minimum set of permissions Π required to satisfy ϕ given a heap H and variable environment ρ:

$$\lfloor \phi \rfloor_{H,\rho} = \min \; \{ \; \Pi \in \text{Permissions} \mid \langle H, \rho, \Pi \rangle \vDash_I \phi \; \}$$

The footprint is defined (*i.e.* there exists a unique minimal set of permission Π) for formulas satisfiable under H and ρ. It can be more directly implemented by simply evaluating ϕ using H and ρ, granting and recording precisely the permissions required. The footprint of `l.val >= 2` is empty, while the footprint of **acc**`(l.val) * l.val >= 2` is $\{\langle \texttt{o}, \texttt{val} \rangle\}$ when `l` points to `o`.

A formula is said to be *framed* by permissions Π iff it only mentions fields and unfolds predicates in Π. We give select inference rules for formula framing in Fig. 2.4 and give the rest in the Appendix Fig. A.3. Note that FrmUnfolding allows one unrolling of a predicate to frame a part of a formula. Now, formula ϕ is called *self-framed* (we write $\vdash_{\texttt{frm}} \phi$) if for all H, ρ, Π $\langle H, \rho, \Pi \rangle \vDash_I \phi$ implies $\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \phi$. We define the set of self-framed formulas $\text{SfrmFormula} \overset{\text{def}}{=} \{ \; \phi \in \text{Formula} \mid \; \vdash_{\texttt{frm}} \phi \; \}$. `l.val >= 2` is not self-framed, because it can evaluate to `true` even when Π does not contain **acc**`(l.val)`. On the other hand, **acc**`(l.val) * l.val >= 2` is self-framed, because it does not evaluate to `true` unless Π contains **acc**`(l.val)`. Similarly, `unfolding acyclic(l) in l.val >= 2` is not self-framed while `acyclic(l) * unfolding acyclic(l) in l.val >= 2` is for `acyclic(l)` with body **acc**`(l.val)`. We write θ to denote self-framed formulas.

Relating Permission Sets and Footprints. By using the *footprint* defined previously, we can formally relate dynamic permission sets to dynamic footprints via the partial function $\langle\langle \; \cdot \; \rangle\rangle_H$ of type $\text{Permissions} \times \text{Heap} \rightharpoonup \text{DynFprint}$:

$$\langle\langle\ \Pi\ \rangle\rangle_H = \{\langle o, f\rangle \mid \langle o, f\rangle \in \Pi\} \cup \langle\langle\ \Pi'\ \rangle\rangle_H \qquad \texttt{where}\ \Pi' = \cup_{\langle p, v_1, ..., v_n\rangle \in \Pi} \lfloor \mathsf{body}_\mu(p)(v_1, ..., v_n) \rfloor_{H,[]}$$

This function completely unrolls the predicate instances in a dynamic permission set gathering owned heap locations on the way. For example, given $\langle \texttt{acyclic}, \texttt{o}\rangle$, with `acyclic` defined precisely as in Fig. 1.2, this function returns all the heap locations ($\{\langle \texttt{o}, \texttt{val}\rangle,\ \langle \texttt{o}, \texttt{next}\rangle,\ \langle \texttt{o.next}, \texttt{val}\rangle,\ \langle \texttt{o.next}, \texttt{next}\rangle,\ \ldots\}$) in the list `o`. Note that $\langle\langle\ \cdot\ \rangle\rangle_H$ is only defined when predicates can be finitely unrolled.

2.1.3 Static Verification

Static verification relies on a *weakest liberal precondition calculus* [10] to generate verification conditions. We now present this WLP calculus, which is defined iso-recursively.

WLP Calculus. Select rules for a weakest liberal precondition function $\mathsf{WLP}(s, \theta)$ of type $\textsc{Stmt} \times (\textsc{SatFormula} \cap \textsc{SfrmFormula}) \rightharpoonup (\textsc{SatFormula} \cap \textsc{SfrmFormula})$ are given in Fig. 2.5 (all rules are in the Appendix Fig. A.5). Note, we explicitly restrict the domain and codomain of the WLP function to contain only satisfiable and self-framed formulas. These restrictions are often ensured in Fig. 2.5 by finding a maximum self-framed and satisfiable formula with respect to implication (the weakest formula).

The statement-specific rules for WLP are standard, save for specific care related to field accesses, accessibility predicates, and predicate instances. Rules for variable and field assignment, conditionals, and while loops produce accessibility predicates for field accesses in the program statement, *e.g.* the WLP for `y := l.val` must contain **`acc`**`(l.val)`. Some rules rely on the function $\mathsf{acc}(e) : \textsc{Expr} \to \textsc{Formula}$, which returns a formula of accessibility predicates corresponding to field accesses in e. More interestingly, the rule for a method call frames off information in the method's postcondition from θ producing the frame θ_f. If the accessibility predicates and predicate instances in θ_f are not in the method's precondition, then θ_f is joined with the precondition to produce the WLP. Consider computing the WLP for a call to `this.foo()` where foo's precondition is `bar(this) * unfolding bar(this) in this.val >= 2` and postcondition is `bar(this)` and θ `= bar(this)`. Therefore, θ_f `= true` and the WLP is `this != null * bar(this) * unfolding bar(this) in this.val >= 2 * true`.

Static Verification. A SVL_{RP} program is statically verified if it is a *valid program*:

Definition 2.1.4 (Valid Method)**.** *A method with contract* `requires` θ_p `ensures` θ_q*, parameters* $\overline{x}$*, and body* s *is considered valid if* $\theta_p \Rightarrow \mathsf{WLP}(s, \theta_q)[\overline{x/\texttt{old}(x)}]$ *holds.*

Definition 2.1.5 (Valid Program)**.** *A program with entry point statement* s *is considered valid if* `true` $\Rightarrow \mathsf{WLP}(s, \texttt{true})$ *holds,* $\theta_i \wedge$ `(e = true)` $\Rightarrow \mathsf{WLP}(r, \theta_i)$ *and* $\theta_i \Rightarrow \mathsf{acc}(e)$ *hold for all loops with condition* e*, body* r*, and invariant* θ_i*, and all methods are valid.*

$$
\begin{aligned}
\mathsf{WLP}(x := e, \theta) &= \max_{\Rightarrow} \{ \theta' \mid \theta' \Rightarrow \theta[e/x] \quad \land \quad \theta' \Rightarrow \mathsf{acc}(e) \} \\
\mathsf{WLP}(x.f := y, \theta) &= \mathbf{acc}(x.f) \quad \land \quad \theta[y/x.f] \\
\mathsf{WLP}(y := z.m(\overline{x}), \theta) &= \max_{\Rightarrow} \{ \theta' \mid \ y \notin \mathsf{FV}(\theta') \quad \land \\
&\quad \exists \theta_f \,.\, \theta' \Rightarrow (z \neq \texttt{null}) * \mathsf{mpre}(m)[z/\texttt{this}, \overline{x/\mathsf{mparam}(m)}] * \theta_f \quad \land \\
&\quad \theta_f * \mathsf{mpost}(m)[z/\texttt{this}, \overline{x/\mathbf{old}(\mathsf{mparam}(m))}, y/\texttt{result}] \Rightarrow \theta \}
\end{aligned}
$$

Figure 2.5: SVL_{RP}: Weakest liberal precondition calculus (select rules)

2.1.4 Dynamic Semantics

The soundness of static verification is relative to SVL_{RP}'s dynamic semantics, which we now expose.

Program States. Program states consist of a heap and a stack, *i.e.* $\text{STATE} = \text{HEAP} \times \text{STACK}$. A stack is made of stack frames that contain a variable environment $\rho \in \text{ENV}$, a dynamic footprint $\pi \in \text{DYNFPRINT} = \mathcal{P}(\text{LOC} \times \text{FIELDNAME})$, and a program statement $s \in \text{STMT}$:

$$S \in \text{STACK} ::= E \cdot S \mid \mathsf{nil} \qquad \text{where} \qquad E \in \text{STACKFRAME} = \text{ENV} \times \text{DYNFPRINT} \times \text{STMT}$$

During execution of an SVL_{RP} program, expressions and statements operate on the topmost variable environment ρ. Expressions and statements may additionally access and mutate the heap as long as the topmost dynamic footprint contains the corresponding object-field permissions. Thus, the memory accessible at any point of execution can be viewed as a tuple of type $\text{MEM} = \text{HEAP} \times \text{ENV} \times \text{DYNFPRINT}$.

Reduction Rules. Fig. 2.6 presents select rules for SVL_{RP}'s small-step semantics $\cdot \longrightarrow \cdot \subseteq \text{STATE} \times \text{STATE}$. Complete rules are in the Appendix Fig. A.6. Notably, we structure the rules so as to not require a sequence rule. This aligns the small-step semantics more closely with the WLP calculus, and makes the SVL_{RP} soundness proof easier.

The semantics gets stuck when a statement accesses a field that the current state does not own, as specified in SSASSIGN. Notice that SSASSIGN relies on $\mathsf{acc}(e)$ to check the accessibility of field accesses on the right-hand side. The semantics also gets stuck when preconditions (SSCALL), postconditions (SSCALLFINISH), loop invariants, or assertions (SSASSERT) do not hold.

To determine whether a field access is valid at runtime, the semantics tracks a set of owned heap locations π. This set is expanded during allocation with heap locations for the object's fields. At a method call (SSCALL) π is split into disjoint caller and callee sets using the method's precondition. The *callee set* π' is derived from the precondition's accessibility predicates and the accessibility predicates gained from unrolling the predicates in the precondition. Ownership of the heap locations in π' is passed to the callee, so the *caller set* is defined as $\pi \setminus \pi'$. After execution of the callee's body finishes (SSCALLFINISH), execution resumes at the call site. The

$$\frac{\langle H, \rho, \pi\rangle \vDash_E \phi}{\langle H, \langle \rho, \pi, \texttt{assert}\ \phi\texttt{;}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho, \pi, s\rangle \cdot S\rangle}\ \textsc{SsAssert}$$

$$\frac{\langle H, \rho, \pi\rangle \vDash_E \mathsf{acc}(e) \qquad H, \rho \vdash e \Downarrow v \qquad \rho' = \rho[x \mapsto v]}{\langle H, \langle \rho, \pi, x\ \texttt{:=}\ e\texttt{;}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho', \pi, s\rangle \cdot S\rangle}\ \textsc{SsAssign}$$

$$\frac{\begin{array}{c}\mathsf{method}(m) = T_r\ m\,(\overline{T\ x'})\ \texttt{requires}\ \theta_p\ \texttt{ensures}\ \theta_q\ \{\ r\ \} \qquad H, \rho \vdash z \Downarrow o \qquad \overline{H, \rho \vdash x \Downarrow v} \\ \rho' = [\texttt{this} \mapsto o, \overline{x' \mapsto v}, \overline{\textbf{old}\,(x') \mapsto v}] \qquad \pi' = \langle\langle\ \lfloor \theta_p \rfloor_{H,\rho'}\ \rangle\rangle_H \qquad \pi' \subseteq \pi \qquad \langle H, \rho', \pi'\rangle \vDash_E \theta_p\end{array}}{\langle H, \langle \rho, \pi, y\ \texttt{:=}\ z.m(\overline{x})\texttt{;}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho', \pi', r\texttt{;}\ \texttt{skip}\rangle \cdot \langle \rho, \pi \backslash \pi', y\ \texttt{:=}\ z.m(\overline{x})\texttt{;}\ s\rangle \cdot S\rangle}\ \textsc{SsCall}$$

$$\frac{\mathsf{mpost}(m) = \theta_q \qquad \langle H, \rho', \pi'\rangle \vDash_E \theta_q \qquad \rho'' = \rho[y \mapsto \rho'(\texttt{result})]}{\langle H, \langle \rho', \pi', \texttt{skip}\rangle \cdot \langle \rho, \pi, y\ \texttt{:=}\ z.m(\overline{x})\texttt{;}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho'', \pi \cup \pi', s\rangle \cdot S\rangle}\ \textsc{SsCallFinish}$$

$$\frac{}{\langle H, \langle \rho, \pi, \texttt{fold}\ p\,\texttt{(}e_1\texttt{,}...\texttt{,}e_n\texttt{)}\texttt{;}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho, \pi, s\rangle \cdot S\rangle}\ \textsc{SsFold}$$

Figure 2.6: SVL_{RP}: Small-step semantics (select rules)

callee returns to the call site ownership of all received heap locations and new heap locations gained during execution.

Notice that we treat predicates equi-recursively when we track π, determine whether field accesses are valid, and determine whether contracts, loop invariants, or assertions hold. We also treat folds and unfolds equi-recursively as skip statements (SsFold). SVL_{RP}'s dynamic semantics is equi-recursively defined so the gradual verifier, which builds on SVL_{RP}'s semantics, adheres to the dynamic gradual guarantee (as discussed in §1.3.2).

2.1.5 Soundness

As explained above, the dynamic semantics of SVL_{RP} is designed to get stuck when assertions, method contracts, or loop invariants are violated during program execution. The dynamic semantics also gets stuck if a program accesses fields it does not own during execution. Thus informally, *soundness* says that valid SVL_{RP} programs do not get stuck, *i.e.* verified programs respect program specifications at run time. Just as with SVL from Bader et al. [3], we use a syntactic statement of soundness via progress and preservation.

Now, we introduce the formal definition of a valid state in Definition 2.1.6. This definition is an invariant that relates the static verification and dynamic semantics of valid SVL_{RP} programs. It also relates the formal statements of progress and preservation in Propositions 2.1.7 and 2.1.8. Informally, if the current program state satisfies the WLP of a program, then execution does not get stuck (progress), and after each step of execution, the new state satisfies the WLP of the remaining program (preservation).

Definition 2.1.6 (Valid State, Final State)**.**
We call the state $\langle H, \langle \rho_n, \pi_n, s_n\rangle \cdot ... \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \in \textsc{State}$ valid *if*
$s_n = s\texttt{;}\ \texttt{skip}$ *or* $\texttt{skip}$ *for some* $s \in \textsc{Stmt}$*,*
$s_i = s_i'\texttt{;}\ \texttt{skip}$ *for some* $s_i' \in \textsc{Stmt}$ *for all* $1 \leq i < n$*,*

$\pi_i \cap \pi_j = \emptyset$ *for all* $1 \leq i \leq n$*,* $1 \leq j \leq n$ *such that* $i \neq j$*,*
and $\langle H, \rho_i, \pi_i \rangle \vDash_E \mathsf{sWLP}_i(s_n \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ *for all* $1 \leq i \leq n$ *(*$\mathsf{sWLP}_i(\overline{s}, \theta)$ *is the i-th component of* $\mathsf{sWLP}(\overline{s}, \theta)$*).*
A state ψ *is* final *if* $\psi = \langle H, \langle \rho, \pi, skip \rangle \cdot \mathsf{nil} \rangle$ *for some* H, ρ, π*.*

Note that the definition above relies on sWLP, a stack-aware extension of WLP (defined in the Appendix Fig. A.7). sWLP ensures that access permissions are not duplicated in different stack frames. Program validity (Def. 2.1.5) gives the validity of the initial program state.

Proposition 2.1.7 (SVL$_{\mathrm{RP}}$ Progress)**.** *If* ψ *is a valid non-final state then* $\psi \longrightarrow \psi'$ *for some* ψ'*.*

Proposition 2.1.8 (SVL$_{\mathrm{RP}}$ Preservation)**.** *If* ψ *is a valid state and* $\psi \longrightarrow \psi'$ *for some* ψ' *then* ψ' *is a valid state.*

2.2 GVL$_{\mathrm{RP}}$: Static Semantics

We now derive GVL$_{\mathrm{RP}}$, the gradually-verified language counterpart of SVL$_{\mathrm{RP}}$, essentially following the Abstracting Gradual Typing methodology [19], whose main principles and mechanisms apply beyond type systems. This section presents the syntax and static semantics of GVL$_{\mathrm{RP}}$. §2.3 develops the run-time semantics, and §2.4 establishes the main properties of GVL$_{\mathrm{RP}}$.

2.2.1 Syntax

The syntax of GVL$_{\mathrm{RP}}$ is the same as SVL$_{\mathrm{RP}}$ except for the addition of gradual formulas $\widetilde{\phi}$. Gradual formulas replace formulas θ in method contracts, predicate definitions, and loop invariants:

$$\texttt{pred} ::= \texttt{predicate}\ p(\overline{T\,x}) = \widetilde{\phi} \qquad s ::= ... \mid \texttt{while}\ (e)\ \texttt{inv}\ \widetilde{\phi}\ \{\ s\ \}$$
$$\texttt{contract} ::= \texttt{requires}\ \widetilde{\phi}\ \texttt{ensures}\ \widetilde{\phi} \qquad \widetilde{\phi} ::= \theta \mid ? * \phi$$

A gradual formula is either a self-framed *syntactically precise formula* θ or an imprecise formula `? *` ϕ. Note that the static part of an imprecise formula does not need to be self-framed and `?` is syntactic sugar for `? * true`. Additionally, the set of all gradual formulas is given by $\widetilde{\text{FORMULA}}$. A *syntactically* precise formula does not contain `?` directly, *i.e.* it is not visibly partial. However, it may contain hidden `?`s by containing predicates that, when unrolled, expose `?`, *e.g.* `acyclic(l)` where `acyclic`'s body is `?`. Self-framing is augmented to handle nested imprecision in GVL$_{\mathrm{RP}}$, and its new definition is given in §2.2.2. We will refer to formulas that do not contain `?`, neither directly nor nested in predicates, as *semantically* precise formulas, *e.g.* `acyclic(l)` where `acyclic` is from Fig. 1.2. Note that all semantically precise formulas are syntactically precise, but not all syntactically precise formulas are semantically precise.

2.2.2 Framing

Definitions for framing and self-framing syntactically precise formulas in GVL$_{\mathrm{RP}}$ are redefined to handle imprecise predicate definitions exposed by the FRMUNFOLDING rule. For example, `foo(x)`'s body is analyzed for the permissions required to frame `x.f >= 2` in `unfolding foo(x) in x.f >= 2`. If `foo(x)`'s body is imprecise, then SVL$_{\mathrm{RP}}$'s framing

definition would be undefined for this formula. Therefore, formula framing in GVL$_{\text{RP}}$, $\langle H, \rho, \Pi\rangle \mathrel{\widetilde{\vdash}_{\texttt{frmI}}} \phi$, is defined as in SVL$_{\text{RP}}$ except for the FRMUNFOLDING rule:

$$\frac{\begin{array}{c}\langle H,\rho,\Pi\rangle \vDash_I p(e_1,...,e_n) \qquad \langle H,\rho,\Pi\rangle \mathrel{\widetilde{\vdash}_{\texttt{frmI}}} e_1 \qquad ... \qquad \langle H,\rho,\Pi\rangle \mathrel{\widetilde{\vdash}_{\texttt{frmI}}} e_n \\ \langle H,\rho,\Pi'\rangle \mathrel{\widetilde{\vdash}_{\texttt{frmI}}} \phi \qquad \Pi' = \Pi \cup \lfloor \mathsf{body}_\mu(p)(e_1,...,e_n)\rfloor_{\mathsf{TotalFP}(\phi,H,\rho),H,\rho}\end{array}}{\langle H,\rho,\Pi\rangle \mathrel{\widetilde{\vdash}_{\texttt{frmI}}} \texttt{unfolding}\ p(e_1,...,e_n)\ \texttt{in}\ \phi}\ \widetilde{\text{FRMUNFOLDING}}$$

This rule differs from its SVL$_{\text{RP}}$ counterpart in computing Π', which aides in framing ϕ. In particular, the retrieval of accessibility predicates and predicate instances from $\mathsf{body}_\mu(p)(e_1, ..., e_n)$ now accounts for imprecision. The $\mathsf{TotalFP}(\cdot,\cdot,\cdot) : \text{FORMULA} \times \text{HEAP} \times \text{ENV} \to \text{PERMISSIONS}$ function returns the explicit and implicit iso-recursive permissions required by ϕ ($\{\langle \texttt{o}, \texttt{f}\rangle\}$ for `x.f >= 2` where `x` points to `o`). Then, a new footprint definition $\lfloor \widetilde{\phi} \rfloor_{\Pi,H,\rho}$ is used to either frame ϕ optimistically with this maximal permission set or precisely with calculated permissions from $\mathsf{body}_\mu(p)(e_1, ..., e_n)$. The result depends on whether $\mathsf{body}_\mu(p)(e_1, ..., e_n)$ is imprecise or precise, respectively (`foo`'s body is `?` so $\{\langle \texttt{o}, \texttt{f}\rangle\}$ is used):

$$\lfloor \theta \rfloor_{\Pi,H,\rho} = \lfloor \theta \rfloor_{H,\rho} \qquad\qquad \lfloor ?\ *\ \phi \rfloor_{\Pi,H,\rho} = \Pi$$

Now, a formula ϕ is called *self-framed* (we write $\widetilde{\vdash}_{\texttt{frm}}\ \phi$) if for all H, ρ, Π, $\langle H,\rho,\Pi\rangle \vDash_I \phi$ implies $\langle H,\rho,\Pi\rangle \mathrel{\widetilde{\vdash}_{\texttt{frmI}}} \phi$. We redefine the set of self-framed formulas: $\text{SFRMFORMULA} \overset{\text{def}}{=} \{ \phi \in \text{FORMULA} \mid \widetilde{\vdash}_{\texttt{frm}}\ \phi \}$, and we still write θ to denote self-framed formulas. As a result, `foo(x) * unfolding foo(x) in x.f >= 2` is self-framed when `foo`'s body is `?`.

2.2.3 Interpretation of Gradual Formulas

Gradual formulas are given meaning by the set of precise formulas that they represent. The interpretation of gradual formulas is used to define variants of formula evaluation, formula implication, and the WLP calculus that operate over gradual formulas and are *consistent liftings* [3, 19] of their SVL$_{\text{RP}}$ counterparts. Then, the static verification judgment in GVL$_{\text{RP}}$ is defined similarly to SVL$_{\text{RP}}$ using these lifted definitions. The set denoted by a gradual formula is obtained via a concretization function [25]:

Definition 2.2.1 (Concretization of Gradual Formulas). $\gamma : \widetilde{\text{FORMULA}} \rightharpoonup \mathcal{P}^{\text{FORMULA}}$ *is defined as:*

$$\gamma(\theta) = \{\ \theta\ \} \qquad\qquad \gamma(?\ *\ \phi) = \{\ \theta' \in \text{SATFORMULA} \mid \theta' \Rightarrow \phi\ \} \quad \textit{if}\ \phi \in \text{SATFORMULA}$$

$$\gamma(?\ *\ \phi) \quad \textit{undefined otherwise}$$

The concretization of a syntactically precise formula is the singleton set of this formula. The concretization of an imprecise formula is the (infinite) set of syntactically precise formulas that are 1) satisfiable and 2) imply the static part of the imprecise formula. For example, $\gamma(? * x \geq 0) = \{x = 2,\ y = x * x \geq 0,\ \ldots\}$. Notice, $x < 0 * x \geq 0 \notin \gamma(? * x \geq 0)$, because it is not satisfiable.

Novel compared to Bader et al. [3]'s work is the requirement that all syntactically precise formulas represented by gradual formulas must be *self-framed* (§2.2.2). This extra condition allows `?` to frame the static part of an imprecise formula, a requirement we motivated in §1.3.1.

Additionally, γ treats predicates opaquely by relying on iso-recursively defined satisfiability, self-framing, and implication. We make this design choice, because γ is an integral part of GVL$_{\text{RP}}$'s static verification system, which we want to be iso-recursive (§1.3.2). This choice has implications. For example, when both `p(x)` and `q(x)`'s bodies contain **`acc`**`(x.f)`, `p(x)` $*$ `q(x)` is equi-recursively unsatisfiable but iso-recursively satisfiable. Therefore, `p(x)` $*$ `q(x)` $\in \gamma(? *$ `q(x)`$)$. On the other hand, **`acc`**`(x.f)` $*$ **`acc`**`(x.f)` $\notin \gamma(? *$ **`acc`**`(x.f)`$)$, since **`acc`**`(x.f)` $*$ **`acc`**`(x.f)` is also iso-recursively unsatisfiable.

Definition 2.2.1 induces a natural definition of the *(im)precision* of gradual formulas:

Definition 2.2.2 (Precision of Gradual Formulas). *$\widetilde{\phi}_1$ is more precise (*i.e. *less imprecise) than $\widetilde{\phi}_2$, written $\widetilde{\phi}_1 \sqsubseteq \widetilde{\phi}_2$, if and only if $\gamma(\widetilde{\phi}_1) \subseteq \gamma(\widetilde{\phi}_2)$.*
Ex. $? *$ **`acc`**`(x.f)` $*$ `foo(y)` $\sqsubseteq$ $? *$ **`acc`**`(x.f)`.

Semantic Interpretation of Gradual Formulas. Since Definition 2.2.1 is interpreted iso-recursively, even if `foo`'s body is `?`, we can have `foo(x) * unfolding foo(x) in x.f>=2` $\in \gamma(? *$ `x.f >= 2`$)$. That is, γ in Definition 2.2.1 may give *syntactically* precise, but *semantically* imprecise formulas. We therefore need a semantic interpretation of gradual formulas that extends the concept of concretization to also cover imprecise predicate bodies. As a result, such a *semantic concretization* of gradual formulas would only give *semantically precise* formulas.

A difficulty with writing semantic concretization is that in order to fully interpret formulas, we require an additional function body_μ, which returns predicate bodies from the ambient program given a predicate instance, *e.g.* body_μ`(foo)(x) = ?`. Since body_μ may return imprecise formulas, we cannot use it to interpret formulas that we want to be semantically precise. Instead, we must rely on some new function $\mathsf{body}_\triangle : \text{PredName} \rightarrow \text{Expr}^* \rightarrow \text{Formula}$, which returns only precise formulas. As a result, we work with *local formulas* $\langle \phi, \mathsf{body}_\triangle \rangle \in \text{Formula} \times (\text{PredName} \rightarrow \text{Expr}^* \rightarrow \text{Formula})$ that explicitly drag along their body function.

Existing rules can easily be adjusted in order to deal with this new parameter, for example:

$$\frac{\mathsf{body}_\triangle(p)(e_1, ..., e_n) = \phi \qquad H, \rho \vdash e_1 \Downarrow v_1 \quad ... \quad H, \rho \vdash e_n \Downarrow v_n \quad \langle H, \rho, \pi \rangle \vDash_E \langle \phi, \mathsf{body}_\triangle \rangle}{\langle H, \rho, \pi \rangle \vDash_E \langle p\,(e_1, ..., e_n), \mathsf{body}_\triangle \rangle} \text{ EvPred}$$

The EvPred rule now uses $\mathsf{body}_\triangle$ to lookup predicate bodies, rather than using the designated body_μ. Notice the function $\mathsf{body}_\triangle$ is carried around for reference, simply making explicit what was previously assumed as constant and ambient in SVL$_{\text{RP}}$.

Now, we can give an interpretation to *gradual body functions* $\widetilde{\mathsf{body}}_\triangle$ by concretizing them into sets of $\mathsf{body}_\triangle$ functions that produce precise, self-framed formulas. Given a $\widetilde{\mathsf{body}}_\triangle$, Definition 2.2.3 returns a set of $\mathsf{body}_\triangle$ functions constructed from formulas that are in the γ (Def. 2.2.1) of each gradual formula in $\widetilde{\mathsf{body}}_\triangle$. For example, if $\mathsf{dom}(\widetilde{\mathsf{body}}_\triangle) = \{\texttt{foo}\}$, $\widetilde{\mathsf{body}}_\triangle(\texttt{foo})(\texttt{x}) = \texttt{?}$, and $\mathsf{body}_\triangle(\texttt{foo})(\texttt{x}) =$ **`acc`**`(x.f)`, then $\mathsf{body}_\triangle \in \gamma(\widetilde{\mathsf{body}}_\triangle)$. Additionally, each $\mathsf{body}_\triangle$ function must be well-formed with respect to self-framing, *i.e.* the body that $\mathsf{body}_\triangle$ returns for each predicate must be self-framed with respect to the $\mathsf{body}_\triangle$ function itself. For example, if $\mathsf{body}_\triangle(\texttt{q})(\texttt{x}) = \texttt{foo}(\texttt{x}) *$ **`unfolding`** `foo(x)` **`in`** `x.f` $>= 2$, then $\mathsf{body}_\triangle(\texttt{foo})(\texttt{x})$ must contain **`acc`**`(x.f)`.

Definition 2.2.3 (Concretization of Gradual Formulas (continued))**.** *Concretization of a gradual body function* $\gamma : (\textsc{PredName} \rightarrow \textsc{Expr}^* \rightarrow \widetilde{\textsc{Formula}}) \rightharpoonup \mathcal{P}^{\textsc{PredName}\rightarrow\textsc{Expr}^*\rightarrow\textsc{SfrmFormula}}$ *is defined as:*

$$\gamma(\widetilde{\mathsf{body}_\Delta}) = \{\ \mathsf{body}_\Delta = \lambda p_i \in \mathsf{dom}(\widetilde{\mathsf{body}_\Delta}).\ \lambda\overline{e} \in \textsc{Expr}^*.\ \theta_{p_i}[\overline{e}/\overline{\mathit{tmp}_i}] \mid \langle \theta_{p_1}, \theta_{p_2}, ...\rangle \in \gamma(\widetilde{\mathsf{body}_\Delta}(p_1)(\overline{\mathit{tmp}_1})) \times \gamma(\widetilde{\mathsf{body}_\Delta}(p_2)(\overline{\mathit{tmp}_2})) \times ..., \forall p_i \in \mathsf{dom}(\widetilde{\mathsf{body}_\Delta}).\ \vdash_{\mathit{frm}} \langle \mathsf{body}_\Delta(p_i)(\overline{\mathit{tmp}_i}), \mathsf{body}_\Delta\rangle\ \}$$

where $\mathsf{dom}(\widetilde{\mathsf{body}_\Delta}) = \{\ p_1, p_2, ...\ \} \subseteq \textsc{PredName}$.

Given this partial function, we can concretize a gradual formula and its gradual body function, yielding a set of semantically precise self-framed formulas:

$$\gamma(\langle\widetilde{\phi}, \widetilde{\mathsf{body}_\Delta}\rangle) = \{\ \langle\theta, \mathsf{body}_\Delta\rangle \mid \theta \in \gamma(\widetilde{\phi}), \mathsf{body}_\Delta \in \gamma(\widetilde{\mathsf{body}_\Delta}),\ \vdash_{\mathit{frm}} \langle\theta, \mathsf{body}_\Delta\rangle\ \}$$

As before, Definition 2.2.3 allows us to give a natural (semantic) definition for formula precision:

Definition 2.2.4 (Precision of Formulas (continued))**.** $\langle\widetilde{\phi}_1, \widetilde{\mathsf{body}^1_\Delta}\rangle$ *is more precise than* $\langle\widetilde{\phi}_2, \widetilde{\mathsf{body}^2_\Delta}\rangle$*, written* $\langle\widetilde{\phi}_1, \widetilde{\mathsf{body}^1_\Delta}\rangle \sqsubseteq \langle\widetilde{\phi}_2, \widetilde{\mathsf{body}^2_\Delta}\rangle$ *if and only if* $\gamma(\langle\widetilde{\phi}_1, \widetilde{\mathsf{body}^1_\Delta}\rangle) \subseteq \gamma(\langle\widetilde{\phi}_2, \widetilde{\mathsf{body}^2_\Delta}\rangle)$.

2.2.4 Lifting Predicates

We lift predicates on formulas in SVL_{RP} to handle gradual formulas in GVL_{RP} such that they are consistent liftings of corresponding SVL_{RP} predicates. Following AGT [19], the *consistent lifting* $\widetilde{P} \subseteq \widetilde{\textsc{Formula}} \times \widetilde{\textsc{Formula}}$ of predicate $P \subseteq \textsc{Formula} \times \textsc{Formula}$ is defined as:

$$\widetilde{P}(\widetilde{\phi}_1, \widetilde{\phi}_2) \overset{\mathsf{def}}{\iff} \exists\phi_1 \in \gamma(\widetilde{\phi}_1), \phi_2 \in \gamma(\widetilde{\phi}_2).\ P(\phi_1, \phi_2).$$

The existential in this definition expresses the optimistic nature of gradual semantics: we want a gradual predicate to be true if there exists any interpetation of ? that makes the static version of the predicate true.

Since we rely on an equi-recursive dynamic semantics for SVL_{RP} and GVL_{RP} and allow predicate definitions to be imprecise, we now give a semantic definition of gradual formula evaluation:

Definition 2.2.5 (Consistent Formula Evaluation)**.**

Let $\cdot \mathrel{\widetilde{\vDash}} \cdot \subseteq \textsc{Mem} \times (\widetilde{\textsc{Formula}} \times (\textsc{PredName} \rightarrow \textsc{Expr}^* \rightarrow \widetilde{\textsc{Formula}}))$ *be defined inductively as*

$$\frac{\langle H, \rho, \pi\rangle \vDash_E \langle\phi, \mathsf{body}_\Delta\rangle \qquad \langle H, \rho, \pi\rangle \vdash_{\mathtt{frmE}} \langle\phi, \mathsf{body}_\Delta\rangle}{\langle H, \rho, \pi\rangle \mathrel{\widetilde{\vDash}} \langle ?\ *\ \phi, \widetilde{\mathsf{body}_\Delta}\rangle}$$

$$\frac{\langle H, \rho, \pi\rangle \vDash_E \langle\theta, \mathsf{body}_\Delta\rangle \qquad \langle H, \rho, \pi\rangle \vdash_{\mathtt{frmE}} \langle\theta, \mathsf{body}_\Delta\rangle}{\langle H, \rho, \pi\rangle \mathrel{\widetilde{\vDash}} \langle\theta, \widetilde{\mathsf{body}_\Delta}\rangle}$$

where $\mathsf{body}_\Delta = \lambda p \in \mathsf{dom}(\widetilde{\mathsf{body}_\Delta}).\ \lambda\overline{e} \in \textsc{Expr}^*.\ \mathsf{static}(\widetilde{\mathsf{body}_\Delta}(p)(\overline{e}))$
and $\mathsf{static} : \widetilde{\textsc{Formula}} \rightarrow \textsc{Formula}$ *s.t.* $\mathsf{static}(\theta) = \theta$ *and* $\mathsf{static}(?\ *\ \phi) = \phi$.

Note that $\cdot \mathrel{\widetilde{\vDash}} \cdot$ is a consistent lifting of $\cdot \vDash_E \cdot$ (with γ from Def. 2.2.3). Our definition is conveniently implementable for equi-recursive dynamic checking: it simply evaluates the static

parts of predicates, and ensures that any heap accesses touch only owned locations. For example, if `foo`'s body is `?` and `x` points to `o`, then `foo(x) * unfolding foo(x) in x.f >= 2` evaluates to true when `o.f` is owned and `o.f` $\geq$ `2`. The static part of `?` is true, so `foo(x)` is ignored.

Additionally, gradual formula evaluation depends on an equi-recursive framing judgment for semantically precise formulas. The framing judgment $\langle H, \rho, \pi \rangle \vdash_{\texttt{frmE}} \phi$ is defined similarly (replacing Π with π and iso-recursive formula evaluation with equi-recursive formula evaluation) to its iso-recursive counterpart in SVL$_{\text{RP}}$, except for FRMPRED and FRMUNFOLDING. Equi-recursive variants of these rules are:

$$\frac{\forall i,\ \langle H, \rho, \pi \rangle \vdash_{\texttt{frmE}} e_i \qquad \langle H, \rho, \pi \rangle \vdash_{\texttt{frmE}} \mathsf{body}_{\mu}(p)(e_1, ..., e_n)}{\langle H, \rho, \pi \rangle \vdash_{\texttt{frmE}} p(e_1, ..., e_n)}$$

$$\frac{\langle H, \rho, \pi \rangle \vdash_{\texttt{frmE}} \phi}{\langle H, \rho, \pi \rangle \vdash_{\texttt{frmE}} \textbf{unfolding}\ p(e_1, ..., e_n)\ \textbf{in}\ \phi}$$

Then, a formula is said to be (equi-recursively) framed by permissions π if its complete unrolling only mentions fields in π. For example, `acyclic(l)`, where `acyclic`'s body is defined as in Figure 1.2, is framed by π if π contains all of list `l`'s heap locations. We can also easily adjust the equi-recursive framing judgment to pass around and use a body_{Δ} context, as described in §2.2.3.

In contrast to gradual formula evaluation (Def. 2.2.5), gradual formula implication is a consistent lifting of SVL$_{\text{RP}}$ formula implication with the syntactic interpretation of gradual formulas given in Definition 2.2.1. This is because SVL$_{\text{RP}}$ implication is defined iso-recursively, *i.e.* hides imprecision in predicates. We give the definition for gradual formula implication in Definition 2.2.6.

Definition 2.2.6 (Consistent Formula Implication)**.**
Let $\cdot \mathrel{\widetilde{\Rightarrow}} \cdot \subseteq \widetilde{\text{FORMULA}} \times \widetilde{\text{FORMULA}}$ *be defined inductively as*

$$\frac{\theta_1 \Rightarrow \mathsf{static}(\widetilde{\phi}_2)}{\theta_1 \mathrel{\widetilde{\Rightarrow}} \widetilde{\phi}_2}\ \widetilde{\text{IMPLSTATIC}} \qquad \frac{\theta \in \text{SATFORMULA} \qquad \theta \Rightarrow \phi_1 \qquad \theta \Rightarrow \mathsf{static}(\widetilde{\phi}_2)}{?\ *\ \phi_1 \mathrel{\widetilde{\Rightarrow}} \widetilde{\phi}_2}\ \widetilde{\text{IMPLGRAD}}$$

Here also, $\cdot \mathrel{\widetilde{\Rightarrow}} \cdot$ is a consistent lifting of $\cdot \Rightarrow \cdot$ (with γ from Def. 2.2.3). For example, `?` $\widetilde{\Rightarrow}$ `? *` **`acc`**`(x.f) * x.f >= 2` because **`acc`**`(x.f) * x.f >= 2` is satisfiable and implies the static part of both sides of the implication.

2.2.5 Lifting Functions

Functions that operate over formulas in SVL$_{\text{RP}}$ must also be lifted to handle gradual formulas in GVL$_{\text{RP}}$. The resulting GVL$_{\text{RP}}$ functions should approximate consistent liftings of corresponding SVL$_{\text{RP}}$ functions. Following AGT [19], given a partial function f : FORMULA $\rightharpoonup$ FORMULA, its *consistent lifting* $\widetilde{f} : \widetilde{\text{FORMULA}} \rightharpoonup \widetilde{\text{FORMULA}}$ is defined as:

$$\widetilde{f}(\widetilde{\phi}) = \alpha(\{\ f(\phi) \mid \phi \in \gamma(\widetilde{\phi})\ \}).$$

$$\widetilde{\mathsf{WLP}}(\texttt{if}\ (e)\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}, \widetilde{\phi}) = \alpha(\{\ \max_{\Rightarrow}\{\ \phi' \in \textsc{SatFormula} \mid \phi' \Rightarrow \texttt{if}\ e\ \texttt{then}\ \theta_1\ \texttt{else}\ \theta_2\ \ \wedge$$
$$\phi' \Rightarrow \mathsf{acc}(e)\ \wedge\ \vdash_{\texttt{frm}} \langle \phi', \mathsf{body}_\Delta{}' \rangle\ \} \mid \theta_1 \in \gamma(\widetilde{\mathsf{WLP}}(s_1, \widetilde{\phi})), \theta_2 \in \gamma(\widetilde{\mathsf{WLP}}(s_2, \widetilde{\phi})),$$
$$\mathsf{body}_\Delta{}' \in \gamma(\mathsf{body}_\mu),\ \vdash_{\texttt{frm}} \langle \theta_1, \mathsf{body}_\Delta{}' \rangle,\ \vdash_{\texttt{frm}} \langle \theta_2, \mathsf{body}_\Delta{}' \rangle\ \})$$
$$\widetilde{\mathsf{WLP}}(y\ \texttt{:=}\ z.m(\overline{x}), \widetilde{\phi}) = \alpha(\{\ \max_{\Rightarrow}\{\ \phi' \in \textsc{SatFormula} \mid y \notin \mathsf{FV}(\phi')\ \wedge\ \vdash_{\texttt{frm}} \langle \phi', \mathsf{body}_\Delta{}' \rangle\ \ \wedge$$
$$\exists \phi_f\ .\ \phi' \Rightarrow \texttt{(}z\ \neq\ \texttt{null)}\ *\ \theta_p[z/\texttt{this}, \overline{x/\mathsf{mparam}(m)}]\ *\ \phi_f\ \ \wedge$$
$$\phi_f\ *\ \theta_q[z/\texttt{this}, \overline{x/\textbf{old}(\mathsf{mparam}(m))}, y/\texttt{result}] \Rightarrow \theta\ \wedge\ \vdash_{\texttt{frm}} \langle \phi_f, \mathsf{body}_\Delta{}' \rangle\ \}$$
$$\mid \theta \in \gamma(\widetilde{\phi}),\ \theta_p \in \gamma(\mathsf{mpre}(m)),\ \theta_q \in \gamma(\mathsf{mpost}(m)),\ \mathsf{body}_\Delta{}' \in \gamma(\mathsf{body}_\mu),$$
$$\vdash_{\texttt{frm}} \langle \theta, \mathsf{body}_\Delta{}' \rangle,\ \vdash_{\texttt{frm}} \langle \theta_p, \mathsf{body}_\Delta{}' \rangle,\ \vdash_{\texttt{frm}} \langle \theta_q, \mathsf{body}_\Delta{}' \rangle\ \})$$
$$\widetilde{\mathsf{WLP}}(\texttt{while (}e\texttt{) inv}\ \widetilde{\phi_i}\ \{\ s\ \}, \widetilde{\phi}) = \alpha(\{\ \max_{\Rightarrow}\{\ \phi' \in \textsc{SatFormula} \mid \phi' \Rightarrow \mathsf{acc}(e)\ \wedge\ \vdash_{\texttt{frm}} \langle \phi', \mathsf{body}_\Delta{}' \rangle\ \ \wedge$$
$$\exists \phi_f\ .\ \phi' \Rightarrow \theta_i\ *\ \phi_f\ \ \wedge\ \ \overline{x_i} \notin \mathsf{FV}(\phi_f)\ \wedge\ \vdash_{\texttt{frm}} \langle \phi_f, \mathsf{body}_\Delta{}' \rangle\ \ \wedge$$
$$\phi_f\ *\ (\theta_i\ *\ \texttt{(}e\ \texttt{=}\ \texttt{false)})[\overline{x_i/y_i}] \Rightarrow \theta[\overline{x_i/y_i}]\ \}$$
$$\mid \theta \in \gamma(\widetilde{\phi}),\ \theta_i \in \gamma(\widetilde{\phi_i}),\ \mathsf{body}_\Delta{}' \in \gamma(\mathsf{body}_\mu),\ \vdash_{\texttt{frm}} \langle \theta, \mathsf{body}_\Delta{}' \rangle,\ \vdash_{\texttt{frm}} \langle \theta_i, \mathsf{body}_\Delta{}' \rangle\ \})$$
$$\texttt{where}\ \overline{y_i}\ \texttt{are vars modified by the loop body}\ s\ \texttt{and}\ \overline{x_i}\ \texttt{are fresh}$$
$$\widetilde{\mathsf{WLP}}(\texttt{fold}\ p\texttt{(}\overline{e}\texttt{)}, \widetilde{\phi}) = \alpha(\{\ \max_{\Rightarrow}\{\ \phi' \in \textsc{SatFormula} \mid \phi'\ *\ p\texttt{(}\overline{e}\texttt{)} \Rightarrow \theta\ \ \wedge\ \ \phi'\ *\ p\texttt{(}\overline{e}\texttt{)} \in \textsc{SatFormula}\ \wedge$$
$$\vdash_{\texttt{frm}} \langle \phi' * \mathsf{body}_\Delta{}'(p)(\overline{e}), \mathsf{body}_\Delta{}' \rangle\ \} * \mathsf{body}_\Delta{}'(p)(\overline{e}) \in \textsc{SatFormula}$$
$$\mid \theta \in \gamma(\widetilde{\phi}),\ \mathsf{body}_\Delta{}' \in \gamma(\mathsf{body}_\mu),\ \vdash_{\texttt{frm}} \langle \theta, \mathsf{body}_\Delta{}' \rangle\ \})$$

Figure 2.7: GVL_{RP}: Weakest liberal precondition calculus (select rules).

Notice, the definition of a *consistent function lifting* requires an abstraction function α, which given a set of formulas produces the most precise gradual formula representing this set. We define $\alpha : \mathcal{P}^{\text{Formula}} \rightharpoonup \widetilde{\text{Formula}}$ as $\alpha(\overline{\phi}) = \min_{\sqsubseteq}\ \{\ \widetilde{\phi} \in \widetilde{\text{Formula}} \mid \overline{\phi} \subseteq \gamma(\widetilde{\phi})\ \}$, *e.g.* $\alpha(\{\textbf{acc}(\texttt{x}_1.\texttt{f}), \textbf{acc}(\texttt{x}_1.\texttt{f}) * \textbf{acc}(\texttt{x}_2.\texttt{f})\}) = \texttt{?} * \textbf{acc}(\texttt{x}_1.\texttt{f})$. Then, α clearly creates a Galois connection with γ from Def. 2.2.1.

Fig. 2.7 shows select rules for $\widetilde{\mathsf{WLP}}$ (complete rules are in Appendix Fig. A.9), which approximate the consistent function lifting of WLP. Rules for method call, while loop, and if statements lift the corresponding WLP rules with respect to two (while loop and if statements) or three (method call statements) formula parameters instead of one formula parameter as in other rules. These corresponding WLP rules rely on extra (often implicit) formula parameters that may be imprecise in GVL_{RP}, and therefore, must be accounted for in the lifting. Similarly, WLP implicitly exposes predicate definitions in body_μ through self-framing (§2.2.2) and in fold and unfold rules. In GVL_{RP}, predicate definitions may be imprecise, so non-sequence statement WLP rules are lifted with respect to body_μ.

2.2.6 Lifting the Verification Judgment

We define static verification in $\mathrm{GVL_{RP}}$ using lifted formula implication ($\widetilde{\Rrightarrow}$, §2.2.4) and lifted WLP ($\widetilde{\mathsf{WLP}}$, §2.2.5):

Definition 2.2.7 (Valid Method)**.** *A method with contract* `requires` $\widetilde{\phi}_p$ `ensures` $\widetilde{\phi}_q$*, parameters* $\overline{x}$*, and body* s *is considered valid if* $\widetilde{\phi}_p \,\widetilde{\Rrightarrow}\, \widetilde{\mathsf{WLP}}(s, \widetilde{\phi}_q)[\overline{x/\mathtt{old}(x)}]$ *holds.*

Definition 2.2.8 (Valid Program)**.** *A program with entry point statement* s *is considered valid if* $\mathtt{true} \,\widetilde{\Rrightarrow}\, \widetilde{\mathsf{WLP}}(s, \mathtt{true})$ *holds,* $\widetilde{\phi}_i \wedge \mathsf{acc}(e) \wedge \mathtt{(e = true)} \,\widetilde{\Rrightarrow}\, \widetilde{\mathsf{WLP}}(r, \widetilde{\phi}_i \wedge \mathsf{acc}(e))$ *holds for all loops with condition* e*, body* r*, and invariant* $\widetilde{\phi}_i$*, and all methods are valid.*

2.3 $\mathrm{GVL_{RP}}$: Dynamic Semantics

A valid $\mathrm{GVL_{RP}}$ program will plausibly remain valid during each step of execution. To ensure that it does, the dynamic semantics of $\mathrm{SVL_{RP}}$ are extended with run-time checks and considerations for imprecise specifications.

2.3.1 Footprint Splitting

To split dynamic footprints at method calls and loop entries in $\mathrm{GVL_{RP}}$'s small-step semantics, we use $\lfloor \widetilde{\phi} \rfloor_{\pi,H,\rho}$:

$$\lfloor \theta \rfloor_{\pi,H,\rho} = \langle\langle\, \lfloor \theta \rfloor_{H,\rho} \,\rangle\rangle_{\pi,H} \qquad\qquad \lfloor ?\, *\, \phi \rfloor_{\pi,H,\rho} = \pi$$

This definition relies on $\langle\langle\, \Pi \,\rangle\rangle_{\pi,H} : \textsc{Permissions} \times \textsc{DynFprint} \times \textsc{Heap} \rightharpoonup \textsc{DynFprint}$, which returns the given dynamic footprint when any predicate bodies analyzed by the function are imprecise. Otherwise, the function returns the dynamic footprint generated from unrolling predicates in Π[2]:

$$\begin{aligned}
&\langle\langle\, \Pi \,\rangle\rangle_{\pi,H} = \{\langle o, f\rangle \mid \langle o, f\rangle \in \Pi\} \cup \pi' \\
&\mathtt{where}\ \pi' \ = \begin{cases} \pi & \mathtt{if}\ \exists \langle p, v_1, ..., v_n\rangle \in \Pi. \exists \phi \in \textsc{Formula}.\mathsf{body}_\mu(p)(v_1, ..., v_n) = \ ?\, *\, \phi \\ \langle\langle\, \Pi' \,\rangle\rangle_{\pi,H} & \mathtt{otherwise} \\ \mathtt{for}\ \Pi' = \cup_{\langle p, v_1, ..., v_n\rangle \in \Pi} \lfloor \mathsf{body}_\mu(p)(v_1, ..., v_n) \rfloor_{H,[]} & \end{cases}
\end{aligned}$$

Therefore, $\lfloor \widetilde{\phi} \rfloor_{\pi,H,\rho}$ returns the given dynamic footprint π when $\widetilde{\phi}$ is imprecise or contains nested imprecision, and it returns a more precise dynamic footprint computed when $\widetilde{\phi}$ is semantically precise. Example, if `acyclic`'s body is `?`, then $\lfloor \mathtt{acyclic(l)} \rfloor_{\pi,H,\rho}$ will return π. It will return all of list `l`'s heap locations when `acyclic` is defined as in Fig. 1.2.

[2]Note that $\langle\langle\, \Pi \,\rangle\rangle_{\pi,H}$ is a partial function, as it may not be well-defined if a predicate instance held in Π has an infinite completely unrolling and no nested imprecise predicates.

$$\frac{\langle H, \rho, \pi \rangle \mathrel{\widetilde{\vDash}} \langle ?\ *\ \phi, \mathsf{body}_\mu \rangle}{\langle H, \langle \rho, \pi, \mathtt{assert}\ \phi\mathtt{;}\ s \rangle \cdot S \rangle \mathrel{\widetilde{\longrightarrow}} \langle H, \langle \rho, \pi, s \rangle \cdot S \rangle} \text{ SSASSERT}$$

$$\frac{\begin{array}{c} \mathsf{method}(m) = T_r\ m\ (\overline{T\ x'})\ \mathtt{requires}\ \widetilde{\phi}_p\ \mathtt{ensures}\ \widetilde{\phi}_q\ \{\ r\ \} \\ H, \rho \vdash z \Downarrow o \qquad \overline{H, \rho \vdash x \Downarrow v} \qquad \rho' = [\mathtt{this} \mapsto o, \overline{x' \mapsto v}, \overline{\mathbf{old}(x') \mapsto v}] \\ \pi' = \lfloor \widetilde{\phi}_p \rfloor_{\pi, H, \rho'} \qquad \pi' \subseteq \pi \qquad \langle H, \rho', \pi' \rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}_p, \mathsf{body}_\mu \rangle \end{array}}{\langle H, \langle \rho, \pi, y \mathrel{\mathtt{:=}} z.m(\overline{x})\mathtt{;}\ s \rangle \cdot S \rangle \mathrel{\widetilde{\longrightarrow}} \langle H, \langle \rho', \pi', r\mathtt{;}\ \mathtt{skip} \rangle \cdot \langle \rho, \pi \backslash \pi', y \mathrel{\mathtt{:=}} z.m(\overline{x})\mathtt{;}\ s \rangle \cdot S \rangle} \text{ SSCALL}$$

$$\frac{\mathsf{mpost}(m) = \widetilde{\phi}_q \qquad \langle H, \rho', \pi' \rangle \mathrel{\widetilde{\vDash}} \langle \widetilde{\phi}_q, \mathsf{body}_\mu \rangle \qquad \rho'' = \rho[y \mapsto \rho'(\mathtt{result})]}{\langle H, \langle \rho', \pi', \mathtt{skip} \rangle \cdot \langle \rho, \pi, y \mathrel{\mathtt{:=}} z.m(\overline{x})\mathtt{;}\ s \rangle \cdot S \rangle \mathrel{\widetilde{\longrightarrow}} \langle H, \langle \rho'', \pi \cup \pi', s \rangle \cdot S \rangle} \text{ SSCALLFINISH}$$

Figure 2.8: GVL_{RP}: Small-step semantics adjusted from Fig. 2.6 (select rules)

2.3.2 Small-Step Semantics

We give an augmented version of SVL_{RP}'s small-step semantics ($\cdot \mathrel{\widetilde{\longrightarrow}} \cdot \ \subseteq \text{STATE} \times (\text{STATE} \cup \{\mathsf{error}\})$) for GVL_{RP}. We make considerations for imprecision and for run-time verification. Representative rules are given in Fig. 2.8 (complete rules are in Appendix Fig. A.10).

Imprecision in Specifications. Method preconditions, postconditions, and loop invariants are now checked with gradual formula evaluation (SSCALL, SSCALLFINISH). Asserted formulas must also be checked with gradual formula evaluation due to potentially hidden imprecision (SSASSERT). Additionally, we must ensure that introducing imprecision will not introduce a run-time error caused by lack of accessibility (dynamic gradual guarantee, Prop. 2.4.6). Therefore, if a method precondition in SSCALL (or loop invariant) is imprecise or contains nested imprecision, then all owned heap locations are forwarded from the call site to the callee (or loop body) for execution. Otherwise, the call site's owned heap locations can be precisely transferred to the callee (or loop body) as in SVL_{RP}. Heap locations held after the callee's (or loop body's) execution are returned as usual to the call site.

Runtime Verification. Even for valid GVL_{RP} programs, when specifications are imprecise the formula evaluation premises in GVL_{RP}'s small-step semantics are not guaranteed to hold. Therefore, these premises are turned into run-time checks. If an assertion, accessibility predicate, method precondition, method postcondition, or loop invariant does not hold in a program state where it should, then program execution steps into a dedicated **error** state (extra rules illustrating this can be found in Appendix Fig. A.10).

2.4 Properties of GVL_{RP}

GVL_{RP} is a sound gradually-verified language that conservatively extends SVL_{RP} and adheres to gradual guarantees. GVL_{RP} is a *conservative extension* of SVL_{RP}—meaning that GVL_{RP} and

SVL_{RP} coincide on fully precise programs—by construction following the Abstracting Gradual Typing methodology [3, 19].

Soundness. Soundness for GVL_{RP} is conceptually similar to soundness for SVL_{RP} except that a GVL_{RP} program may step to a dedicated error state when run-time verification fails. We establish soundness via progress and preservation.

Definition 2.4.1 (Valid State, Final State)**.** *We call the state* $\langle H, \langle \rho_n, \pi_n, s_n\rangle \cdot \ldots \cdot \langle \rho_1, \pi_1, s_1\rangle \cdot \mathsf{nil}\rangle \in$ STATE valid *if*
$s_n = s$; `skip` *or* `skip` *for some* $s \in$ STMT*,*
$s_i = s'_i$; `skip` *for some* $s'_i \in$ STMT $\forall\,.\,1 \le i < n$*,*
and $s_i = s^1_i$; s^2_i *for some* $s^1_i, s^2_i \in$ STMT *where* s^1_i *is a method call or while loop statement* $\forall\,.\,1 \le i < n$*.*
A state ψ *is* final *if* $\psi = \langle H, \langle \rho, \pi, \texttt{skip}\rangle \cdot \mathsf{nil}\rangle$ *for some* H, ρ, π*.*

Proposition 2.4.2 (GVL_{RP} Progress)**.** *If* ψ *is a valid non-final state then* $\psi \widetilde{\longrightarrow} \psi'$ *for some* ψ' *or* $\psi \widetilde{\longrightarrow}$ **error***.*

Proposition 2.4.3 (GVL_{RP} Preservation)**.** *If* ψ *is a valid state and* $\psi \widetilde{\longrightarrow} \psi'$ *for some* ψ' *then* ψ' *is a valid state.*

Gradual Guarantees. GVL_{RP} satisfies both the static and the dynamic gradual guarantees, originally formulated for gradual type systems [43], and first adapted to gradual verification by Bader et al. [3]. These properties ensure in GVL_{RP} that decreasing the precision of specifications never breaks the verifiability and reducibility of a program, *i.e.* losing precision is harmless.

These properties rely on a notion of precision for programs. We say a program p_1 is more precise than program p_2 ($p_1 \sqsubseteq p_2$) if 1) p_1 and p_2 are equivalent except in terms of contracts, loop invariants, and/or predicate definitions, and 2) p_1's contracts, loop invariants, and predicate definitions are more precise than p_2's corresponding contracts, loop invariants, and predicate definitions. A contract `requires` $\widetilde{\phi}^1_p$ `ensures` $\widetilde{\phi}^1_q$ is more precise than contract `requires` $\widetilde{\phi}^2_p$ `ensures` $\widetilde{\phi}^2_q$ if $\widetilde{\phi}^1_p \sqsubseteq \widetilde{\phi}^2_p$ and $\widetilde{\phi}^1_q \sqsubseteq \widetilde{\phi}^2_q$. Similarly, a loop invariant (predicate definition) $\widetilde{\phi}^1_i$ is more precise than loop invariant (predicate definition) $\widetilde{\phi}^2_i$ if $\widetilde{\phi}^1_i \sqsubseteq \widetilde{\phi}^2_i$.

Using this notion of program precision, the static gradual guarantee can now be stated as follows:

Proposition 2.4.4 (GVL_{RP} Static gradual guarantee)**.**
Let $p_1, p_2 \in$ PROGRAM *such that* $p_1 \sqsubseteq p_2$*. If* p_1 *is valid then* p_2 *is valid.*

In general, the static gradual guarantee ensures that reducing the precision of specifications never breaks static verification (*i.e.* makes a valid program invalid).

For the dynamic gradual guarantee, the fact that footprint tracking and splitting is influenced by increasing imprecision (*i.e.* increasing imprecision results in larger parts of footprints being passed up the stack) means that we must define an asymmetric *state precision* relation $\lesssim$:

Definition 2.4.5 (State Precision)**.** *Let* $\psi_1, \psi_2 \in \textsc{State}$*. Then* ψ_1 *is more precise than* ψ_2*, written* $\psi_1 \lesssim \psi_2$*, if and only if all of the following applies:*
a) ψ_1 *and* ψ_2 *have stacks of size* n *and identical heaps.*
b) ψ_1 *and* ψ_2 *have stacks of variable environments that are identical.*
c) *Let* $s^1_{1..n}$ *and* $s^2_{1..n}$ *be the stack of statements of* ψ_1 *and* ψ_2*, respectively. Then for* $1 \leq i \leq n$*,* $s^1_i \sqsubseteq s^2_i$*:*

$s \sqsubseteq s'$ *if and only if* s *is a fold or unfold statement and* s' *is a skip statement or equal to* s*,*
$s = \texttt{while}\ (e)\ \texttt{inv}\ \widetilde{\phi}_i\ \{\ r\ \}$ *and* $s' = \texttt{while}\ (e)\ \texttt{inv}\ \widetilde{\phi}'_i\ \{\ r\ \}$ *where* $\widetilde{\phi}_i \sqsubseteq \widetilde{\phi}'_i$*,*
$s = s_{c_1}; s_{c_2}$ *and* $s' = s'_{c_1}; s'_{c_2}$ *where* $s_{c_1} \sqsubseteq s'_{c_1}$ *and* $s_{c_2} \sqsubseteq s'_{c_2}$*, or* $s = s'$*.*

d) *Let* $\pi^1_{1..n}$ *and* $\pi^2_{1..n}$ *be the stack of footprints of* ψ_1 *and* ψ_2*, respectively. Then the following holds for* $1 \leq m \leq n$*:*

$$\bigcup_{i=m}^{n} \pi^1_i \subseteq \bigcup_{i=m}^{n} \pi^2_i$$

Additionally, as long as it does not break the static gradual guarantee, we allow increased imprecision through dropped fold and unfold statements from one program to the next. This is reflected in condition c) in Definition 2.4.5 and an adjusted program precision definition $\sqsubseteq_d$. That is, a program p_1 is more precise than a program p_2 if 1) the programs are equivalent except for in terms of contracts, loop invariants, and/or predicate definitions and fold and unfold statements in p_1 may be replaced with `skip` statements in p_2, and 2) p_1's contracts, loop invariants, and predicate definitions are more precise than p_2's corresponding contracts, loop invariants, and predicate definitions. Now, the *dynamic gradual guarantee* can be given:

Proposition 2.4.6 (GVL$_{\text{RP}}$ Dynamic gradual guarantee)**.**
Let $p_1, p_2 \in \textsc{Program}$ *such that* $p_1 \sqsubseteq_d p_2$*, and* $\psi_1, \psi_2 \in \textsc{State}$ *such that* $\psi_1 \lesssim \psi_2$*.*
If $\psi_1 \widetilde{\longrightarrow}_{p_1} \psi'_1$*, then* $\psi_2 \widetilde{\longrightarrow}_{p_2} \psi'_2$*, with* $\psi'_1 \lesssim \psi'_2$*.*

Since GVL$_{\text{RP}}$ adheres to the dynamic gradual guarantee, reducing the precision of specifications and/or dropping fold and unfold statements does not affect the program's observable behavior.

Chapter 3

Gradual C0: The First Gradual Verifier

This chapter presents the design and implementation of Gradual C0[1]—the first gradual verifier for imperative programs manipulating recursive heap data structures. Gradual C0 targets C0, a safe subset of C designed for education, with appropriate support (and pedagogical material) for dynamic verification. Technically, Gradual C0 is built on top of the Viper static verification infrastructure [33], which facilitates the development of program verifiers supporting IDF and recursive abstract predicates. Gradual C0's back-end leverages this infrastructure to simplify the implementation of gradual verifiers for other programming languages, and Gradual C0's front-end demonstrates how this is done for C0. Furthermore, Gradual C0 relies on symbolic execution for static reasoning rather than on the *weakest liberal precondition* approach theorized in Chpt. 2. We switched to symbolic execution from weakest liberal preconditions, because it is the ideal reasoning technique for tools based on separation logic or IDF. Indeed, Viper [33], VeriFast [22], JStar [13], and SmallFoot [4] all support these permission logics with symbolic execution, not weakest liberal preconditions. So, we hope the work in this chapter serves as a guide on how to build gradual verifiers that use symbolic execution for reasoning. Finally, Gradual C0 minimizes the insertion of dynamic checks using statically available information and optimizes the checks' overhead at run time—making advancements over our run-time system from Chpt. 2, §2.3, which checks everything at run time despite the precision of specifications.

Overall, in this chapter, we address new technical challenges in gradual verification related to symbolic execution, extensibility to multiple programming languages, and minimizing run-time checks and their overhead:

- Gradual C0's symbolic execution algorithm is responsible for statically verifying programs with imprecise specifications and producing minimized run-time checks. In particular, Gradual C0 tracks the branch conditions created by program statements and specifications to produce run-time checks for corresponding execution paths. At run time, branch conditions are assigned to variables at the branch point that introduced them, which are then used to coordinate the successive checks as required. Further, Gradual C0 creates run-time checks by translating symbolic expressions into specifications—reversing the symbolic execution process.
- The run-time checks produced by Gradual C0 contain branch conditions, simple logical ex-

[1]Gradual C0 is hosted on Github: https://github.com/gradual-verification/gvc0.

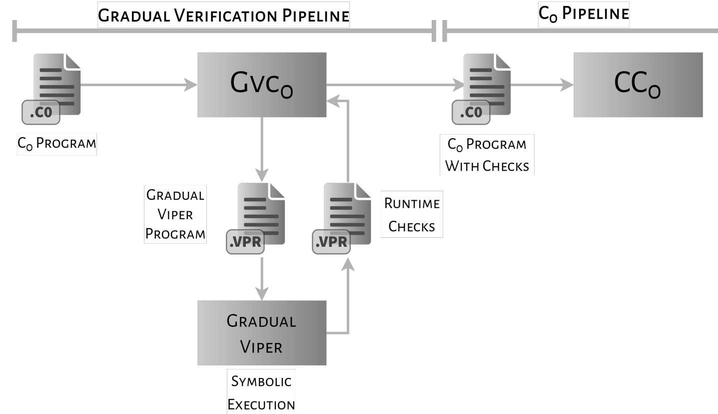


Figure 3.1: System design of Gradual C0

pressions, accessibility predicates, separating conjunctions, and predicates. Each of these constructs is specially translated into source code that can be executed at run time for dynamic verification. Logical expressions are turned into assertions. Accessibility predicates and separating conjunctions are checked by tracking and updating a set of owned heap locations. Finally, predicates are translated into recursive boolean functions. By encoding runtime checks into C0 source code, we avoid complexities from augmenting the C0 compiler to support dynamic verification. We also design these encodings to be performance friendly, *e.g.* owned heap locations are tracked in a dynamic hash table.

Note, Gradual C0's design, which newly uses symbolic execution for reasoning and minimizes run-time checks with statically available information, has been formalized and proven sound by Zimmerman et al. [50].

3.1 Gradual C0's Overall Design

Gradual C0 is a working gradual verifier for the C0 programming language [2] that is built as extension of the Viper static verifier [33]. Our goals for the design and implementation of Gradual C0 are to:

- be easily extensible to other programming languages beyond C0,
- minimize run-time overhead from verification as much as possible without introducing highly complex algorithms, and
- use symbolic execution for static reasoning.

Consequently, we settled on the design illustrated in Fig. 3.1. Gradual C0 is structured in two major sub-systems: 1) the gradual verification pipeline and 2) the C0 pipeline. Within the gradual verification pipeline, a C0 program is first translated AST-to-AST into a Gradual Viper program by Gradual C0's frontend module, GVC0. The GVC0 module implements a simple parser, abstract syntax, and type checker for C0 programs to facilitate the translation. The Gradual Viper module is the backend of Gradual C0 and implements its own parser, abstract syntax, and type checker for its own imperative language called the Gradual Viper language.

This language is the Viper language plus imprecise formulas and allows Gradual Viper to support multiple frontend languages, not just C0. We chose to build a C0 frontend first because C0 is a pedagogical version of C designed with dynamic verification in mind, and we plan to use it in the classroom.

Once the C0 program is translated into a Gradual Viper program, it is optimistically statically verified by the Gradual Viper module. Gradual Viper extends Viper's symbolic execution based verifier to support imprecise formulas and resulting holes in static reasoning as inspired by Chpt. 2 and gradual typing [20, 42, 43]. Consequently, by construction Gradual Viper supports full static verification of programs when specifications are complete. Differing from the work in Chpt. 2, Gradual Viper also extends the symbolic execution algorithm to create a description of needed run-time checks in support of static holes. The run-time checks are minimized with statically available information during reasoning. Finally, GVC0 takes these run-time checks and encodes them in the original C0 program to produce a sound, gradually-verified program. The C0 pipeline takes this C0 program and feeds it to the C0 compiler, CC0, which is used to execute the program. Note, the encoding of checks into C0 source code optimizes for run-time performance and simplifies extending C0 with dynamic verification in our domain.

The rest of this chapter describes the implementation of Gradual Viper and GVC0's designs in more detail and illustrates the concepts via example. We also highlight design and implementation choices influenced by our goals. §3.2 discusses how C0 programs are translated to Gradual Viper programs, along with modifications made to both C0 and Viper for gradual verification. Then, §3.3 and §3.4 detail Gradual Viper's symbolic execution approach and how it produces minimized run-time checks. Finally, §3.5 focuses on how GVC0 turns run-time checks from Gradual Viper into C0 code for dynamic verification.

3.2 Translating C0 Source Code to Gradual Viper Source Code for Verification

The C0 language, with its minimal set of language features and its existing support for specifications, serves well as the target language for our implementation. As its name suggests, C0 borrows heavily from C, but its feature set is reduced to better suit its intended purpose as a tool in computer science education [2]. It is a memory-safe subset of C that forbids casts, pointer arithmetic, and pointers to stack-allocated memory. All pointers are created with heap allocation, and de-allocation is handled by a garbage collector.

The abstract syntax for C0 programs supported by Gradual C0 is given in Fig. 3.3, *i.e.* GVC0's abstract syntax. GVC0 programs are made of struct and method declarations that largely follow C syntax. What differs from both C and C0 is GVC0's specification language. Methods may specify constraints on their input and output values as side-effect-free gradual formulas $\widetilde{\phi}$, usually in `//@requires` or `//@ensures` clauses in the method header. Loops and abstract predicates contain invariants and bodies respectively that are made of gradual formulas. Such formulas $\widetilde{\phi}$ are imprecise formulas `? &&` ϕ[2] or complete boolean formulas ϕ (Note, in this case, ϕ must be self-framed as defined in IDF). A formula ϕ joins boolean values, boolean operators,

[2]We switch from $*$ to denote the separating conjuction to `&&` since Viper uses this instead.

$x \in VAR$	(variables)	$S \in STRUCTNAME$	(struct names)
$v \in VAL$	(values)	$f \in FIELDNAME$	(field names)
$e \in EXPR$	(expressions)	$p \in PREDNAME$	(predicate names)
$s \in STMT$	(statements)	$m \in METHODNAME$	(method names)
$op \in$ +, -, /, *, ==, !=, <=, >=, <, >			(operators)

Figure 3.2: Shared abstract syntax definitions

```
P          ::= ‾struct‾ ‾predicate‾ ‾method‾
struct     ::= struct S { ‾T f‾ }
predicate  ::= //@predicate p(‾T x‾) = φ̃
method     ::= T̃ m(‾T x‾) contract { s }
contract   ::= //@requires φ̃; //@ensures φ̃;
T          ::= struct S | int | bool | char
             | T*
T̃          ::= void | T
s          ::= s; s | T x | T x = e | x = e
             | l = e | e | assert(e)
             | //@assert φ | //@fold p(ẽ)
             | //@unfold p(ẽ)
             | if (e) { s } else { s }
             | while (e) //@loop_invariant φ̃ { s }
             | for (s; e; s) //@loop_invariant φ̃ { s }
e          ::= v | x | op(ē) | e->f | *e | m(ē)
             | alloc(T) | e ? e : e
ẽ          ::= v | x | op(ẽ) | ẽ->f | *ẽ
l          ::= x->f | *x | l->f | *l
x          ::= \result | id
v          ::= n | c | NULL | true | false
φ̃          ::= ? && φ | θ
θ          ::= self-framed φ
φ          ::= ẽ | p(ẽ) | acc(l) | φ && φ
             | ẽ ? φ : φ
```

Figure 3.3: GVC0 abstract syntax

```
P          ::= ‾field‾ ‾predicate‾ ‾method‾
field      ::= field f : T
predicate  ::= predicate p(‾x : T‾) { φ̃ }
method     ::= method m(‾x : T‾) returns (‾y : T‾)
               contract { s }
contract   ::= requires φ̃ ensures φ̃
T          ::= Int | Bool | Ref
s          ::= s; s | var x : T | x := e | x.f := e
             | x := new(‾f‾) | ‾x‾ := m(ē) | assert φ
             | fold acc(p(ē)) | unfold acc(p(ē))
             | if (e) { s } else { s }
             | while (e) invariant φ̃ { s }
e          ::= v | x | op(ē) | e.f
x          ::= result | id
v          ::= n | null | true | false
φ̃          ::= ? && φ | θ
θ          ::= self-framed φ
φ          ::= e | acc(p(ē)) | acc(e.f) | φ && φ
             | e ? φ : φ
```

Figure 3.4: Gradual Viper abstract syntax

■	Representation differs slightly in GVC0 vs. Gradual Viper	■	Functionality in GVC0 that requires non-trivial translation to Gradual Viper

Figure 3.5: Abstract syntax comparison for GVC0 and Gradual Viper

predicate instances, accessibility predicates, and conditionals via the separating conjunction `&&`. GVC0 programs also contain `//@fold p(`$\bar{\bar{e}}$`)` and `//@unfold p(`$\bar{\bar{e}}$`)` statements for predicates and `//@assert` ϕ statements for convenience.

To support the gradual verification of many different imperative programming languages, Gradual Viper verifies programs written in its own custom imperative language, which is designed to ease the translation from other imperative languages into it. The Gradual Viper language's abstract syntax is given in Fig. 3.4. The GVC0 and Gradual Viper languages are roughly 1-to-1, including their specification languages, so translation is mostly straightforward, but there are some differences as highlighted in yellow (trivial) and blue (non-trivial) in Fig. 3.5. For example, `for` loops in GVC0 are rewritten as `while` loops in Gradual Viper, and `alloc(struct T)` expressions are translated to `new` statements containing `struct T`'s fields. Additionally, GVC0 allows method calls, `alloc`s, and ternaries in arbitrary expressions, while Gradual Viper only allows such constructs in corresponding program statements[3]. Therefore, GVC0 uses fresh temporary variables to version expressions containing the aformentioned constructs into program statements in Gradual Viper. The temporary variables are then used in the original expression in place of the corresponding method call, `alloc`, or ternary. Nested field assignments, such as `x->y->z = a`, are similarly expanded into multiple program statements using temporary variables. Value type pointers in GVC0 are rewritten as pointers to single-value structs that can be easily translated into Gradual Viper syntax. Finally, `assert(`e`)` statements are essentially ignored in Gradual Viper; e is translated into Gradual Viper syntax to verify its heap accesses, but e is not asserted. Instead, the assert is always kept in the original C0 program and is checked exclusively at run time. Fig. 3.8 provides a simple example program written in both the GVC0 language (Fig. 3.6) and Gradual Viper language (Fig. 3.7) for reference.

Note that GVC0 does not support array and string values since gradually verifying any interesting properties about such constructs requires non-trivial extensions to current gradual verification theory. Similarly, the Gradual Viper language, in contrast to the Viper language, does not support the aforementioned constructs and fractional permissions.

3.3 Gradual Viper: Symbolic Execution for Gradual Verification

In this section, we describe the design and implementation of Gradual Viper's symbolic execution based algorithm supporting the static verification of imprecise formulas. Our static reasoning algorithm also soundly reduces the number of run-time checks required during dynamic verification with statically available information. That is, during a single execution of Gradual Viper a program is statically verified and a set of minimized run-time checks is produced for program points where the algorithm is optimistic during verification due to imprecision.

Before formalizing Gradual Viper's implementation in §3.4, we first demonstrate at a high-level with examples how symbolic execution is used to perform optimistic static verification of programs containing recursive heap data structures and how minimized run-time checks are produced during this process. We also point out novel technical challenges faced and solutions

[3]Note, ternaries correspond to if statements

```
1  struct Account { int balance; };
2  typedef struct Account Account;
3
4  /*@ predicate geqTo(Account* a1, Account* a2)
5    = ? && a1->balance >= a2->balance  &&
6      a2->balance >= 0; @*/
7  /*@ predicate positive(Account* a) =
8    acc(a->balance) && a->balance >= 0; @*/
9
10 Account* withdraw(Account* a1, Account* a2)
11   //@ requires geqTo(a1,a2) ;
12   //@ ensures ? && positive(a2) &&
13                positive(\result) ;
14 {
15   //@ unfold geqTo(a1,a2);
16   if (a1 == NULL || a2 == NULL) {
17     return a1;
18   } else {
19     int newB = a1->balance - a2->balance;
20     a1->balance = newB;
21     //@ fold positive(a1);
22     //@ fold positive(a2);
23     return a1;
24   }
25 }
26
```

Figure 3.6: A simple bank withdraw example written in the Gradual C0 language

```
1  field balance: Int
2
3
4  predicate geqTo(a1: Ref, a2: Ref)
5    { ? && a1.balance >= a2.balance &&
6    a2.balance >= 0 }
7  predicate positive(a: Ref)
8   { acc(a.balance) && a.balance >= 0 }
9
10 method withdraw(a1: Ref, a2: Ref)
11 returns (res: Ref)
12   requires acc(geqTo(a1,a2))
13   ensures ? && acc(positive(a2)) &&
14              acc(positive(res))
15 {
16   unfold acc(geqTo(a1,a2))
17   if (a1 == null || a2 == null) {
18     res := a1
19   } else {
20     var newB: Int = a1.balance-a2.balance
21     a1.balance := newB
22     fold acc(positive(a1))
23     fold acc(positive(a2))
24     res := a1
25   }
26 }
```

Figure 3.7: A simple bank withdraw program written in the Gradual Viper language

☐ Program code ▪ Static specification ▪ Imprecise specification

Figure 3.8: A gradually verified, bank withdraw program that is contrived to illustrate how Gradual Viper works

developed thanks to relying on symbolic execution both for static verification and minimizing run-time checks.

3.3.1 Optimistic static verification in Gradual Viper by example

The simple program given in Fig. 3.7 implements a `withdraw` function (method), which subtracts the balance in one bank account (the subtrahend) from the balance in another account (the minuend) returning the result.[4] Any client program of `withdraw` must ensure the subtrahend's balance is less than or equal to the minuend's balance and that both balances are positive as specified by `withdraw`'s precondition (line 12). Then, `withdraw` will return an account with a positive balance as specified by `withdraw`'s postcondition (lines 13-14). Additionally, `withdraw`'s postcondition ensures the subtrahend's balance remains positive as well. Note, the

[4]Note, we refer to the version of the withdraw program written in the Gradual Viper language rather than its Gradual C0 counterpart in Fig. 3.6 as we will be discussing how Gradual Viper works in this section.

Lns	Impr-ecise	Opt. Heap	Heap	Var Store	Path Condition	Run-time Checks
15-16	No	∅	geqTo(t1,t2)	a1→t1 ; a2→t2 ; res→t3	∅	∅
16-17	Yes	acc(t1,balance,p1) ; acc(t2,balance,p2)	∅	a1→t1 ; a2→t2 ; res→t3	t1 != null ; t2 != null ; p1 >= p2 ; p2 >= 0	∅
17-18	-	-	-	-	-	-
18-19	-	-	-	-	-	-
19-20	Yes	acc(t1,balance,p1) ; acc(t2,balance,p2)	∅	a1→t1 ; a2→t2 ; res→t3	t1 != null ; t2 != null ; p1 >= p2 ; p2 >= 0	∅
20-21	Yes	acc(t1,balance,p1) ; acc(t2,balance,p2)	∅	a1→t1 ; a2→t2 ; res→t3 ; newB→t4	t1 != null ; t2 != null ; p1 >= p2 ; p2 >= 0 ; t4 = p1 - p2	∅
21-22	Yes	∅	acc(t1,balance,p3)	a1→t1 ; a2→t2 ; res→t3 ; newB→t4	t1 != null ; t2 != null ; p1 >= p2 ; p2 >= 0 ; t4 = p1 - p2 ; p3 = t4	∅
22-23	Yes	∅	positive(t1)	a1→t1 ; a2→t2 ; res→t3 ; newB→t4	t1 != null ; t2 != null ; p1 >= p2 ; p2 >= 0 ; t4 = p1 - p2 ; p3 = t4	∅
23-24	Yes	∅	positive(t2)	a1→t1 ; a2→t2 ; res→t3 ; newB→t4	t1 != null ; t2 != null ; p1 >= p2 ; p2 >= 0 ; t4 = p1 - p2 ; p3 = t4	l_{bc}, ¬(a1 = null ‖ a2 = null) ⟶ l_{c1}, acc(a2.balance) ; l_{bc}, ¬(a1 = null ‖ a2 = null) ⟶ l_{c2}, a2.balance >= 0
24-25	Yes	∅	positive(t2)	a1→t1 ; a2→t2 ; res→t3 ; newB→t4	t1 != null ; t2 != null ; p1 >= p2 ; p2 >= 0 ; t4 = p1 - p2 ; p3 = t4 ; t3 = t1	l_{bc}, ¬(a1 = null ‖ a2 = null) ⟶ l_{c1}, acc(a2.balance) ; l_{bc}, ¬(a1 = null ‖ a2 = null) ⟶ l_{c2}, a2.balance >= 0
25	l_{bc}, ¬(a1 = null ‖ a2 = null) ⟶ l_{c1}, acc(a2.balance) ; l_{bc}, ¬(a1 = null ‖ a2 = null) ⟶ l_{c2}, a2.balance >= 0 ; l_{bc}, ¬(a1 = null ‖ a2 = null) ⟶ l_{c3}, acc(positive(res))					

Figure 3.9: Contents of the symbolic state at each program point during Gradual Viper's static verification of `withdraw` in Fig. 3.7

`withdraw` example is contrived to better illustrate how Gradual Viper works and its interesting aspects.

Well-formedness of user written specifications. Gradual Viper begins static verification by first checking user written specifications, like predicate bodies, preconditions, and postconditions, for well-formedness. That is, user specifications must be self-framed and cannot contain duplicate accessibility predicates or predicates joined by the separating conjunction `&&`. Self-framing from IDF [44] simply means that a formula must contain accessibility predicates for any heap locations accessed in the formula. In gradual verification, `?` can represent these accessibility predicates. For example, in the `withdraw` program `geqTo`'s body (lines 5-6) is self-framed, because `?` can represent **`acc`**`(a1.balance)` and **`acc`**`(a2.balance)` to frame `a1.balance` and `a2.balance`. On the other hand, `positive`'s body (line 8) is classically self-framed as it explicitly contains **`acc`**`(a.balance)` to frame `a.balance`. All of the user written formulas, which are `geqTo`'s body (lines 5-6), `positive`'s body (line 8), `withdraw`'s precondition (line 12), and its postcondition (lines 13-14), are well-formed.

Next, Gradual Viper optimistically statically verifies each function in the given program, *e.g.* the `withdraw` function in our running example. This involves symbolically executing the function from top to bottom and tracking information in a symbolic state. Information is gathered from the execution of both specifications and code, and proof obligations are established by the

symbolic state. If any obligations are established optimistically, corresponding run-time checks are stored in the symbolic state. Fig. 3.9 displays the contents of the symbolic state at every program point (marked by program lines) during the verification of `withdraw`. In general, a symbolic state can be thought of as writing an intermediate logical formula in a special form. We will discuss the contents of a symbolic state in more detail as we work through the `withdraw` example.

Producing a precondition. At the start of `withdraw` (lines 15-16), information in the precondition, *e.g.* `geqTo(a1,a2)`, is *produced* or translated into an empty symbolic state resulting in the first state in the table in Fig. 3.9. As with formulas, symbolic states may be imprecise or not, meaning information may be missing from the state due to imprecision. In fact, you can think of an imprecise symbolic state as representing an imprecise intermediate formula. Here, the precondition `geqTo(a1,a2)` is precise,[5] so the state also remains precise. Had the precondition been imprecise, then the state would also become imprecise. Local variables are mapped to symbolic values in a symbolic, variable store. Since `a1` and `a2` are arguments to withdraw and `res` the return value, they are all assigned fresh symbolic values `t1`, `t2`, and `t3` respectively in the store. Then, permissions like accessibility predicates and predicates can be stored in a symbolic heap (either the optimistic heap or heap) in terms of the symbolic values. We call symbolic versions of permissions *heap chunks*. Since `geqTo(a1,a2)` is concretely known it is added directly to the heap as the heap chunk `geqTo(t1,t2)`. An important invariant of the heap is that permissions in it are guaranteed to be separated in memory, *i.e.* when they are joined by the separating conjunction they return true. The optimistic heap contains heap chunks for accessibility predicates that are optimistically assumed during verification and is introduced in this work to reduce the number of run-time checks produced by Gradual Viper. We will see how this works as we continue to discuss the `withdraw` example. For now, the optimistic heap is empty. Similarly, both the path condition and set of run-time checks both remain empty. The path condition contains constraints on symbolic values that have been collected on the current verification path. The precondition `geqTo(t1,t2)` only contains permission information, so the path condition is empty. Further, producing a formula into the symbolic state does not introduce any run-time checks.

Unfolding a predicate. Next, Gradual Viper executes the unfold statement on line 16 causing the predicate `geqTo(a1,a2)` to be *consumed* and then its body to be *produced* into the state on lines 16-17. In general, consuming a formula 1) checks whether the formula is established by the symbolic state, 2) generates minimized run-time checks for the state to establish the formula soundly, and 3) removes permissions asserted in the formula from the symbolic state. That is, consume is Gradual Viper's mechanism for checking proof obligations; and as we will see, is used a few different times throughout the verification of `withdraw`. Here, since `geqTo(t1,t2)` is in the heap, `geqTo(a1,a2)` is established by the symbolic state and no run-time checks are required. It is then removed from the heap as it is "consumed". After consumption, `geqTo(a1,a2)`'s body (lines 5-6) is produced into the current state (the one without `geqTo(t1,t2)`). The body

[5]Note, precision in a static context as in Gradual Viper means the formula does not contain `?` at the top-level. Predicates are treated as black-boxes, so even if their bodies are imprecise, as with `geqTo(a1,a2)`, a formula containing them, such as `geqTo(a1,a2)`, can be precise.

of `geqTo` is imprecise, so the symbolic state is made imprecise (as seen in Fig. 3.9 at lines 16-17). The rest of `geqTo`'s body is a boolean expression constraining `a1` and `a2`'s account balances: `a1`'s balance is greater than or equal to `a2`'s balance and `a2`'s balance is positive. Before adding these constraints to the path condition, Gradual Viper first looks for heap chunks in the current symbolic state corresponding to accessibility predicates that frame `a1.balance` and `a2.balance` in `geqTo`'s body. Both the heap and optimistic heap are empty, but the state is imprecise so the missing heap chunks are optimistically assumed to be in the state. In fact, it is sound to make this assumption without any run-time checks, because we are producing (rather than consuming) the predicate body. As a result, fresh symbolic values `p1` and `p2` for `a1.balance` and `a2.balance` respectively are generated and used to record constraints on the balances in the path condition. Gradual Viper also records that the receivers `a1` and `a2` are non-null in the path condition, because Gradual Viper assumed they can be safely de-referenced. Finally, Gradual Viper adds heap chunks **acc**`(t1,balance,p1)` and **acc**`(t2,balance,p2)` to the optimistic heap to 1) record the mappings of locations to their values and 2) avoid producing run-time checks for the accessibility predicates later in the program. These heap chunks are added to the optimistic heap rather than the heap, because `getTo`'s body does not specify whether or not `a1` (`t1`) or `a2` (`t2`) alias. So adding them to the heap would break the heap's invariant. The optimistic heap, however, does not maintain any invariants. It is also convenient to store optimistic heap chunks in their own heap signaling that they are available due to optimism in the verification. The final symbolic state after consuming `geqTo(a1,a2)` and producing its body is given in Fig. 3.9 lines 16-17.

Branching. After the unfold on line 16, Gradual Viper reaches the start of the if statement on the following line 17. As is common with static verifiers based on symbolic execution, Gradual Viper's execution branches at if statements. Execution also branches at other conditioned points, such as logical conditionals and loops. Gradual Viper analyzes the *then* branch (lines 17-19) under the assumption the condition `a1 == null|| a2 == null` is true, and the *else* branch (lines 19-25) under the assumption `a1 == null || a2 == null` is false. These assumptions are added to the path condition for each execution path respectively. However, in our example the symbolic state going into the if statement (Fig. 3.9 lines 16-17) states that both `a1` and `a2` are non-null. So the *then* branch is infeasible, and Gradual Viper prunes this execution path resulting in the blank symbolic states in Fig. 3.9 from lines 17-19. Both accounts being non-null means the *else* branch condition for sure holds and so execution proceeds down this branch without any changes to the symbolic state. That is, for `withdraw` to be statically verified, this one execution path must successfully verify. If Gradual Viper executed both branches, then determining verification success is a bit more complicated:

- If the current symbolic state is precise, then both execution paths must successfully verify. This is the default functionality in static verifiers.
- If the current symbolic state is imprecise, then verification succeeds when one or both paths successfully verify. When only one path succeeds and the state is imprecise, Gradual Viper optimistically assumes the state contains information that forces program execution down the success path only at run time. This more permissive static functionality is critical for adhering to the gradual guarantee at branch points. To ensure the program will never actually execute

the failing branch (*i.e.* to ensure soundness), Gradual Viper adds a run-time check for the success path's condition at the branch point.

Variable assignment. Let's look now at how Gradual Viper verifies the `else` branch (lines 19-25). At the variable assignment on line 20, Gradual Viper first evaluates the right-hand expression to the symbolic value `p1 - p2`. To do this, Gradual Viper first looks for heap chunks for `a1.balance` and `a2.balance` in the current symbolic heaps (Fig. 3.9 lines 19-20) to both frame the locations and get their values. Both heap chunks are in the optimistic heap, so no run-time checks are required for framing and `p1` and `p2` are used in the evaluation of the right-hand expression. Note, if Gradual Viper did not add the aforementioned heap chunks to the optimistic heap when producing the body of `geqTo(a1,a2)`, then Gradual Viper would create run-time checks for them here in the program. However, these checks would be duplicates, because Gradual Viper also checks that these heap chunks are available when ensuring the precondition `getTo(a1,a2)` holds in client contexts at calls to `withdraw`. So sound tracking of heap chunks in an optimistic heap has helped us avoid duplicating run-time checks! Finally, Gradual Viper adds a new mapping to the variable store for `newB` and its fresh symbolic value `t4`; and then, adds the constraint `t4 = p1 - p2` to the path condition to record information from the assignment in the symbolic state (Fig. 3.9 lines 20-21).

Field assignment. Next, `a1.balance` is assigned `newB`'s value in the field assignment on line 21. Gradual Viper mimics this behavior symbolically by first pulling `newB`'s value `t4` from the symbolic state (Fig. 3.9 lines 20-21). Then, Gradual Viper looks for `a1.balance`'s heap chunk in the state for framing, and if there, removes the chunk as `a1.balance`'s value may change in the write, *i.e.* **`acc`**`(a1.balance)` is *consumed*. Gradual Viper also asserts that `a1` is non-null. The heap chunk for `a1.balance` is in the optimistic heap and `a1 != NULL` is in the path condition, so no run-time checks are needed here. Then, `a1.balance`'s heap chunk is removed from the state; and, unfortunately, this action causes `a2.balance`'s heap chunk to be removed from the state as well. Gradual Viper does not know whether or not `a1` and `a2` alias, because this information does not appear in the current path condition and the optimistic heap does not maintain the separation invariant. Then, since the state is imprecise Gradual Viper could assume that `a1.balance` and `a2.balance` refer to the same heap location, *i.e.* `a1` and `a2` are aliased. In this case, removing `a1.balance`'s heap chunk requires also removing `a2.balance`'s heap chunk as `a2.balance` may have also changed with the write. On the other hand, the case where they do not alias and `a2.balance`'s heap chunk can stay in the optimistic heap is also possible from Gradual Viper's perspective. To simply and soundly cover both cases Gradual Viper removes `a2.balance`'s heap chunk by default when alias information is unknown. As we will see next, this comes at the cost of additional run-time checks later in the verification of `withdraw`. Finally, Gradual Viper *produces* a new heap chunk for `a1.balance` into the state to track its new, fresh symbolic value `p3` after the write and updates the path condition with the assignment information `p3 = t4` (Fig. 3.9 lines 21-22).

Folding a predicate. After `a1.balance` is assigned a new balance, Gradual Viper executes the fold statement on line 22. Folding a predicate is similar to unfolding a predicate except

that the functionality is reversed: `positive(a1)`'s body is *consumed* from the current state (Fig. 3.9 lines 21-22) and then `positive(a1)` is *produced* into the state after consumption (Fig. 3.9 lines 22-23). The body of `positive(a1)` is **`acc`**`(a1.balance) && a1.balance >= 0`, so **`acc`**`(a1.balance)` is consumed first then `a1.balance >= 0` second. The heap chunk **`acc`**`(t1,balance,p3)` corresponding to **`acc`**`(a1.balance)` is in the heap and `a1 != null` holds in the path condition, so no run-time check is required for consuming **`acc`**`(a1.balance)`. Then, `a1.balance`'s heap chunk is removed from the heap as seen in Fig. 3.9 lines 22-23. Next, the boolean expression `a1.balance >= 0` is evaluated to its symbolic value `p3 >= 0`. Recall that to do this, Gradual Viper must look up a heap chunk for `a1.balance` in the symbolic state to both frame the heap location and get its value. However, Gradual Viper just removed this heap chunk from the current state due to the left-to-right execution of consume. To solve this issue, Gradual Viper looks for framing and value information in the state before the fold, *i.e.* the state before consumption at lines 21-22 in Fig. 3.9. As we know, `a1.balance`'s heap chunk is in this state and maps `a1.balance` to the value `p3`, so no run-time check is needed for framing. Then, `p3 >= 0` is asserted against the current path condition. Since `p1 >= p2 >= 0` and `p3 = t4 = p1 - p2` are in the path condition, `p3` is clearly greater than or equal to 0 and is proven directly by the path condition. That is, no run-time check is needed for `p3 >= 0`. Finally, `positive(a1)` is produced into the current state, which adds `positive(t1)` to the heap resulting in the final version of the state in Fig. 3.9 lines 22-23.

Next, Gradual Viper executes the second fold statement on line 23 with the aforementioned state. This fold statement consumes `positive(a2)`'s body and then produces `positive(a2)` into the state. So as before, Gradual Viper first consumes **`acc`**`(a2.balance)` and then `a2.balance >= 0`. The receiver `a2` is proved to be non-null by the path condition; however this time, the heap chunk for **`acc`**`(a2.balance)` is not in either of the heaps. Fortunately for us, the state is imprecise and can optimistically contain this heap chunk, so a run-time check is produced for **`acc`**`(a2.balance)` as seen in the state after folding `positive(a2)` in Fig. 3.9 lines 23-24. Since this run-time check occurs down the `else` branch of the if statement in `withdraw`, branch information, *e.g.* l_{bc}, `¬(a1 = null || a2 = null)`, is included with the check. The location l_{bc} specifies where the branch point originated in the program, *e.g.* line 17, and `¬(a1 = null || a2 = null)` is the assumption made at the branch point for the current execution path. Additionally, l_{c1} contains the location where the check itself is required in the program, *e.g.* line 23. While it does not happen in the `withdraw` example, sometimes different checks are required at the same program point down different execution paths. So, Gradual Viper attaches branch information for the entire execution path to each run-time check to allow GVC0 (or other frontends) to apply checks only on the execution path they are required. This ensures soundness and reduces run-time checking. Now, Gradual Viper removes **`acc`**`(a2.balance)` from the current state (Fig. 3.9 lines 22-23), which actually causes `positive(t1)` to be removed from the heap as well. Predicates are treated as black boxes in Gradual Viper; so unless told otherwise, Gradual Viper conservatively assumes **`acc`**`(a2.balance)` is in `positive(t1)` and removes `positive(t1)` from the heap alongside **`acc`**`(a2.balance)`. The only way Gradual Viper can guarantee **`acc`**`(a2.balance)` is not in `positive(t1)` is if a heap chunk for `a2.balance` and `positive(t1)` both exist in the heap, as the heap maintains the separation invariant. In this case, `positive(t1)` can remain in the heap while only **`acc`**`(a2.balance)` is removed. Of course, in our example the heap chunk for `a2.balance` is definitely not in the

heap, so `postive(t1)` is removed.

Continuing, Gradual Viper consumes `a2.balance >= 0`, which first looks for a heap hunk to frame `a2.balance` in the state before the consume (Fig. 3.9 lines 22-23). However, neither of the heaps contain a heap chunk for `a2.balance`. As before, Gradual Viper uses imprecision to optimistically assume the heap chunk is in the state and produces a run-time check for **acc**`(a2.balance)`. Since this run-time check for the same location already exists in the state, the two checks are condensed into the first one. Then, Gradual Viper returns a fresh symbolic value for `a2.balance`, say `p4`, to evaluate `a2.balance >= 0` down to `p4 >= 0`. Note, Gradual Viper can only return a fresh value here, because the heaps do not contain a heap chunk recording `a2.balance`'s value in the state. Unfortunately, this means Gradual Viper cannot prove `a2.balance >= 0` holds as no constraints exist for `p4` in the path condition. But, this also means `p4 >= 0` does not contradict existing information in the path condition. So, imprecision in the state can optimistically represent `p4 >= 0` and a run-time check for `a2.balance >= 0` is generated as seen in Fig. 3.9 lines 23-24. A few things of note here:

- Run-time checks are originally computed in terms of symbolic values, *e.g.* `p4 >= 0`, but are ultimately replaced with counterparts written in terms of program variables, *e.g.* `a2.balance >= 0`. This replacement by the *translate* function in Gradual Viper simplifies the implementation of run-time checks for frontends like GVC0, which operate on program variables and concrete values not symbolic values. The `translate` function uses mappings in the symbolic heaps and store to reverse the symbolic execution. Special considerations are made for fresh symbolic values like `p4`, aliasing between object values, and different variable contexts.
- On another note, if consuming **acc**`(a1.balance)` at the field assignment on line 21 did not also consume the heap chunk for `a2.balance`, then the run-time checks for **acc**`(a2.balance)` and `a2.balance >= 0` would not be necessary. Gradual Viper conservatively assumed `a1` and `a2` were aliased at the consume, so it removed both chunks from the state. However, in practice `a1` and `a2` are likely to be distinct objects; and in fact, folding `positive(a1)` then `positive(a2)` is a good sign the developer of `withdraw` expects `a1` and `a2` to be distinct. In this case, `a2.balance`'s heap chunk does not need to be removed making the aforementioned run-time checks unnecessary. Unfortunately, since we designed Gradual Viper to be conservative for simplicity, these run-time checks are only eliminated when the developer explicitly specifies that `a1` and `a2` are not aliased, such as in the precondition of `withdraw`. So, we are trading more optimal run-time checks for simplicity in our consume algorithm.
- While it does not happen here in our `withdraw` example, there may be times where parts of a symbolic, boolean expression are proven statically and the rest optimistically. In this case, Gradual Viper re-writes the expression into conjunctive normal form and computes the conjuncts in this form that cannot be proven statically by the path condition. These conjuncts (after translation) will then be checked at run time. We call this process computing the *difference* between the expression and the path condition, and it results in minimized run-time checks given statically available information.

Finally, a heap chunk for `positive(a2)` (*e.g.* `positive(t2)`) is produced into the heap resulting in the final form of the state in Fig. 3.9 lines 23-24.

Return value assignment. Then, Gradual Viper reaches the variable assignment on line 24, which assigns `a1` to `res`—the return value of `withdraw`. Gradual Viper first looks up the symbolic value `t1` for `a1` and then the symbolic value `t3` for `res` in the variable store. Gradual Viper stores the information `t3 = t1` from the assignment in the path condition resulting in the next symbolic state in Fig. 3.9 lines 24-25.

Consuming a postcondition. Finally, Gradual Viper reaches the end of `withdraw` down its one and only execution path on line 25. So the last thing Gradual Viper must do to verify the function, is to *consume* the postcondition `? && acc(positive(a2)) && acc(positive(res))` (lines 13-14) in the current symbolic state (Fig. 3.9 lines 24-25). Gradual Viper begins by first consuming `positive(a2)` then `positive(res)`. The heap chunk for `positive(a2)` is in the heap, so no run-time check is needed for it. Then `positive(t2)` is removed from the heap leaving both symbolic heaps empty. As a result (and because the state is imprecise), consuming `positive(res)` in the next step results in a run-time check for the predicate as seen in the final set of run-time checks required for `withdraw` given in Fig. 3.9 line 25. Note, consuming `acc(positive(a2)) && acc(positive(res))` requires both consuming the predicates individually (which we've done) and ensuring that one predicate does not access heap locations overlapping with the other (in adherence with the separating conjunction `&&`). Unfortunately, the state does not contain enough information to prove this fact statically, *e.g.* only the heap chunk for `positive(a2)` appears in the heap, but the state is imprecise! So when Gradual Viper optimistically assumes `positive(res)` holds, it also assumes `positive(res)` is separated from `positive(a2)`. Gradual Viper flags `positive(res)`'s run-time check with this additional check for GVC0 to handle. Additionally, after consuming the static part of an imprecise formula, *e.g.* `acc(positive(a2)) && acc(positive(res))` in `withdraw`'s postcondition, Gradual Viper makes the state imprecise and empties both symbolic heaps. The `?` in the imprecise formula can represent any permission available in the state, so they must be removed by *consume*.

Takeaways. To summarize, Gradual Viper statically verifies the `withdraw` function successfully, and produces run-time checks for `acc(a2.balance)` before line 21, `a2.balance >= 0` also before line 21, and `positive(res)` at the end of `withdraw` (line 25). The `withdraw` function will be completely verified if these checks succeed at run time. During our discussion of the `withdraw` function, we highlighted a number of technical challenges addressed and solutions developed related to designing and implementing Gradual Viper. One of our goals was for Gradual Viper to minimize run-time checks as much as possible without using highly complex algorithms. For this we introduced the optimistic heap, which tracks heap chunks that are optimistically assumed during static verification and can be soundly used to reduce run-time checking in successive program statements from where they originated. In `withdraw`, we saw the heap chunks for `a1.balance` and `a2.balance`, which were added to the optimistic heap during the production of `geqTo(a1,a2)`'s body (line 16), be used to eliminate duplicate run-time checks at the assignment on line 20. We had to make careful considerations for the separating conjunction and removal of heap chunks at consumes to ensure sound tracking of heap chunks in the optimistic heap. We also defined and implemented the `diff` function, which utilizes conjunctive normal form to optimize run-time checks for boolean expressions. Finally, Gradual

Viper conservatively removes heap chunks from the symbolic heaps that may alias with other heap chunks removed at a *consume*. We saw in `withdraw` that this comes at the cost of additional run-time checks: consuming `a1.balance`'s heap chunk at the field assignment on line 21 also consumed `a2.balance`'s heap chunk resulting in run-time checks for **acc**`(a2.balance)` and `a2.balance >= 0` before line 21. That is, we are trading more optimal run-time checks for simplicity in our consume algorithm.

Another goal for Gradual Viper, is for it to use symbolic execution for static reasoning. We accomplished this goal, but not without dealing with some technical challenges. Symbolic execution based static verifiers generate and discharge proof obligations written in terms of symbolic values, causing Gradual Viper, which extends this system, to follow suit. As a result, Gradual Viper naturally generates run-time checks written in terms of symbolic values as well. Unfortunately, dynamic verifiers only operate on program variables and concrete values not symbolic ones. To bridge this gap between the static and dynamic systems, we implemented a `translate` function in Gradual Viper that re-writes run-time checks containing symbolic values to ones containing program variables and concrete values while being careful about aliases. Finally, execution splitting at branch points led to some trickiness in gradual verification. Different run-time checks may appear at the same program point along different execution paths, so we augmented Gradual Viper to attach branching information to run-time checks. We also augmented Gradual Viper to be more optimistic about verification success when dealing with failing execution paths in the presence of imprecision. This was done in compliance with the gradual guarantee.

3.4 Gradual Viper: Implemented Algorithm

In this section, we formalize the symbolic execution algorithm implemented by Gradual Viper. A high-level description of how it works is given in §3.3. Our algorithm extends Viper's symbolic execution algorithm, and so Gradual Viper's design is heavily influenced by Müller et al. [33]'s work. Like Viper, Gradual Viper's algorithm consists of 4 major functions: eval, produce, consume, and exec. The functions evaluate expressions, produce (inhale) and consume (exhale) formulas, and execute program statements respectively. Following Viper's lead, our 4 functions are defined in continuation-passing style, where the last argument of each of the aforementioned functions is a continuation Q. The continuation is a function that represents the remaining symbolic execution that still needs to be performed. Note that the last continuation returns a boolean (λ _ . success`()` or λ _ . failure`()`), indicating whether or not symbolic execution was successful.

The rest of this section is outlined as follows. Run-time checks and the collections that hold them are described in §3.4.1. We define symbolic states in §3.4.2 and preliminaries in §3.4.3. Finally, the 4 major functions of our algorithm are given in their own sections: eval §3.4.4, produce §3.4.5, consume §3.4.6, and exec §3.4.7. Throughout this section, we make clear where Viper has been extended to support imprecise formulas with yellow highlighting in figures. We also use blue highlighting to indicate extensions for run-time check generation and collection.

3.4.1 Run-time checks

Run-time checks produced by Gradual Viper are collected in the $\mathfrak{R}$ set. A run-time check is a 4-tuple $(\texttt{bcs}_c,\ \texttt{origin}_c,\ \texttt{location}_c,\ \phi_c)$, where $\texttt{bcs}_c$ is a set of branch conditions, $\texttt{origin}_c$ and $\texttt{location}_c$ denote where the run-time check is required in the program, and ϕ_c is what must be checked. A branch condition in $\texttt{bcs}_c$ is also a tuple of $(\texttt{origin}_e,\ \texttt{location}_e,\ e)$, where $\texttt{origin}_e$ and $\texttt{location}_e$ define the program location at which Gradual Viper's execution branches on the condition e. A `location` is the AST element in the program where the branch or check occurs, denoted as a formula ϕ_l. Sometimes, the condition being checked is defined elsewhere in the program (*e.g.* in the precondition of a method) but we need to relate it to the method being verified. The `origin` is used to do this. It is `none` when the condition is in the method being verified; otherwise, it contains a method call, fold, unfold, or special loop statement from the method being verified that referenced the check specified in the `location`. An example run-time check is: $(\{(\texttt{none},\ x > 2,\ \neg(x > 2))\},\ z := m(y),\ \textbf{acc}(y.f),\ \textbf{acc}(y.f))$. The check is for accessing $y.f$, and it is required for m's precondition element $\textbf{acc}(y.f)$ at the method call statement $z := m(y)$. The check is only required when $\neg(x > 2)$, which is evaluated at the program point where the AST element $x > 2$ exists. Since $\neg(x > 2)$'s `origin` is `none`, it comes from an if or assert statement.

Further, $\mathcal{R}$ is used to collect run-time checks down a particular execution path in Gradual Viper. $\mathcal{R}$ is a 3-tuple $(\texttt{bcs}_p,\ \texttt{origin}_p,\ \texttt{rcs}_p)$ where $\texttt{bcs}_p$ is the set of branch conditions collected down the execution path p, $\texttt{origin}_p$ is the current origin that is set and reset during execution, and `rcs` is the set of run-time checks collected down p. Two auxiliary functions are used to modify $\mathcal{R}$: addcheck and addbc. The addcheck function takes an $\mathcal{R}$ collection $\mathcal{R}_{arg}$, a `location` ϕ_l for a check, and the check itself, and returns a copy of $\mathcal{R}_{arg}$ with the run-time check added to $\mathcal{R}_{arg}.\texttt{rcs}$. If necessary, addcheck uses $\mathcal{R}_{arg}.\texttt{origin}$ and substitution to ensure ϕ_l and the check refer to the correct context. For example, let ϕ_l and check ϕ_c come from asserting a precondition for $z := m(y)$. Then, addcheck performs the substitutions: $\phi_l[t \mapsto m_{arg}]$ (precondition declaration context) and $\phi_c[t \mapsto y]$ (method call context) where t is the symbolic value for y. The addbc function operates similarly to addcheck but for branch conditions.

3.4.2 Symbolic State

We use $\sigma \in \Sigma$ to denote a symbolic state, which is a 6-tuple
$(\texttt{isImprecise},\ h_?,\ h,\ \gamma,\ \pi,\ \mathcal{R})$ consisting of a boolean `isImprecise`, a symbolic heap $h_?$, another symbolic heap h, a symbolic store γ, a path condition π, and a collection $\mathcal{R}$ (defined in §3.4.1). The boolean `isImprecise` records whether or not the state is imprecise, the symbolic store γ maps local variables to their symbolic values, and the path condition π (defined in §3.4.3) contains constraints on symbolic values that have been collected on the current verification path.

A symbolic heap is a multiset of heap chunks for fields or predicates that are currently accessible. A field chunk $id(r;\ \delta)$ (representing expression $r.id$) consists of the field name id, the receiver's symbolic value r, and the field's symbolic value δ—also referred to as the *snapshot* of a heap chunk. For a predicate chunk $id(args;\ \delta)$, id is the predicate name, $args$ is a list of symbolic values that are arguments to the predicate, and δ is the snapshot of the predicate. A predicate's snapshot represents the values of the heap locations abstracted over by the predicate.

The symbolic, *optimistic* heap $h_?$ contains heap chunks that are accessible due to optimism in the symbolic execution, while h contains heap chunks that are statically accessible. Further, only h maintains the invariant that its heap chunks are separated in memory, and thus, can be joined successfully by the separating conjunction. The *empty symbolic state* is
$\sigma_0 = (\texttt{isImprecise} := \texttt{false},\ h_? := \emptyset,\ h := \emptyset,\ \gamma := \emptyset,\ \pi := \emptyset, \mathcal{R} := (\emptyset,\ \texttt{none},\ \emptyset))$.

3.4.3 Preliminaries

We introduce a few preliminary definitions here that will be helpful later. A path condition π is a stack of tuples $(id,\ bc,\ pcs)$. An id is a unique identifier that determines the constraints on symbolic values that have been collected between two branch points in execution. The bc entry is the symbolic value for the branch condition from the first of two branch points, and pcs is the set of constraints that have been collected. Branch points can be from if statements and logical conditionals in formulas. Functions pc-all, pc-add, and pc-push manipulate path conditions and are formally defined in Appendix Fig. A.11. The pc-all function collects and returns all the constraints in π, pc-add adds a new constraint to π, and pc-push adds a new stack entry to π. Similarly, snapshots for heap chunks have their own related functions: *unit*, *pair*, *first*, and *second*. The constant *unit* is the empty snapshot, *pair* constructs pairs of snapshots, and *first* and *second* deconstruct pairs of snapshots into their sub-parts. Further, fresh is used to create fresh snapshots, symbolic values, and other identifiers depending on the context. The havoc function similarly updates a symbolic store by assigning a fresh symbolic value to each variable in a given collection of variables. Finally, check$(\pi,\ t)$ = pc-all$(\pi) \Rightarrow t$ queries the underlying SAT solver to see if the given constraint t is valid in a given path condition π (*i.e.* π proves or implies t).

3.4.4 Symbolic execution of expressions

The symbolic execution of expressions by the eval function is defined in Fig. 3.10. Using the current symbolic state, eval evaluates an expression to a symbolic value t and returns t and the current state to the continuation Q. Variable values are looked up in the symbolic store and returned. For $op(\overline{e})$, its arguments $\overline{e}$ are each evaluated to their symbolic values $\overline{t}$. A symbolic value $op'(\overline{t})$ is then created and returned with the state after evaluation. Each op has a corresponding symbolic value op' of the same arity. For example, $e_1 + e_2$ results in the symbolic value $add(t_1, t_2)$ where e_1 and e_2 evaluate to t_1 and t_2 respectively.

Finally, the most interesting rule is for fields $e.f$. The receiver e is first evaluated to t resulting in a new state σ_2. Then, eval looks for a heap chunk for $t.f$ first in the current heap h.[6] If a chunk exists, then the heap read succeeds and σ_2 and the chunk's snapshot δ is returned to the continuation. If a chunk does not exist in h, then eval looks for a chunk in the optimistic heap $h_?$, and if found the chunk's snapshot is returned with σ_2. If a heap chunk for $t.f$ is not found in either heap, then the heap read can still succeed when σ_2 is imprecise. As long as $t \neq \texttt{null}$ does not contradict the current path condition $\sigma_2.\pi$ (the call to assert, Appendix Fig. A.18), σ_2's

[6]Heap lookup in eval also looks for heap chunks that are aliases (according to the path condition) to the chunk in question.

$$
\begin{aligned}
&\mathsf{eval}(\sigma,\ t,\ Q) &&= Q(\sigma,\ t)\\
&\mathsf{eval}(\sigma,\ x,\ Q) &&= Q(\sigma,\ \sigma.\gamma(x))\\
&\mathsf{eval}(\sigma_1,\ op(\overline{e}),\ Q) &&= \mathsf{eval}(\sigma_1,\ \overline{e},\ (\lambda\ \sigma_2,\ \overline{t}\ .\ Q(\sigma_2,\ op'(\overline{t}))))\\
&\mathsf{eval}(\sigma_1,\ e.f,\ Q) &&= \mathsf{eval}(\sigma_1,\ e,\ (\lambda\ \sigma_2,\ t\ .\\
& &&\quad \texttt{if}\ \ (\exists\ f(r;\ \delta) \in \sigma_2.h\ .\ \mathsf{check}(\sigma_2.\pi,\ r = t))\ \ \texttt{then}\\
& &&\quad\quad Q(\sigma_2,\ \delta)\\
& &&\quad \texttt{else if}\ \ (\exists\ f(r;\ \delta) \in \sigma_2.h_?\ .\ \mathsf{check}(\sigma_2.\pi,\ r = t))\ \ \texttt{then}\\
& &&\quad\quad Q(\sigma_2,\ \delta)\\
& &&\quad \texttt{else if}\ \ (\sigma_2.\texttt{isImprecise})\ \ \texttt{then}\\
& &&\quad\quad res,\ _ := \mathsf{assert}(\sigma_2.\texttt{isImprecise},\ \sigma_2.\pi,\ t \neq \texttt{null})\\
& &&\quad\quad e_t := \mathsf{translate}(\sigma_2,\ t)\\
& &&\quad\quad \mathcal{R}' := \mathsf{addcheck}(\sigma_2.\mathcal{R},\ e.f,\ \mathbf{acc}(e_t.f))\\
& &&\quad\quad \delta := \mathsf{fresh}\\
& &&\quad\quad res \wedge Q(\sigma_2\{h_? := \sigma_2.h_? \cup f(t;\ \delta),\ \pi := \mathsf{pc\text{-}add}(\sigma_2.\pi,\ \{t \neq \texttt{null}\}),\ \mathcal{R} := \mathcal{R}'\},\ \delta)\\
& &&\quad \texttt{else}\ \ \mathsf{failure}()
\end{aligned}
$$

Handles imprecision	Handles run-time check generation and collection

Figure 3.10: Rules for symbolically executing expressions

imprecision optimistically provides access to $t.f$. Therefore, a run-time check for $\mathbf{acc}(e_t.f)$ is created and added to σ_2's set of run-time checks (highlighted in blue). Note that $e_t.f$ is used in the check rather than $t.f$, because—unlike t which is a symbolic value—the expression e_t can be evaluated at run time. Specifically, translate (described in Appendix Fig. A.15) is called on t with the current state σ_2 to compute e_t. Additionally, the AST element $e.f$ is used to denote the check's location.

Afterwards, a fresh snapshot δ is created for $t.f$'s value, and a heap chunk $f(t;\ \delta)$ for $t.f$ and δ is created and added to σ_2's optimistic heap passed to the continuation. Similarly, the constraint $t \neq \texttt{null}$ is added to σ_2's path condition. By adding $f(t;\ \delta)$ to the optimistic heap, the following accesses of $t.f$ are statically verified by the optimistic heap, which reduces the number of run-time checks produced. Finally, verification of the heap read for $t.f$ fails when none of the aforementioned cases are true. Fig. A.12 and Fig. A.13 in the Appendix define variants of eval, called eval-p and eval-c, that are used in produce and consume respectively. The eval-p variant does not introduce run-time checks and eval-c does not extend the optimistic heap and path condition, because the aforementioned functionalities are not needed in these contexts.

3.4.5 Symbolic production of formulas

Produce (Fig. 3.11) is responsible for adding information to the symbolic state, in particular, the path condition and the heap h. Producing an imprecise formula makes the symbolic state

$$
\begin{aligned}
&\text{produce}(\sigma,\ ?\ \&\&\ \phi,\ \delta,\ Q) &&= \text{produce}(\sigma\{\mathtt{isImprecise}\ := \mathtt{true}\},\ \phi,\ \text{second}(\delta),\ Q)\\
&\text{produce}(\sigma_1,\ e,\ \delta,\ Q) &&= \text{eval-p}(\sigma_1,\ e,\ (\lambda\ \sigma_2,\ t\ .\ Q(\sigma_2\{\pi\ := \text{pc-add}(\sigma_2.\pi,\ \{t, \delta = unit\})\})))\\
&\text{produce}(\sigma_1,\ \mathbf{acc}(p(\overline{e})),\ \delta,\ Q) &&= \text{eval-p}(\sigma_1,\ \overline{e},\ (\lambda\ \sigma_2,\ \overline{t}\ .\ Q(\sigma_2\{h\ := \sigma_2.h \uplus p(\overline{t};\ \delta)\})))\\
&\text{produce}(\sigma_1,\ \mathbf{acc}(e.f),\ \delta,\ Q) &&= \text{eval-p}(\sigma_1,\ e,\ (\lambda\ \sigma_2,\ t\ .\\
& &&\quad Q(\sigma_2\{h\ := \sigma_2.h \uplus f(t;\ \delta),\ \pi\ := \text{pc-add}(\sigma_2.\pi,\ \{t \neq \mathtt{null}\})\})))\\
&\text{produce}(\sigma_1,\ \phi_1\ \&\&\ \phi_2,\ \delta,\ Q) &&= \text{produce}(\sigma_1,\ \phi_1,\ \text{first}(\delta),\ (\lambda\ \sigma_2\ .\ \text{produce}(\sigma_2,\ \phi_2,\ \text{second}(\delta),\ Q)))\\
&\text{produce}(\sigma_1,\ e\ ?\ \phi_1\ :\ \phi_2,\ \delta,\ Q) &&= \text{eval-p}(\sigma_1,\ e,\ (\lambda\ \sigma_2,\ t\ .\\
& &&\quad \text{branch}(\sigma_2,\ e,\ t,\ (\lambda\ \sigma_3\ .\ \text{produce}(\sigma_3,\ \phi_1,\ \delta,\ Q)),\ (\lambda\ \sigma_3\ .\\
& &&\quad\ \text{produce}(\sigma_3,\ \phi_2,\ \delta,\ Q)))))
\end{aligned}
$$

Handles imprecision   Handles run-time check generation and collection

Figure 3.11: Rules for symbolically producing formulas

imprecise. The produce rule for an expression e evaluates e to its symbolic value and produces it into the path condition. The produce rules for accessibility predicates containing fields and predicates are similar, so we focus on the rule for fields only. The field $e.f$ in **acc**$(e.f)$ first has its receiver e evaluated to a symbolic value t. Then, using the parameter δ a fresh heap chunk $f(t;\ \delta)$ is created and added to the heap before invoking the continuation. Note, the disjoint union $\uplus$ ensures $f(t;\ \delta)$ is not already in the heap before $f(t;\ \delta)$ is added; otherwise, verification fails. Further, **acc**$(e.f)$ implies $e \neq \mathtt{null}$ and so that fact is recorded in the path condition as $t \neq \mathtt{null}$. When the separating conjunction $\phi_1\ \&\&\ \phi_2$ is produced, ϕ_1 is first produced into the symbolic state, followed by ϕ_2. Finally, to produce a conditional, Gradual Viper branches on the symbolic value t for the condition e splitting execution along two different paths. Along one path ϕ_1 is produced into the state under the assumption that t is true, and along the other path ϕ_2 is produced under the $\neg t$ assumption. Both paths follow the continuation to the end of its execution, and a branch condition corresponding to the t assumption made is added to the symbolic state. Paths are pruned when they are infeasible (the assumption about t would contradict the current path conditions). Overall verification success is computed from the results of the two execution paths, and an imprecise state allows this computation to be optimistic when one path successfully verifies and the other doesn't. In this case, branch optimistically marks verification a success when normally it should fail, because the state may optimistically contain information that prunes the failure case. A run-time check is then added for the success path's condition to ensure soundness. This functionality is important for adhering to the gradual guarantee. The formal definition of branch is in Fig. 3.12, and other details for branch and produce are given in Chpt. 3's Appendix §A.2.2. Note, produce only adds run-time checks for branching to the symbolic state.

3.4.6 Symbolic consumption of formulas

The goals of consume are 3-fold: 1) given a symbolic state σ and formula $\widetilde{\phi}$, check whether $\widetilde{\phi}$ is established by σ, *i.e.* $\widetilde{\phi}_\sigma \mathrel{\widetilde{\Rrightarrow}} \widetilde{\phi}$ where $\widetilde{\phi}_\sigma$ is the formula which represents the state σ, 2)

```
branch(σ, e, t, Q_t, Q_¬t) =
  (π_T, R_T) := (pc-push(σ.π, fresh, t), addbc(σ.R, e, e))
  (π_F, R_F) := (pc-push(σ.π, fresh, ¬t), addbc(σ.R, e, ¬e))
  if (σ.isImprecise) then
    res_T := (if ¬check(σ.π, ¬t) then Q_t(σ{π := π_T, R := R_T}) else failure())
    res_F := (if ¬check(σ.π, t) then Q_¬t(σ{π := π_F, R := R_F}) else failure())
    if ((res_T ∧ ¬res_F) ∨ (¬res_T ∧ res_F)) then
      R' := addcheck(σ.R, e, (if (res_T) then e else ¬e))
      ℜ := ℜ ∪ R'.rcs.last
    res_T ∨ res_F
  else
    (if ¬check(σ.π, ¬t) then Q_t(σ{π := π_T, R := R_T}) else success()) ∧
    (if ¬check(σ.π, t) then Q_¬t(σ{π := π_F, R := R_F}) else success())
```

Handles imprecision | Handles run-time check generation and collection

Figure 3.12: Formally defining the branch function

produce and collect run-time checks that are minimally sufficient for σ to establish $\widetilde{\phi}$ soundly, and 3) remove accessibility predicates and predicates that are asserted in $\widetilde{\phi}$ from σ. The rules for consume are given in full and described in great detail in Chpt. 3's Appendix §A.2.3. We give select rules in Fig. 3.13 and an abstract description here.

The functionality of consume is split across two functions: consume, which is the interface to consume accepting only a state, formula, and continuation, and consume', which is a helper function performing consume's major functionality. Note, before calling consume', consume first adds non-alias information from the heap to the path condition and checks that the heap and path condition are non-contradictory using consolidate [39].

Then, the two functions work together to accomplish the aforementioned goals. For the first and second goals, heap chunks representing accessibility predicates and predicates in $\widetilde{\phi}$ are looked up in the heap h and optimistic heap $h_?$ from σ. When σ is precise, the heap chunks must be in h or verification fails. If σ is imprecise, then the heap chunks are always justified either by the heaps or imprecision. Run-time checks for heap chunks that are verified by imprecision are collected in $\sigma.\mathcal{R}$. The consume' rule for **acc**$(e.f)$ (and the rule for **acc**$(p(\overline{e}))$ which is similar) supports this functionality by calling heap-rem-acc (defined in Appendix Fig. A.17) for the look-up, assigning the boolean results to b_1 and b_2, and then using them in `if-then-else` and `else-if` casing. The blue highlighting in the `isImprecise` is `true` case in the aforementioned rule handles the run-time checks. Clauses in $\widetilde{\phi}$ containing logical expressions are first evaluated to a symbolic value t, which is then checked against σ's path condition π. If σ is precise, then pc-all$(\pi) \Rightarrow t$ must hold (*i.e.* the constraints in π prove t) or verification fails. In contrast, when

$$\textsf{consume}(\sigma_1,\ \theta,\ Q) = \sigma_2 := \sigma_1\{\ h,\ \pi := \textsf{consolidate}(\sigma_1.h,\ \sigma_1.\pi)\ \}$$
$$\textsf{consume'}(\sigma_2,\ \sigma_2.\texttt{isImprecise},\ \sigma_2.h_?,\ \sigma_2.h,\ \theta,\ (\lambda\ \sigma_3,\ h'_?,\ h_1,\ \delta_1\ .$$
$$Q(\sigma_3\{\ h_? := h'_?,\ h := h_1\},\ \delta_1)))$$
$$\textsf{consume}(\sigma_1,\ ?\ \&\&\ \phi,\ Q) = \sigma_2 := \sigma_1\{\ h,\ \pi := \textsf{consolidate}(\sigma_1.h,\ \sigma_1.\pi)\ \}$$
$$\textsf{consume'}(\sigma_2,\ \texttt{true},\ \sigma_2.h_?,\ \sigma_2.h,\ \phi,\ (\lambda\ \sigma_3,\ h'_?,\ h_1,\ \delta_1\ .$$
$$Q(\sigma_3\{\ \texttt{isImprecise} := \texttt{true},\ h_? := \emptyset,\ h := \emptyset\ \},\ pair(unit,\ \delta_1))))$$

$$\textsf{consume'}(\sigma,\ f_?,\ h_?,\ h,\ (e,t),\ Q) = res,\ \bar{t} := \textsf{assert}(\sigma.\texttt{isImprecise},\ \sigma.\pi,\ t)$$
$$\mathcal{R}' := \textsf{addcheck}(\sigma.\mathcal{R},\ e,\ \textsf{translate}(\sigma,\ \bar{t}))$$
$$res \wedge Q(\sigma\{\ \mathcal{R} := \mathcal{R}'\ \},\ h_?,\ h,\ unit)$$
$$\textsf{consume'}(\sigma_1,\ f_?,\ h_?,\ h,\ e,\ Q) = \textsf{eval-c}(\sigma_1\{\ \texttt{isImprecise} := f_?\ \},\ e,\ (\lambda\ \sigma_2,\ t\ .$$
$$\textsf{consume'}(\sigma_2\{\ \texttt{isImprecise} := \sigma_1.\texttt{isImprecise}\ \},\ f_?,\ h_?,\ h,\ (e,t),\ Q)))$$
$$\textsf{consume'}(\sigma_1,\ f_?,\ h_?,\ h,\ \textbf{acc}(e.f),\ Q) = \textsf{eval-c}(\sigma_1\{\ \texttt{isImprecise} := f_?\ \},\ e,\ (\lambda\ \sigma_2,\ t\ .$$
$$\sigma_3 := \sigma_2\{\texttt{isImprecise} := \sigma_1.\texttt{isImprecise}\}$$
$$res,\ \bar{t} := \textsf{assert}(\sigma_3.\texttt{isImprecise},\ \sigma_3.\pi,\ t \neq \texttt{null})$$
$$res \quad \wedge \quad ($$
$$\mathcal{R}' := \textsf{addcheck}(\sigma_3.\mathcal{R},\ \textbf{acc}(e.f),\ \textsf{translate}(\sigma_3,\ \bar{t}))$$
$$(h_1,\ \delta_1,\ b_1) := \textsf{heap-rem-acc}(\sigma_3.\texttt{isImprecise},\ h,\ \sigma_3.\pi,\ f(t))$$
$$\texttt{if}\ (\sigma_3.\texttt{isImprecise})\ \texttt{then}$$
$$(h'_?,\ \delta_2,\ b_2) := \textsf{heap-rem-acc}(\sigma_3.\texttt{isImprecise},\ h_?,\ \sigma_3.\pi,\ f(t))$$
$$\texttt{if}\ (b_1 = b_2 = \texttt{false})\ \texttt{then}$$
$$\mathcal{R}'' := \textsf{addcheck}(\mathcal{R}',\ \textbf{acc}(e.f),\ \textbf{acc}(\textsf{translate}(\sigma_3,\ t).f))$$
$$\texttt{else}\ \ \mathcal{R}'' := \mathcal{R}'$$
$$Q(\sigma_3\{\ \mathcal{R} := \mathcal{R}''\ \},\ h'_?,\ h_1,\ (\texttt{if}\ (b_1)\ \texttt{then}\ \delta_1\ \texttt{else}\ \delta_2))$$
$$\texttt{else}\ \texttt{if}\ (b_1)\ \texttt{then}\ Q(\sigma_3\{\ \mathcal{R} := \mathcal{R}'\ \},\ \sigma_3.h_?,\ h_1,\ \delta_1)$$
$$\texttt{else}\ \ \textsf{failure}()\ )))$$

Handles imprecision   Handles run-time check generation and collection

Figure 3.13: *Select rules* for symbolically consuming formulas

$$\begin{aligned}
&\text{exec}(\sigma_1,\ x.f\ := e,\ Q) = \text{eval}(\sigma_1,\ e,\ (\lambda\ \sigma_2,\ t\ .\ \text{consume}(\sigma_2,\ \textbf{acc}(x.f),\ (\lambda\ \sigma_3,\ _\ .\\
&\quad \text{produce}(\sigma_3,\ \textbf{acc}(x.f)\ \&\&\ x.f = t,\ pair(\text{fresh},\ unit),\ Q))))\\
&\text{exec}(\sigma_1,\ \overline{z}\ := m(\overline{e}),\ Q) = \text{eval}(\sigma_1,\ \overline{e},\ (\lambda\ \sigma_2,\ \overline{t}\ .\\
&\quad \mathcal{R}' := \sigma_2.\mathcal{R}\{\texttt{origin} := (\sigma_2,\ \overline{z} := m(\overline{e}),\ \overline{t})\}\\
&\quad \text{consume}(\sigma_2\{\ \mathcal{R} := \mathcal{R}'\ \},\ meth_{pre}[\overline{meth_{args} \mapsto t}],\ (\lambda\ \sigma_3,\ \delta\ .\\
&\quad\quad \texttt{if}\ (\text{equi-imp}(meth_{pre}))\ \texttt{then}\\
&\quad\quad\quad \sigma_4 := \sigma_3\{\texttt{isImprecise} := \texttt{true},\ h_? := \emptyset,\ h := \emptyset,\ \gamma := \text{havoc}(\sigma_3.\gamma,\ \overline{z})\}\\
&\quad\quad \texttt{else}\quad \sigma_4 := \sigma_3\{\gamma := \text{havoc}(\sigma_3.\gamma,\ \overline{z})\}\\
&\quad\quad \text{produce}(\sigma_4,\ meth_{post}[\overline{meth_{args} \mapsto t}][\overline{meth_{ret} \mapsto z}],\ \text{fresh},\\
&\quad\quad\quad (\lambda\ \sigma_5\ .\ Q(\sigma_5\{\ \mathcal{R} := \sigma_5.\mathcal{R}\{\texttt{origin} := \texttt{none}\}\ \}))))))
\end{aligned}$$

Handles imprecision | Handles run-time check generation and collection

Figure 3.14: *Select rules* for symbolically executing program statements

σ is imprecise, $\bigwedge$ pc-all $(\pi) \wedge t$ must hold (*i.e.* t does not contradict constraints in π) otherwise verification fails. In this case, a run-time check is added to $\sigma.\mathcal{R}$ for the set of residual symbolic values in t that cannot be proved statically by π. The consume' rules for expressions and symbolic values implement this behavior. The call to assert (defined in Appendix Fig. A.18) checks t against π and returns the result and any residual symbolic values. Note, assert uses diff from Chpt. 3's Appendix §A.2.1 to compute the residuals. The part highlighted in blue adds the run-time check for the residuals to the state. Finally, fields used in $\widetilde{\phi}$ must have corresponding heap chunks in h when σ and $\widetilde{\phi}$ are precise; otherwise when σ or $\widetilde{\phi}$ are imprecise, field access can be justified by either the heaps or imprecision. A run-time check containing an accessibility predicate for the field is added to $\sigma.\mathcal{R}$ when imprecision is relied on. This is all handled by the second argument $f_?$ to consume' and eval-c called by consume' on expressions.

The third goal of consume is to remove heap chunks $\overline{hc_i}$ representing accessibility predicates and predicates in $\widetilde{\phi}$ from σ, and in particular, from heaps h and $h_?$. When σ and $\widetilde{\phi}$ are both precise, the heap chunks in $\overline{hc_i}$ are each removed from h ($h_?$ is empty here). If $\widetilde{\phi}$ is imprecise, then all heap chunks in both heaps are removed as they may be in $\overline{hc_i}$ or $\widetilde{\phi}$ may represent them with imprecision. Finally, when σ is imprecise and $\widetilde{\phi}$ is precise, any heap chunks in h or $h_?$ that overlap with or may potentially overlap with (thanks to σ's imprecision) heap chunks in $\overline{hc_i}$ are removed. The calls to heap-rem-acc (and its counterpart heap-rem-pred) in consume', the extra heaps tracked in consume', and the heap assignments in the continuations from consume come together to implement heap chunk removal.

3.4.7 Symbolic execution of statements

The exec rules in Gradual Viper, which symbolically execute program statements, are largely unchanged from Viper. The only differences are 1) the rules now utilize versions of eval, produce, consume, and branch defined previously in this paper and 2) the rules track `origins` where

appropriate. To provide an intuition, select rules for exec are given in Fig. 3.14; the full set of rules are listed in Chpt. 3's Appendix §A.2.4. The exec function takes a symbolic state σ, program statement $stmt$, and continuation Q. Then, exec symbolically executes $stmt$ using σ to produce a potentially modified state σ', which is passed to the continuation.

Symbolic execution of field assignments first evaluates the right-hand side expression e to the symbolic value t. Any field reads in e are either directly or optimistically verified using σ_1. Then, the resulting state σ_2 must establish write access to $x.f$ in consume, *i.e.* $\sigma_2 \widetilde{\Rightarrow}$ **acc**$(x.f)$. Calling consume also removes the field chunk for **acc**$(x.f)$ from σ_2 (if it is in there) resulting in σ_3. Therefore, the call to produce can safely add a fresh field chunk for **acc**$(x.f)$ alongside $x.f = t$ to σ_3 before it is passed to the continuation Q. Under the hood, run-time checks are collected and passed to Q.

The method call rule evaluates the arguments $\bar{e}$ to symbolic values $\bar{t}$, consumes the method precondition (substituting arguments with $\bar{t}$) while making sure the origin is set properly for check and branch condition insertion, havocs existing assumptions about the variables being assigned to, produces knowledge from the postcondition, and finally continues after resetting the origin to none. An in-depth explanation is in the Appendix section for Chpt. 3 (§A.2), along with the other exec rules and equi-imp definition. Note that while Gradual Viper treats predicates iso-recursively in all other cases, it makes an exception when consuming preconditions at method calls (and loop invariants before entering loops), which can be seen in the `if-then` in the method call rule (Fig. 3.14). If Gradual Viper determines the precondition (invariant) is equi-recursively imprecise, then it will conservatively remove all the heap chunks from both symbolic heaps after the consume. This exception ensures the static verification semantics in Gradual Viper lines up with the equi-recursive, dynamic verification semantics encoded by GVC0 (describe in §3.5) such that Gradual C0 is sound. Interestingly, the gradual verifier defined in Chpt. 2 does not need this special case, because it does not optimize run-time checks with statically available information. Once optimization is introduced, the semantics across the two systems need to be more tightly integrated to ensure soundness.

3.4.8 Valid Gradual Viper programs

Finally, putting everything together, a Gradual Viper program is checked by examining each of its method and predicate definitions to ensure they are well-formed (formally defined in Appendix Fig. A.21). The formal definitions are given in Fig. 3.15, and a more detailed description of the rules is given in Chpt. 3's Appendix §A.2.5. Intuitively, for each method, we define symbolic values for the method arguments, and then create an initial symbolic state by calling the produce function on the method precondition[7]. We then call the exec function on the method body, which symbolically executes the body and ensures that all operations are valid based on that precondition. Finally, we invoke the consume function on the final symbolic state and the postcondition, verifying that the former implies the latter. Throughout these operations a set of run-time checks is built up, which (along with success or failure) is the ultimate result of gradual verification.

[7]Note, produce is part of well-formed.

$$\text{verify}(\texttt{method}\ \ m(\overline{x:T})\ \texttt{returns}\ (\overline{y:T})) = \text{well-formed}(\sigma_0\{\gamma := \sigma_0.\gamma[\overline{x \mapsto \text{fresh}}][\overline{y \mapsto \text{fresh}}]\},\ meth_{pre},\ \text{fresh},\ (\lambda\ \sigma_1.$$
$$\text{well-formed}(\sigma_1\{\ \texttt{isImprecise} := \texttt{false},\ h_? := \emptyset,\ h := \emptyset\},\ meth_{post},$$
$$\text{fresh},\ (\lambda\ _\ .\ \text{success}()))$$
$$\wedge$$
$$\text{exec}(\sigma_1,\ meth_{body},\ (\lambda\ \sigma_2\ .$$
$$\text{consume}(\sigma_2,\ meth_{post},\ (\lambda\ \sigma_3,\ _\ .$$
$$\mathfrak{R} := \mathfrak{R} \cup \sigma_3.\mathcal{R}.\texttt{rcs}\ ;\ \text{success}()))))))$$
$$\text{verify}(\texttt{predicate}\ p(\overline{x:T})) = \text{well-formed}(\sigma_0\{\gamma := \sigma_0.\gamma[\overline{x \mapsto \text{fresh}}]\},\ pred_{body},\ \text{fresh},\ (\lambda_.\text{success}()))$$

Handles imprecision | Handles run-time check generation and collection

Figure 3.15: Rules defining a valid Gradual Viper program

3.5 Dynamic Verification: Encoding Run-time Checks into C0 Source Code

After static verification, Gradual Viper returns a collection of run-time checks $\mathfrak{R}$ that are required for soundness to GVC0. Then, GVC0 creates a C0 program from the run-time checks in $\mathfrak{R}$ and the original C0 program by encoding the checks in C0 source code. The C0 program is sent to the C0 compiler to be compiled, executed, and thus dynamically verified. We chose to encode the run-time checks directly in source code to avoid complexities from augmenting the C0 compiler with support for dynamic verification. Further, since C0 is a simple imperative language, any more expressive language should be able to encode the checks far more easily. That is, we hope this work serves as a guide to the developers of Gradual Viper frontends for other languages on how to implement efficient dynamic verification for gradual verification—especially, when modifying the compiler for their language is difficult. The rest of this section illustrates GVC0's encoding of run-time checks into C0 source code via example. We also highlight design points in the encoding that minimize run-time overhead of the checks during execution.

Now, consider the C0 program in Fig. 3.16 that implements a method for inserting a new node at the end of a list, called `insertLastWrapper`. Note, when passed a non-empty list, `insertLastWrapper` calls `insertLast` from Fig. 1.1 to perform insertion (line 18). Here, `insertLast` is gradually verified with the simpler and fully specified (precise) `acyclic` predicate given on lines 1-4 in Fig. 3.16. For our purposes, we only need to know that `insertLast`'s precondition is `? && acyclic(list) && list != NULL` and its postcondition is `acyclic(\result) && \result != NULL`. The `insertLastWrapper` method is also gradually specified: its precondition is `?` (line 7)—requiring unknown information—and its postcondition is `acyclic(\result)` (line 8)—ensuring the list after insertion is acyclic. Fig. 3.16 also contains run-time checks generated by Gradual Viper for `insertLastWrapper`, as highlighted in blue. The first check (lines 15-17) ensures the list `l` sent to `insertLast` (line 18) is acyclic, and is only required when `l` is non-empty (non-null). The second check (lines 21-23) ensures the list returned from `insertLastWrapper` is acyclic, and is only required when `insertLastWrapper`'s parameter `l` is empty (null). These checks are not executable by the

```
1  /*@ predicate acyclic(Node* l) =
2        l == NULL ? true :
3          acc(l->val) && acc(l->next) &&
4          acyclic(l->next) ;@*/
5
6  Node* insertLastWrapper(Node* l, int val)
7    //@ requires ?;
8    //@ ensures acyclic(\result);
9  {
10   if (l == NULL) {
11     l = alloc(struct Node);
12     l->val = val;
13     l->next = NULL;
14   } else {
15     ({(none, l == NULL, ¬(l == NULL))},
16       (l=insertLast(l,val), acyclic(l),
17         acyclic(l))
18     l = insertLast(l, val);
19   }
20   return l;
21   ({(none, l == NULL, l == NULL)},
22     (none, acyclic(\result),
23       acyclic(\result))
24 }
```

■ Run-time checks from Gradual Viper

Figure 3.16: `insertLastWrapper` program with run-time checks from Gradual Viper

```
1  Node* insertLastWrapper(Node* l,int val,
2    OwnedFields* _ownedFields )
3  {
4    bool _cond_1 = l == NULL;
5    if (l == NULL) {
6      l = alloc(struct Node);
7      l->_id = addStructAcc(_ownedFields,2);
8      l->val = val; l->next = NULL;
9    } else {
10     if (!_cond_1) {acyclic(l, _ownedFields);}
11     OwnedFields* _tempFields =
12       initOwnedFields(_ownedFields->instCntr);
13     sep_acyclic(l, _tempFields);
14     l = insertLast(l,val, _ownedFields );
15   }
16   if (_cond_1) { acyclic(l, _ownedFields); }
17   OwnedFields* _tempFields1 =
18     initOwnedFields(_ownedFields->instCntr);
19   sep_acyclic(l, _tempFields1);
20   return l;
21 }
```

■ Run-time checks from GVC0

Figure 3.17: `insertLastWrapper` program with run-time checks generated by GVC0

C0 compiler; therefore, GVC0 takes the program and checks in Fig. 3.16 and returns the executable program in Fig. 3.17. That is, GVC0 encodes branch conditions (lines 15 and 21), predicates (lines 16-17 and 22-23), accessibility predicates (**acc**`(l->val)` and **acc**`(l->next)` in acyclic's body, lines 1-4), and separating conjunctions (also in acyclic's body) from Gradual Viper into C0 source code. We discuss the aforementioned encodings in sections §3.5.1, §3.5.2, and §3.5.3 respectively. While not in the `insertLastWrapper` example, GVC0 translates checks of simple logical expressions into C0 assertions: *e.g.* `assert(y >= 0);`.

3.5.1 Encoding branch conditions

Run-time checks contain branch conditions that denote the execution path a check is required on. For example, in Fig. 3.16 `acyclic(\result)` should only be checked at lines 21-23 when `l == NULL`, as indicated by the branch condition `(none,l == NULL,l == NULL)`. Therefore, GVC0 first encodes the condition `l == NULL` into C0 code. In general, conditions are encoded as logical expressions in C0 and assigned to fresh boolean variables at the program point where they originated—we call this *versioning*. Then, the boolean variable is used in checks in place of the condition. For example, the `origin` and `location` pair `(none,l == NULL)` tells GVC0 that `l == NULL` must be evaluated at the program point in `insertLastWrapper` containing the

`l == NULL` AST element. As a result, in Fig. 3.17 a boolean variable `_cond_1` is introduced on line 4 to hold the value of `l == NULL`. The condition variable `_cond_1` is then used in the C0 run-time check for `acyclic(\result)` later in the program (line 16). To reduce run-time overhead, `_cond_1` is also used in the check for `acyclic(l)` on line 10, which relies on the same branch point `(none,l == NULL)`. Further, while not demonstrated here, GVC0 supports short-circuit evaluation of conditions on the same execution path to reduce run-time overhead. Finally, note, GVC0 carefully places versioning code after any run-time checks encoded at the program point where the condition originated to ensure this new assignment code (not verified by Gradual Viper) is framed, *i.e.* correct.

3.5.2 Encoding predicates

Now that GVC0 has versioned the branch conditions in Fig. 3.16 into variables, GVC0 can use the variables to develop C0 run-time checks. The Gradual Viper check `({(none,l == NULL,¬(l == NULL))}, (l=insertLast(l,val),acyclic(l),acyclic(l)))` is translated into `if (!_cond_1) {acyclic(l,_ownedFields);}` on line 10 in Fig. 3.17. GVC0 places this C0 check according to the `origin`, `location` pair `((l=insertLast(l, val), acyclic(l))`, which points to the program point just before the call to `insertLast` on line 14. The branch condition becomes the if statement with condition `!_cond_1` (§3.5.1), and `acyclic(l)` is turned into the C0 function call `acyclic(l,_ownedFields)`. The `acyclic` function implements acyclic's predicate body as C0 code: it asserts true for empty lists and recursively verifies accessibility predicates (using `_ownedFields`) for nodes in non-empty lists. That is, predicates are encoded and treated equi-recursively by GVC0. For efficiency, separation of list nodes is encoded separately on lines 11-13. We discuss the dynamic verification of accessibility predicates and the separating conjunction in C0 code next. Finally, a similar C0 check is created for `acyclic(\result)` on lines 16-19.

3.5.3 Encoding accessibility predicates and separating conjunctions

GVC0 implements run-time tracking of owned heap locations in C0 programs to verify accessibility predicates and uses of the separating conjunction.

Encoding owned fields in C0 source code. An owned field is a tuple $(id, field)$ where id is an integer identifying a struct instance (object in C0) and *field* is an integer indexing a field in the struct. The `OwnedFields` struct, which is implemented as a dynamic hash table to improve check performance, contains currently owned fields. That is, hashed object identifiers (id) and then field identifiers (*field*) are used to index into `OwnedFields` where a boolean that determines whether or not the field is currently owned is stored. Since objects are tracked with integers, all struct definitions in a C0 program are modified to contain an additional `_id` field.

Semantics of tracking owned fields (inspired by Chpt. 2, §2.3). At the entry point to a C0 program (*e.g.* `main`), an empty `OwnedFields` struct called `_ownedFields` is allocated and initialized. This is not shown in Fig. 3.17. Then, when a new struct instance is created—such as allocating a new node on line 6 in Fig. 3.17—the `_id` field is initialized with the value of

a global integer `instCntr` that uniquely identifies the instance. The call to library function `addStructAcc` on line 7 performs this functionality and then increments `instCntr`. It also adds all fields in the struct instance (*e.g.* `l->val:(l->_id,0)`, `l->next:(l->_id,1)`, and `l->_id:(l->_id,2)`) to `_ownedFields` and marks them as owned. The only other times `_ownedFields` can change are at method/function calls. Methods, like `insertLast` and `insertLastWrapper`, may add or drop owned fields during their executions. They may also contain run-time checks, such as the one for `acyclic(l)` on line 10, that need owned fields for verification. So, GVC0 adds an additional parameter to the their declarations (*e.g.* line 2, Fig. 3.17) to accept, initialize, and then modify `_ownedFields` in their contexts. A callee's pre- and postcondition controls what owned fields are passed to and from the callee via this new parameter. When a method's precondition is imprecise[8], then any caller will pass all of its owned fields to the method, as on line 14 for the call to `insertLast`. After execution, the callee method returns all of its owned fields to the caller. When a method's precondition is precise, then any caller only passes its owned fields specified by the precondition to the method. If the method's postcondition is imprecise, then after execution the callee method returns all of its owned fields as before; otherwise, only the owned fields specified by the postcondition are returned. Finally, as an optimization, in precisely specified methods (no external—pre- and postconditions—or internal—loop invariants, unfolds, folds, etc.—specifications contain imprecision and no run-time checks are required), GVC0 does not implement any `_ownedFields` tracking. In this case, GVC0 uses the callee's pre- and postcondition to modularly update `_ownedFields` in the caller.

Verifying accessibility predicates and separating conjunctions with owned fields. Now, `_ownedFields` tracking is used to verify accessibility predicates and uses of the separating conjunction. Run-time checks for accessibility predicates are turned into assertions that ensure the presence of their heap location in `_ownedFields`. For example, **`acc`**`(l->val)` looks like `assertAcc(_ownedFields,l->_id,0);` in C0 code, where 0 is the index for `val` in the `Node` struct. The `assertAcc` library function indexes into `_ownedFields` using `l->_id` and 0 and ensures a boolean entry is in there and is `true`; otherwise, `assertAcc` throws an error. Wherever GVC0 must check separation of heap locations (as indicated in run-time checks from Gradual Viper via a flag),—such as for the nodes in list `l` at lines 10-13—it creates (with the library method `initOwnedFields`) an auxiliary data structure `_tempFields` of type `OwnedFields`. We check that heap cells are disjoint by adding them one at a time to `_tempFields` and failing if the cell is already there. GVC0 generates a `sep_X` method for each predicate `X` to actually perform the separation check; and when done, discards `_tempFields`, as its purpose was only to check separation. Similar checks are created for the `acyclic(\result)` check on lines 16-19.

[8]Here, a formula is also imprecise if it contains predicates that expose `?` when fully unrolled—an equi-recursive treatment.

Chapter 4

Performance Evaluation of Gradual C0

The seminal work on gradual typing [42] selectively inserts run-time casts in support of optimistic static checking: for instance, whenever a function application is deemed well-typed only because of imprecision—such as passing an argument of the unknown type to a function that expects an integer—the type-directed cast insertion procedure inserts a run-time check. But if the application is definitely well-typed, no cast is inserted. This approach ensures that a fully-precise program does not incur any overhead related to run-time type checking. While it is tempting to assume that more precision necessarily results in better performance, the reality has been shown to be more subtle: both the nature of the inserted checks (such as higher-order function wrappers) as well as when/how often they are executed is of utmost importance [32, 46], and anticipating the performance impact of precision is challenging [7]. Consequently, we are interested in exploring whether or not gradual verification may experience similar subtleties where precision does not necessarily correlate with better run-time checking performance.

Since we have developed the working gradual verifier Gradual C0 as described in Chpt. 3, we are in a good position to explore the relation between minimizing dynamic check insertion with statically available information and observed run-time performance in gradual verification with Gradual C0. Specifically, we explore the performance characteristics of Gradual C0 for thousands of partial specifications generated from four data structures, as inspired by Takikawa et al. [46]'s work in gradual typing. In particular, we observe how adding or removing individual atomic formulas and `?` within a specification impacts the degree of static and dynamic verification and, as a result, the run-time overhead of the program. Additionally, we compare the run-time performance of Gradual C0 to a fully dynamic approach, as readily available in C0. The aforementioned ideas are captured in the following research questions:

- **RQ1**: As specifications are made more precise, can more verification conditions be eliminated statically?
- **RQ2**: Does gradual verification result in less run-time overhead than a fully dynamic approach?
- **RQ3**: Are there particular types of specification elements that have significant impact in run-time overhead, and can high overhead be avoided?

4.1 Creating Performance Lattices

We define a *complete* specification as being statically verifiable when all `?`s are removed, and then a *partial* specification as a subset of formulas from a complete specification that are joined with `?`. Like Takikawa et al. [46], we model the gradual verification process as a series of steps from an unspecified program to a statically verifiable specification where, at each step, an *element* is added to the current, partial specification. An element is an atomic conjunct (excluding boolean primitives) in any type of method contract, assertion, or loop invariant. We form a lattice of partial specifications by varying which elements of the complete specification are included. We also similarly vary the presence of `?` in formulas that are complete—contain the same elements as their counterparts in the statically verifiable specification—and have related fold and unfold statements in the partial specification. Otherwise, `?` is always added to incomplete formulas. This strategy creates lattices where the bottom entry is an empty specification containing only `?`s and the top entry is a statically verifiable specification. A *path* through a lattice is the set of specifications created by appending n elements or removing `?`s one at a time from the bottom to the top of the lattice. The large array of partial specifications created in each lattice closely approximates the positive specifications supported by the gradual guarantee [47], which are less precise variants of successfully verified programs. For reference, we give a more formal statement of the gradual guarantee:

> *Let p_1 and p_2 be Gradual C0 programs where $p_1 \sqsubseteq p_2$ (*i.e.* the formulas in p_1 are more precise than those in p_2). If p_1 statically verifies, then p_2 statically verifies. Additionally, p_2 must execute at least as far as p_1 executes at run time.*

Now, to illustrate the aforementioned approach, consider the following loop invariant:

```
//@ loop_invariant sortedSeg(list, curr, curr->val) && curr->val <= val;
```

The invariant is made of two elements: the `sortedSeg` predicate instance and the boolean expression `curr->val <= val;`. The lattice generated for a program with this invariant has five unique specifications: four contain a combination of the two elements joined with `?`, and the fifth is the complete invariant above.

4.2 Data Structures

To apply this methodology, we implemented and fully specified four recursive heap data structures with Gradual C0: binary search tree, sorted linked list, composite tree, and AVL tree. We chose these data structures because complete static specifications exist for them in related work and they are interesting use cases for gradual verification. Linked list is implemented with a while loop rather than recursion. Binary search tree is a more complex data structure with a more complex property (BST property) than a linked list and uses recursion. Composite tree implements a structure where modifications do not have to start at the root, but can be applied directly to any node in the tree. Its invariant also applies to any node in the tree. Finally, AVL tree implements the most complex invariant (the balanced property) and data structure with many interdependent functions and predicates related to tree rotations. Each data structure has a test program that contains its implementation and a main function that adds elements to the structure based on a workload parameter ω. We design the test programs to incur as little run-time overhead as

Example	Unverified Complexity	# Specs	Contents of Complete Spec					
			Fold	*Unfold*	*Pre.*	*Post.*	*Pred. Body*	*Loop Inv.*
Binary Search Tree	$O(n\,log(n))$	3473	43	23	0/20/21/24	0/22/6/24	6/6/7/4	0/2/4/2
Linked List	$O(n)$	1745	17	10	8/6/15/5	4/5/6/5	4/3/4/3	4/3/5/2
Composite	$O(n\,log(n))$	2577	28	15	0/10/2/12	0/11/1/12	32/9/17/3	0/3/2/3
AVL	$O(n\,log(n))$	3057	25	14	3/4/5/9	3/6/9/9	25/8/21/3	1/1/2/1

Table 4.1: Description of benchmark examples. For each example, the table shows the complexity of the test program without verification, the number of sampled partial specifications, and the distribution of specification elements for the complete specification. Element counts are formatted as "*Accessibility Predicate/Predicate Instance/Boolean Expression/Imprecision*"

possible outside of structure size and run-time checks. For each example and corresponding test program, Table 4.1 displays the distribution of elements in the complete specification, as well as the run-time complexity of the test program and the number of unique partial specifications generated by our benchmarking tool.

Binary Search Tree (BST). The implementation of the binary search tree is typical; each node contains a value and pointers to left and right nodes. We statically specify memory safety and preservation of the binary search tree property—that is, any node's value is greater than any value in its left subtree and less than any value in its right subtree. The test program creates a root node with value ω and sequentially adds and removes a set of ω values in the range $[0, 2\omega]$. Note that values are removed in the same order they were added.

Linked List. We implement a linked list with insertion similar to the one given in Fig. 1.1. Insertion is statically specified for memory safety as well as preservation of list sortedness. Its test program creates a new list and inserts ω arbitrary elements.

Composite. The composite data structure is a binary tree where each node tracks the size of its subtree—this is verified by its specification along with memory safety. Its test program starts with a root node and builds a tree of size ω by randomly descending from the root until a node without a left or right subtree is reached. A new node is added in the empty position, and then traversal backtracks to the root.

AVL Tree. The implementation of AVL tree with insertion is standard except that the height of the left and right subtrees is stored in each node (instead of the overall height of the tree). This allows us to easily state the AVL balanced property—for every node in the tree the height difference between its left and right children is at most 1—without using functions or ghost variables, which Gradual C0 does not currently support. In addition to specifying the AVL balanced property for insertion, we also specify memory safety. The AVL test program starts with a root node and builds a tree of size ω by inserting randomly valued nodes into the tree using balanced insertion.

4.3 Experimental Setup

With upwards of 100 elements in the specifications for each data structure, it is combinatorially infeasible to fully explore every partial specification. Therefore, unlike Takikawa et al. [46], we proceed by *sampling* a subset of partial specifications in a lattice, rather than executing them all. Specifically, we sample 16 unique paths through the lattice from randomized orderings of specification elements. We chose partial specifications along lattice paths to explore trends in migration from no specifications to complete specifications, which is how we imagine developers may use our tool. We also randomly sampled paths, rather than using a another heuristic for selecting paths, to be prescriptive to users of Gradual C0. We wanted to find and recommend new specification patterns that users should apply or avoid depending on their performance. Every step is executed with three workloads chosen arbitrarily to ensure observable differences in timing. Each timing measurement is the median of 10 iterations. Programs were executed on four physical Intel Core i5-4250U 1.3GHz Cores with 16 GB of RAM.

We introduce two baseline verifiers to compare Gradual C0 against. The *dynamic verifier* transforms every specification into a run-time check and inserts accessibility predicate checks for field dereferences—thereby emulating a fully dynamic verifier. The *framing verifier* only performs the accessibility predicate checks, and therefore represents the minimal dynamic checks that must be performed in a language that checks ownership.[1] We implement the baseline verifiers ourselves using the dynamic semantics in Chpt. 2, §2.3, which checks everything at run time, as a guide.

4.4 Evaluation

Fig. 4.1 shows how the total number of verification conditions (proof obligations) changes as more of each benchmark is specified (green curve). The figure also similarly shows the number of verification conditions that are statically verified as each benchmark is specified (purple curve). From the green curve, we see that even when there are no specifications, there are verification conditions, *e.g.* before a field is accessed, the object reference must be non-null and the field must be owned. Some of these verification conditions can be verified statically as illustrated by the purple curve. As more of a benchmark is specified, there are more verification conditions (green curve); but also, more of these verification conditions are discharged statically and do not have to be checked dynamically (purple curve). Towards the right end of the plots, the two curves converge until they meet when all the verification conditions are discharged statically. As a result, the answer to **RQ1** is *yes*. Note, the number of verification conditions does decrease when enough of the benchmark is specified. This is due to being able to prune symbolic execution paths with new static information.

The plots in Fig. 4.2 display the run-time performance (in red) of dynamically checking the verification conditions from Fig. 4.1. The plots also show how the run-time performance of the dynamic verifier (in green) and framing verifier (in purple) change as more of each benchmark is specified. The green lines show that as more properties are specified, the cost of run-time

[1] These framing checks could fail, for example, if some function lower in the call stack owns data that is accessed by the currently-executing function.

Mean Verification Conditions

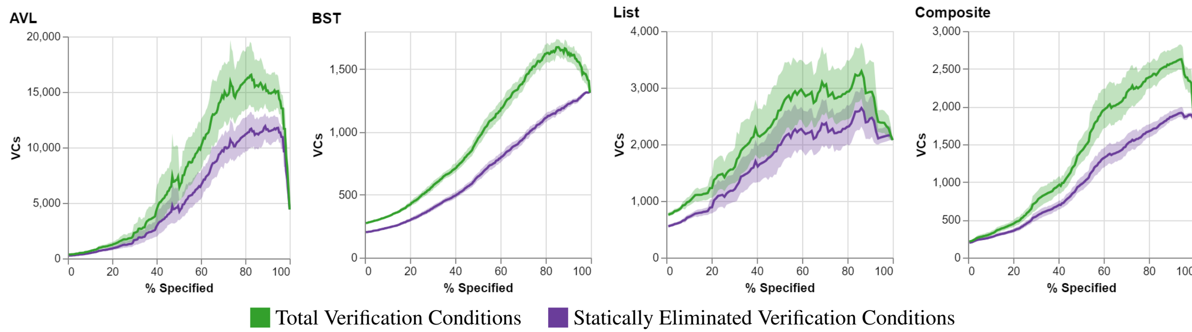


Figure 4.1: For each example, the average quantity of verification conditions and the subset that were eliminated statically at each level of specification completeness across all paths sampled. Shading indicates the standard deviation.

Mean Execution Time Over All Paths

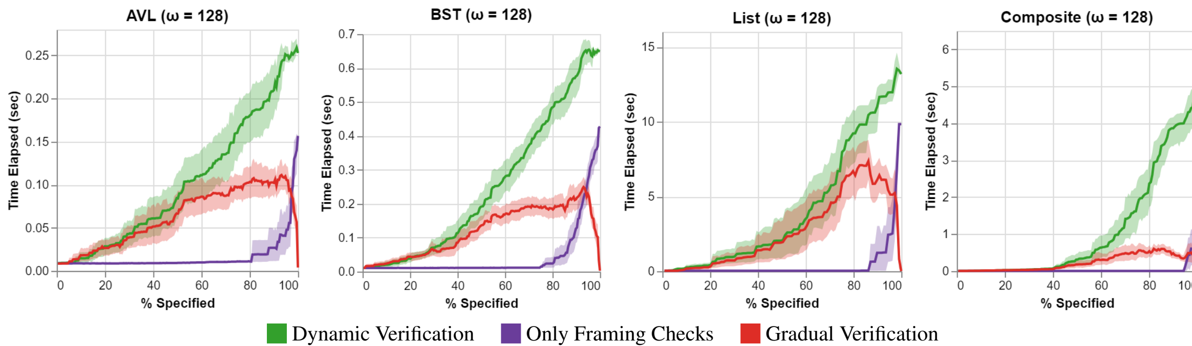


Figure 4.2: The mean time elapsed at each step over the 16 paths sampled. Shading indicates the confidence interval of the mean for each verification type.

verification increases. With Gradual C0, some of these properties can be checked statically; therefore, the run-time cost of gradual verification, shown in red, starts equivalent but eventually ends up significantly lower than the cost of pure run-time verification.

Notably, the purple lines are significantly lower than the red and greens ones until they exhibit a dramatic increase starting at around 80% specified all the way to 100%. Eventually (after about 95% specified), the purple lines end above the red ones (but below the green ones) where running time is orders of magnitude higher than at the start of the incline. The framing verifier (in purple) only checks that heap accesses are safe—*i.e.* they are owned and their receivers are non-null. So unsurprisingly, the dynamic and gradual verifiers, which check more properties like heap separation, nearly always have significantly higher run-time verification overhead than the framing verifier. Eventually, Gradual C0 outperforms the framing verifier when enough properties, including framing, are checked statically.

The dramatic increase in the framing verifier's run-time performance is caused by the owned fields passing strategy employed at method boundaries (described in Chpt. 3, §3.5.3) to verify memory safety at run time. To respect precondition abstractions, only owned fields specified by

Example	ω	% Δt, GV vs. DV				% Steps GV < DV for Paths DV < GV				% Paths GV < DV
		Mean	St. Dev.	Max	Min	Mean	St. Dev.	Max	Min.	
AVL	32	-14.4	29.4	170.0	-88.8	80.6	14.8	95.3	50.8	0.0
	64	-16.1	50.5	256.3	-96.0	84.3	15.5	98.4	55.5	0.0
	128	-11.5	74.0	384.7	-98.6	82.2	16.5	99.5	55.0	0.0
BST	32	-24.7	35.0	144.5	-93.8	74.2	9.0	89.4	60.8	0.0
	64	-23.9	46.8	301.3	-98.5	74.7	9.3	92.2	59.4	0.0
	128	-21.7	55.6	436.1	-99.6	74.7	11.9	96.3	51.6	0.0
Linked List	32	-18.0	27.7	138.9	-95.7	78.8	14.8	96.3	34.9	0.0
	64	-21.0	48.6	389.0	-99.8	85.2	15.1	100.0	37.6	6.3
	128	-20.7	59.7	603.3	-100.0	85.8	15.0	99.1	42.2	0.0
Composite	32	-34.4	40.1	141.7	-99.1	80.3	10.7	94.4	60.9	0.0
	64	-33.1	50.1	258.6	-99.8	80.0	12.2	96.3	50.9	0.0
	128	-30.1	63.9	419.0	-100.0	80.4	13.0	96.3	48.4	0.0

Table 4.2: Summary statistics for the performance of each example over 16 paths at selected workloads (ω), comparing gradual verification (GV) against dynamic verification (DV). The grouped column "% in Δt, GV. vs. DV." displays summary statistics for the percent decrease in time elapsed for each step when using GV versus DV. The column "% Steps GV < DV for Paths DV < GV" shows the distribution of steps that performed best under GV that were part of paths containing steps that performed better under DV. The final column shows the percentage of paths in which every step performed better under GV.

a callee's precondition are passed by the caller to the callee when the precondition is precise. Similarly, when a callee's postcondition is precise, then only the owned fields specified by the postcondition are passed back to the caller. Computing owned fields from precise contracts is costly; and even more-so for contracts containing recursive predicates like in our benchmarks. Further, our benchmarks call such methods frequently during execution. As a result, execution time increases significantly at each path step where one of the aforementioned methods gets a precise pre or postcondition from `?` removal. This, of course, happens more frequently as more of a benchmark is specified. At 100% specified every method contract is precise, and so the owned fields passing strategy is used at every method call and return leading to the highest run-time costs for the framing verifier. In contrast, Gradual C0 checks fully-specified methods completely statically and does not use the owned field passing strategy for calls to these methods. As a result, looking at the red lines, Gradual C0 is not heavily affected by this phenomena—we see slight increases starting at 90% specified but they are significantly less costly. Additionally, once a critical mass of specifications have been written, Gradual C0's run-time verification cost decreases until reaching zero—which is the same as running the raw C0 version of the benchmark. If the spikes around 90% specified are too costly, production gradual verifiers can reduce them by employing more optimal permission passing strategies.

In general, according to the red lines, Gradual C0's performance increases gradually as more proof obligations are specified but are not yet statically verified; and thus, must be checked at run time. When a critical mass of specifications are written that allows more and more of these proof obligations to be proven statically, run-time performance starts to decrease until reaching the spikes around 90% specified caused by owned fields passing. After the spikes, performance decreases to the benchmark's raw baseline. This trend confirms that increasing precision in gradual verification does not always correspond with decreased run-time overhead from dynamic verification.

99th Percentile Changes in Run-time Overhead

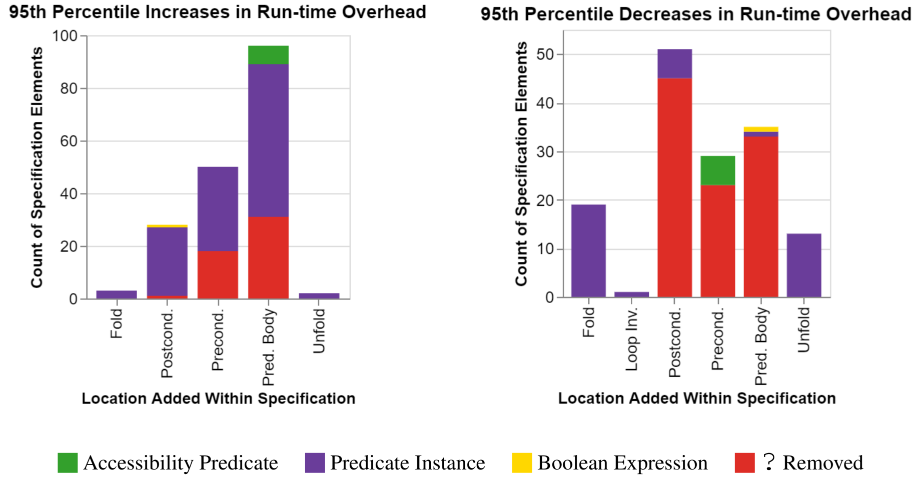


Figure 4.3: The quantity of specification elements, grouped by type and location, that caused the highest ($P_{99\%}$) increases and decreases in time elapsed out of every path sampled

Table 4.2 displays summary statistics for Gradual C0's performance on every sampled partial specification compared to the dynamic verification baseline. Depending on the workload and example, on average Gradual C0 reduces run-time overhead by 11.5-34.4% (Table 4.2, Column 3) compared to the dynamic verifier. Note, the speed-ups are consistent as ω increases: -14.4%, -24.7%, -18.0%, and -34.4% at the lowest ω values compared to -11.5%, -21.7%, -20.7%, and -30.1% at the largest. While Gradual C0 generally improves performance, there are some outliers in the data (Table 4.2, Column 5) where Gradual C0 is slower than dynamic verification by 138.9-603.3%. Fortunately, for lattice paths that produce these poor-performing specifications, gradual verification still outperforms dynamic verification (on average) for 74.2-85.8% (Table 4.2, Column 7) of all steps. Further, these outliers appear under 20% specified where the bookkeeping we insert to track conditionals, which is unoptimized and could be improved, and measurement error are the cause of such outcomes.

Fig. 4.2 displays the average run-time cost across all paths under each of our benchmarks and verifiers. In all the plots, for some early parts of the path the cost of Gradual C0 is comparable to or exceeds the cost of the dynamic verifier, but after 50% completion, static optimization kicks in and Gradual C0 begins to significantly outperform it. Further, Table 4.2 shows that on average Gradual C0 reduces run-time overhead by 11.5-34.4% compared to the dynamic verifier. Therefore, the answer to **RQ2** is *yes*.

Fig. 4.3 captures the impact that different types of specification elements (accessibility predicates, predicates, and boolean expressions) have on Gradual C0's run-time performance when specified in different locations. It also captures the impact removing `?` from a formula has on performance. Elements that when added or `?` that when removed from one step in a lattice path to another increase run-time overhead significantly (in the top 1%) are counted in the left sub-figure, and ones that decrease run-time overhead significantly (top 1%) are counted in the right sub-figure. The count for accessibility predicates is colored in green, predicates in purple, boolean expressions in yellow, and `?` removal in red.

Adding recursive predicates to preconditions, postconditions, and predicate bodies is the most frequent cause (67.6%) of dramatic increases in run-time verification overhead during the specification process. When these predicates are added to preconditions and postconditions they create additional proof obligations for them in callers and callees (respectively) that are frequently checked at run time. Similarly, when they are added to predicate bodies any proof obligations for the enclosing predicate that are checked at run time become far more expensive. Fortunately, folding or unfolding a predicate can decrease run-time cost when doing so discharges such proof obligations statically (as seen in the right sub-figure). Therefore, users of Gradual C0 may consider specifying proofs of recursive predicates in frequently-executed code to significantly reduce checking costs.

Removing `?` from preconditions, postconditions, and predicate bodies when the costly owned fields passing strategy is still required in corresponding methods is the second most frequent cause (27.9%) of increases in Gradual C0's run-time overhead. This corresponds with the spikes at 90% specified in Fig. 4.2 for Gradual C0: removal of `?` in the aforementioned locations leads to precise pre- and postconditions that trigger the use of this costly strategy. Eventually, a critical mass of specifications are written so that when `?`s are removed further this costly strategy is no longer necessary (*i.e.* when callee methods are full statically verified) and so run-time performance improves dramatically—the downward trends seen prior to full static specification in Fig. 4.2. This is reflected in the right sub-figure in Fig. 4.3, where removing `?` from preconditions, postconditions, and predicate bodies is the most frequent cause (68.2%) of significant decreases in run-time overhead. This suggests a strategy for avoiding high checking costs: specify frequently-executed code in critical-mass chunks that are fully statically verifiable, leaving boundaries between statically and dynamically verified code in places that are executed less frequently.

Overall, the answer to **RQ3** is *yes*; we have identified some key contributors to run-time overhead, whose optimization is a promising direction for future work, and we have also identified strategies for minimizing run time overhead in practice.

Finally, all of the partial specification evaluated in our study were successfully verified by Gradual C0. Since they originated from complete and correct specifications on code, we can conclude Gradual C0 adheres to the gradual guarantee for these partial specifications and likely adheres to the gradual guarantee for common use cases of Gradual C0.

4.5 Threats to Validity

Our test programs were executed on multiple devices, each with the same CPU and memory configuration. However, we did not otherwise control for differences in performance between devices. While the test programs we used are of sufficient complexity to demonstrate interesting empirical trends, they are not representative of all software. Further, the baseline we used for dynamic verification is entirely unoptimized as we naively insert a check for each written element of a specification. Finally, due to computational constraints, only a small subset of over 2^{100} possible imprecise specifications were sampled, and we did not use a formal criteria to choose our workload values. Thus, while our results reveal interesting and important trends, more work is needed to validate the robustness of those trends.

4.6 Qualitative Experience with AVL Tree

Notably, it was our experience that the incrementality of gradual verification was very helpful for developing a complete specification of the AVL tree example. In particular, a run-time verification error from a partial specification helped us realize the contract for the `rotateRight` helper function was not general enough. We fully specified `rotateRight` and proved it correct. However, `insert`'s pre- and postconditions were left as `?`, and so static verification could not show us that the contract proved for `rotateRight` was insufficiently general. Nevertheless, we ran the program; gradual verification inserted run-time checks, and the precondition for `rotateRight` failed. This early notification allowed us to identify the problem with the specification and fix it immediately. Otherwise, we would have had to get deep into the static verification of `insert`—a complicated function, 50 lines long, with lots of tricky logic and invariants—before discovering the error, and a lot of verification work built on the faulty specification would have had to be redone. Interestingly, it is conventional wisdom that one of the benefits of static checking is that you get feedback early, when it is easier to correct mistakes. Here, we encountered a scenario where gradual verification had a similar benefit over static verification! We found an error (in a specification) earlier than we would have otherwise, presumably saving time.

Chapter 5

Case Study: Gradual Verification of a C Parser

Our previous evaluation, as described in Chpt. 4, focused on exploring run-time checking performance trends in gradual verification with Gradual C0. But, interestingly, we also ran into a scenario where a run-time verification error from Gradual C0 alerted us to an issue with a user written specification (Chpt. 4, §4.6) earlier than static verification could. Additionally, up until now, we have only explored the benefits of using gradual verification in practice by demonstrating the process and output on smaller pieces of code (*e.g.* Chpt. 1 and Chpt. 4 §4.2 programs are all < 300 lines of code)—including the aforementioned interesting result from specifying AVL tree. So, a natural next step of our work is to investigate how one might use gradual verification in practice on a real codebase and see what patterns and trends emerge. In particular, our precise objective is ***to explore how gradual verification is used to verify real application software that uses recursive heap data structures***—where application software may be a media player, programming language compiler, word processor, etc. We are also interested in uncovering limitations of Gradual C0 and challenges the specific software types pose for gradual verification. That is, we set out to answer the following research questions:

RQ1 What trends or themes occur during the gradual verification process?

RQ2 What types of trade-offs are made during the gradual verification process?

RQ3 Are there instances where static or dynamic feedback is helpful or detrimental or where one is better than the other?

RQ4 How is gradual verification used to find bugs in real-world software?

RQ5 What limitations does Gradual C0 have?

- Engineering limitations?
- User interface limitations? Feedback limitations?
- Inherent limitations of gradual verification?
- Inherent limitations of the underlying static or dynamic verifier technology?

RQ6 What challenges does the type of software being studied pose for gradual verification?

As this is the first work that looks to answer the aforementioned research questions for gradual verification, our study is exploratory in nature. We, therefore, conduct a case study using Gradual C0 to answer our RQs. The selected case is described in §5.1 and the methodology, data, materials, and data analysis in §5.2. We give the results in §5.3 and discuss them in §5.4. Note, all of the materials, software making up our case, and data can be found in a GitHub repository designated for this purpose: https://github.com/gradual-verification/gvc0-cparser-case-study.

5.1 The Case Study: a C Parser

The goal of our case study is to understand how gradual verification can be used to verify application software using recursive heap data structures. A programming language parser is often implemented with recursive algorithms that manipulate lists and trees. Furthermore, the theoretical foundations of parsers are well-understood, but parsers are hard to implement correctly in practice and are error-prone. Therefore, we looked for a parser implementation with the following criteria to study:

- Possible to re-implement in the C0 programming language as Gradual C0 only supports C0 code
- Relies primarily on recursive heap data structures in its main algorithms
- Has multiple modules and over 20 functions, so we can study how gradual verification works in such a setting

We decided on Chibicc's parser for our case. Chibicc (developed by Rui Ueyama) is a 4k lines of code (LoC) C compiler written in C that can compile several real-world programs such as Git, SQLite, and libpng. It contains a tokenizer, preprocessor, parser, and code generator. The tokenizer takes a string as input, turns it into a list of tokens, and sends the list to the preprocessor. The preprocessor expands macro tokens—interpreting preprocessor directives during this process—and sends the resulting token list to the parser. The parser is a recursive descent parser that contructs abstract syntax trees from its given token list. It also adds a type to each AST node. Then, finally, the code generator emits assembly texts for the given AST.

The developer of Chibicc values simplicity and readability of its source code, and so Chibicc's implementation does not include abstractions and clever tricks like higher-order functions, macros, and unions. This makes translating the parser from C to C0, which doesn't support these features, feasible. Chibicc also avoids calling `free`, so Chibicc is structured for garbage collection as required by C0—a garbage collected language. Additionally, Chibicc's parser functions manipulate token lists and abstract syntax trees with loops and recursion. Chibicc's parser uses strings and arrays minimally and outside core functions making it easier to translate the parser into code verifiable by Gradual C0, which does not support such constructs (see §5.1.1 for more details). Chibicc's parser also relies on various modules, such as Types, Tokens, ASTNodes, and HashMaps; and the parser's functions implementing the recursive descent algorithm are plenty (40+) and call each other in intricate ways (directly, indirectly, recursively, or not at all). Between the different modules, plethora of functions, intricate interactions between functions, and manipulation of recursive data structures across such functions, Chibicc should prove challenging to gradually verify.

$id \in IDENT$ *(identifiers)* $n \in NUM$ *(numeric literals)*
$c \in CHAR$ *(char literals)*

<prog> ::= (<typedef> | <funcdef>)∗
<typedef> ::= **typedef** <tp> <declarator> ("," <declarator>) ∗ ";"
<declarator> ::= (<ptrs>)? ("(" id ")" | "(" <declarator> ")" | id) (<func-params>)?
<ptrs> ::= <ptrs> "∗" | "∗"
<func-params> ::= "(" (**void** | <param> ("," <param>) ∗)? ")"
<param> ::= <tp> <declarator>
<funcdef> ::= <tp> <declarator> ";" | <tp> <declarator> "{" <compound-stmt> ∗ "}"
<tp> ::= **void** | **_Bool** | **char** | **int** | **short** | **short int** | **long** | **long int** | **long long**
| **long long int** | **signed** | **signed char** | **signed int** | **signed short**
| **signed short int** | **signed long** | **signed long int** | **signed long long**
| **signed long long int** | **unsigned** | **unsigned char** | **unsigned int**
| **unsigned short** | **unsigned short int** | **unsigned long** | **unsigned long int**
| **unsigned long long** | **unsigned long long int** | **float** | **double** | **long double**
<compound-stmt> ::= <typedef> | <declaration> | <stmt>
<declaration> ::= <tp> <declarator> ("," <declarator>) ∗ ";"
<stmt> ::= **return** <expr>? ";" | **if** "(" <expr> ")" <stmt> (**else** <stmt>)?
| **while** "(" <expr> ")" <stmt> | "{" <compound-stmt> ∗ "}" | <expr>? ";"
<expr> ::= id | <lit> | <expr> <binop> <expr> | <unop> <expr> | "(" <expr> ")"
| id "("<func-args>?")"
<func-args> ::= <expr> ("," <expr>)∗
<binop> ::= "=" | "||" | "&&" | "|" | "^" | "&" | "==" | "!=" | "<=" | ">="
| "<" | ">" | "<<" | ">>" | "+" | "−" | "∗" | "/" | "%"
<unop> ::= "+" | "−" | "∗" | "&" | "!" | "∼"
<lit> ::= n | c | **NULL** | **true** | **false**

Figure 5.1: C Grammar parsed by TinierCP

5.1.1 Translating Chibicc Parser Code into C0 Code

Chibicc's parser is nearly 4k lines of C code: its main functions make up 2,832 LoC and its modules, such as a HashMap, Token, and Type, make up 1,099 LoC.[1] Additionally, while Chibicc's parser implementation is closer to verifiable C0 code than other C parsers written in C, it still uses many features that are not supported by C0 and Gradual C0 as described in Table 5.1. Therefore, to avoid spending too long implementing Chibicc's parser in verifiable C0 code, we implemented only a core subset of Chibicc's parser in verifiable C0 code named TinierCP. TinierCP preserves Chibicc's structure, code patterns, and design choices but parses more simplistic C programs.

The BNF grammar of C programs that can be parsed by TinierCP is found in Fig. 5.1, and an example program following the grammar is given in Fig. 5.2. TinierCP parses programs containing typedefs, function declarations, and function definitions. Typedefs may be defined for a plethora of built-in C types, such as void, bool, char, int, short, long, unsigned, signed, float, dou-

[1]Not all functions implemented in hashmap.c, tokenize.c, and type.c are used in the main parser code, but a significant portion of them are. Furthermore, we do not include LoCs in this calculation from Chibicc's header file, which implements the structs for HashMap, Token, and Type and declares the functions implemented in the .c files. This would add an additional 389 LoCs to the total

C Construct or Feature	Unsupported by C0 or Gradual C0	How Translated into Verifiable C0 code	Code Modified
Switch Statements	C0	Translated into if statements	declspec (function parsing types in TinierCP); add_type (function, adds types to AST nodes in TinierCP)
Enums	C0	Replaced with manual int assignments; Wrapped in a struct and checked or modified with functions when used for tracking Token, Type, and ASTNode kinds	Functions: declspec Modules: Token, Type, and ASTNode
Global Variables	C0	Functions modified to accept global variable, local variable, and scope collections as arguments	All major parser functions (approx. 27 functions); All globals, locals, and scope modifiers (approx. 10 functions)
Static Functions	C0	Removed static tag from functions declared or defined with them	All major parser functions (approx. 27 functions)
Address of (&)	C0	Allocated a pointer to a pointer and added two assignments to track values	All major parser functions (approx. 27 functions)
Implicit Conversions: Bools $\leftrightarrow$ Ints	C0	Expanded into boolean comparisons	More than five parser functions
Break and Continue Statements	C0	Removed with switches; Bools added to replace and track breaks and continues in loops	Functions: parse, declspec, declaration, compound_stmt, func_params, equality, relational, shift, add, mul, and postfix
Strings	Gradual C0	StringList module and helper functions implemented to replace uses of and dependencies on char* arrays; Parser error calls wrapped in library functions	All major parser functions (approx. 27 functions); Approx. 15 helper functions; Obj, ASTNode, Token, and Scope structs
Arrays	Gradual C0	Implemented ScopeMap module and modifiers based on a linked list of key, scope nodes to replace the HashMap module	Functions: new_scope, push_scope, find_func, and find_var (for tracking scopes); Modules: Scope and HashMap
Bitwise Operators	Gradual C0	Refactored declspec's algorithm to rely on regular addition instead of bit operators '<<' and '|'	declspec

Table 5.1: C features used by Chibicc that are unsupported by C0 and Gradual C0, and how they are avoided in the verifiable C0 version of Chibicc's parser

```
1  typedef int t;
2  int f(int p) {
3    int x, y;
4    y = 42;
5    x = y;
6    x = 42 + 239 - y;
7    y = -x;
8    y = x << 42;
9    y = x >> -239;
10   x = ~(y & x);
11   float a, b, c;
12   b = 2.84 * 0.000;
13   c = 2.84 % 0.000;
14   a = b / c;
15   *&x;
16   int *l;
17   *l = &x;
18   int **z;
19   **z = &y;
20   **z;
21   if (p <= 1) {
22     while (p > 1) {
23       _Bool b;
24       _Bool c;
25       _Bool d;
26       b = c == d;
27       b = c <= d;
28       b = c < d;
29       b = c >= d;
30       b = c > d;
31       b = c && d;
32       b = !c;
33       b = c != d;
34     }
35     return 1;
36   } else {
37     return f(p - 1) + f(p - 2);
38   }
39 }
```

Figure 5.2: C program used to test TinierCP

ble, and their various combinations, and pointers to these types. More complicated typedefs like `typedef int int_t, *intp_t;` can also be defined. Functions can be declared or defined with parameters and return types of the aforementioned types, and their bodies may be empty or contain a sequence of typedefs, local variable declarations, and/or program statements. Note that while the grammar in Fig. 5.1 allows functions to take other functions as parameters, in reality TinierCP disallows this. TinierCP parses if statements, while loops, return statements, empty statements, and expression statements (which include variable and pointer assignments). Finally, expressions may contain identifiers, literals, binary ops, unary ops, and function calls. Notably, expressions such as `++i`, `i--`, `i += 1`, etc. do not parse with TinierCP. An example program, which was used to test TinierCP and parses successfully in TinierCP, is given in Fig. 5.2.

Code modifications for C0's limitations. TinierCP follows Chibicc's implementation closely, however TinierCP deviates as described in Table 5.1 due to C0 and Gradual C0's limitations. The declspec function (which parses types) and add_type function (which adds types to AST nodes) each contain a switch statement in Chibicc that is translated into an `if elseif else` statement in TinierCP—C0 does not support switches. Similarly, declspec's enum statement is now a sequence of integer declarations and assignments defined and used manually. Enums are also used in Chibicc to track the kind of a token (*e.g.* punctuation), type (*e.g.* int), or AST node (*e.g.* if statement). In TinierCP, we instead wrap an integer tracking the kind in a structure and define helper functions to read or modify its value:

```
struct TokenKind { int kind; };
TokenKind *new_TK_PUNCT() {
  TokenKind *tk = alloc(struct TokenKind);
  tk->kind = 2;
  return tk;
}
```

```
bool is_PUNCT(Token *t) {
  if (t != NULL && t->kind != NULL)
    return t->kind->kind == 2;
  else
    return false;
}
```

Additionally, global variables are not allowed in C0, but Chibicc uses them to track global variables, local variables, and scopes during parsing. TinierCP instead passes around collections held

in these variables as function arguments resulting in modifications to approximately 37 function definitions and declarations. The `static` tag is not supported by C0, so uses of it in Chibicc were removed from the corresponding functions in TinierCP. C0 also does not support taking the address of pointers, so we performed over 27 of the following code transformations:

```
// In Chibicc
ASTNode *node = logand(&tok, tok);
```

```
// In TinierCP
Token **rst = alloc(Token*);
*rst = tok;
ASTNode *node = logand(rst, tok, scope);
tok = *rst;
```

Chibicc relies on implicit conversions from integers to booleans in over five functions, *e.g.* `if (!ty->name)` where `ty->name` is a pointer. Since C0 does not support such implicit conversions, TinierCP uses the full boolean comparison, *e.g.* `if (ty->name == NULL)`. Finally, `break` and `continue` statements are not allowed in C0, and they are used in 11 parser functions in Chibicc. The breaks in switch statements were removed along with the switch and new booleans were added to replace and track breaks and continues in loops.

Code modifications for Gradual C0's limitations. We also made modifications to Chibicc's parser code to overcome Gradual C0's limitations. Gradual C0 does not yet support strings and arrays even minimally—they will not parse in Gradual C0. Unfortunately, Chibicc's parser contains short string comparisons throughout its code and implements a HashMap for tracking variable scopes with arrays. Worse, Chibicc implements strings as char* arrays! Instead, TinierCP implements strings as char lists with the StringList struct and its modifying functions: new_stringlist, add_char, get_len, and equals. Then, strings needed in comparisons such as `equal(token,"while")` in Chibicc were translated into `equal(token,str_while())` where str_while is defined as:

```
StringList *str_while() {
  StringList *str = new_stringlist();
  add_char(str,'w');
  add_char(str,'h');
  add_char(str,'i');
  add_char(str,'l');
  add_char(str,'e');
  return str;
}
```

The str_while function creates and returns a pointer to a StringList containing "while\0". Also, equal now wraps a call to equals for `token->str`—which is now also a StringList pointer rather than a char* array—and the aforementioned pointer. Fifty-four str_[name] functions were defined to support the comparisons in TinierCP inherited from Chibicc. Approximately 42 functions and four structs were modified from Chibicc's parser to rely on StringLists instead of char* arrays. Unfortunately, StringLists did not work as a replacement for the long strings supplied to error calls in Chibicc's parser, which were translated into `error([string])` calls in C0—error is C0's library function that stops program execution and reports the message in its given string. Therefore, to 1) preserve descriptive error messaging for parser failures in TinierCP and 2) ensure that TinierCP stops execution when bad inputs are given or bad behavior occurs, we wrapped each of these error calls with additional library functions that have empty argument lists and are not verified by Gradual C0. Approximately five helper functions and seven main parser functions were affected by this change; and, 29 wrapper functions were implemented for the

different error messages. To deal with Chibicc's HashMap module using arrays, we replaced it with a new ScopeMap module implementing a linked list of key, scope nodes and related modifiers: new_varscopemap(), varscopemap_get, and varscopemap_put. Chibicc's Scope struct and helper functions new_scope, push_scope, find_func, and find_var were all modified to use ScopeMap instead of HashMap. Lastly, we refactored declspec's algorithm to avoid using bit operators (shift left and bitor) not supported by Gradual C0, which required us to understand declspec's non-trivial algorithm in detail. This process thus took a significant amount of time.

5.1.2 Testing TinierCP

Since Chibicc has been thoroughly tested and debugged over the years, not many bugs if any should exist in the codebase. Therefore, we tested TinierCP to ensure bugs were not introduced in its development process. We manually constructed a token list for the test program given in Fig. 5.2 and had TinierCP parse the list. We printed both the contents of the token list and the AST produced by TinierCP to the console to manually check their correctness. The test program contains a typedef, a function definition, an if statement, a while loop, arithmetic expressions, boolean expressions, and function calls, which covers a large portion of TinierCP's execution paths. We wanted to implement a better testing infrastructure than this, such as one that inputs C files and tokenizes them automatically, but we decided against it. At this point, we already spent too long implementing TinierCP in verifiable C0 code (thanks to the limitations of C0 and Gradual C0), and did not want to spend additional time on testing. Furthermore, our more simplistic infrastructure can be verified by Gradual C0, while one with file I/O and string streams cannot be.

Running our test case through TinierCP first resulted in an infinite loop. We used print statement debugging to locate the cause of the bug, which turned out to be in the function that parses typedefs in TinierCP, called parse_typedef. An assignment, which steps parsing over semicolons in typedefs, was missing at parse_typedef's end leading to an infinite loop in its caller. We added the missing assignment statement fixing the bug. Notably, this bug is not in Chibicc and was introduced during its implementation into verifiable C0. After fixing the infinite loop bug, TinierCP then dereferenced a null pointer for our test case. We again used print statements to locate and debug the error. When translating add_type's switch statement in Chibicc to an `if elseif else` statement in TinierCP, one case was implemented incorrectly the causing error. We changed a couple lines in add_type to fix this bug. Finally, after fixing the aforementioned bugs, the token list for the test program in Fig. 5.2 parsed successfully in TinierCP, and the AST returned by TinierCP for the list was constructed correctly.

5.1.3 Takeaways

After developing TinierCP from Chibicc, testing TinierCP, and debugging it, TinierCP is ready for gradual verification with Gradual C0 and contains 2800 lines of code (including the Obj, Token, Type, ASTNode, StringList, and ScopeMap modules). Despite parsing simpler C programs than Chibicc, TinierCP still maintains the challenges for gradual verification inherent in Chibicc. That is, TinierCP has various structures and modules, plenty of main parser functions (27) that call each other in the same ways as in Chibicc, and TinierCP manipulate lists and trees

with loops and recursion across these functions. Despite Chibicc being C0 friendly compared to other parser implementations, translating select Chibicc code into C0 code involved addressing a number of limitations, such as C0's lack of support for switches, enums, global variables, address of, implicit conversions, and breaks/continues. Related changes dealing with these limitations were busy-work; but, they affected a large number of modules and functions slowing down the development of TinierCP significantly. Similarly, while Chibicc limits its use of strings, arrays, and bitwise operators compared to other parser implementations, Chibicc still used these unsupported Gradual C0 features. Circumventing these features in Chibicc resulted in more complex and time consuming solutions than the ones handling C0's limitations. Furthermore, one would not implement such solutions in practical code, *e.g.* strings being implemented as lists of chars and variable scopes being implemented as maps backed by lists. Ideally, Gradual C0 should support these features in some limited way as soon as possible for usability. Finally, while we found and fixed bugs in TinierCP with our simpler testing infrastructure, more serious bugs like the infinite loop and null pointer ones were difficult to locate and debug thanks to TinierCP's many intricate function calls and heavy use of recursion and loops. Additionally, testing executes these bugs causing TinierCP to hang indefinitely and segmentation fault; and, testing only ensures these bugs are found in covered execution paths—TinierCP has many edge cases. Therefore, these deeper bugs are good targets for gradual verification, which can prove the absence of them statically or guard against them at run time across all execution paths. Frequent verification feedback from gradual verification may also make it easier to find and debug such bugs.

5.2 Methodology

After preparing our case TinierCP (as described in §5.1.1 and §5.1.2), the author of this dissertation gradually verified TinierCP with Gradual C0 and recorded her experience during the process. The author has the deepest understanding of the inner workings of Gradual C0 and how it works to gradually verify code. She can also readily describe how different specifications impact static and dynamic performance with limited feedback from the tool. As a result, she can overcome serious limitations of the tool or new bugs witnessed in Gradual C0 during the verification process (Gradual C0 has been shown to be sufficiently robust for a performance study in Chpt. 4, but Gradual C0 is still in an alpha state). She can also provide an expert user's perspective on the tool and gradual verification process, which we intend to capture in this first case study of Gradual C0. After applying improvements to Gradual C0, based on the results of this study, we hope the tool can then be used by external experts or novices in subsequent studies to confirm, deny, or add to the conclusions of this study.

The author was given one week to gradually verify that the loops in TinierCP's main functions terminate with Gradual C0 and record her experience through journal entries. We also collected various data from Gradual C0 as she used the tool. We describe the data collected, journal entries, and materials in more detail in §5.2.1 and our data analysis procedures in §5.2.2.

Q #	Google Form Questions	Type of Question
Q1-Q2	Date and time of session start.	Calendar/Time
Q3-Q4	Date and time of session end.	Calendar/Time
Q5	What have you been working on? Include: • Functions or modules modified • Specifications written • Modules/functions specified • Types of properties proven • Types of specifications written • Proof obligations written	Long answer
Q6	How does what you have been working on relate to other journal entries and the overall verification process? Is there a reason this is the next set of steps?	Long answer
Q7	Anything of note during the process? • Any trade-offs made? • Any verifier feedback utilized? • Any bugs in specs or code? • Any tool limitations? How were they overcome if applicable? Suggested improvements? • Positives/negatives about the tool/gradual verification? • Gradual C0's reactions to changes in code or specs? • Conversion issues between C and C0?	Long answer
Q8	Theme of this entry.	Short answer
Q9	Reminder to save console output and document where the output is.	Short answer
Q10	What verifier data (from Gradual C0) corresponds to this entry?	Long answer

Figure 5.3: Questions asked in the Google Form used to capture qualitative data from the author. All questions were required

5.2.1 Data & Materials

The author recorded her experience, thought process, and comments about the gradual verification process in a Google Form designed to prompt for this data. She kept the form open during her verification sessions and filled it out when she started or ended a session or finished a portion of work she wanted to document (covering anywhere from one hour to eight hours of work). She also used the form to write notes during her sessions in a think-a-loud manner. A static version of the Google Form can be found in our data repository (https://github.com/gradual-verification/gvc0-cparser-case-study/tree/main/materials) and the questions asked in the form are given in Fig. 5.3. The questions capture the dates and times starting and ending the journal entry (Q1-Q4), what the author has been working on (*e.g.* code modified, specs written, proof obligations written, etc.) (Q5), how that relates to previous entries and why (Q6), anything of note that the author encountered (*e.g.* trade-offs made, feedback utilized, limitations of Gradual C0, bugs found, etc.) (Q7), the theme of the entry (Q8), and locations where Gradual C0 produced data were saved (Q9-Q10).

We implemented a special mode with corresponding command-line option (`-c`) in Gradual C0 for the author to use when verifying TinierCP with Gradual C0. The mode runs Gradual C0 on a given case study (TinierCP) and collects data in a folder named for the case and date/time of the run. The data collected per run are:

- Program file(s) making up the case
- Intermediate files produced during verification by Gradual C0: the case in IR form (.ir.c0), in the Gradual Viper language (.vpr), with run-time checks added (.verified.c0), and with run-time checks added and compiled by the C0 compiler (.verified.c0.h, .verified.c0.c, .bin)
- Various performance measures[2] saved to a log_data.txt file
    - Time it takes Gradual C0 to parse the case into the Gradual Viper language (nsec)
    - Time it takes to statically verify the Gradual Viper program (nsec)
    - Time it takes to add run-time checks to the case for dynamic verification (nsec)
    - Time it takes for the C0 compiler to compile the case with run-time checks (nsec)
    - Time it takes to execute/run the case with run-time checks, *i.e.* the time it take to dynamically verify the case (nsec)
    - Time it takes to verify the case in Gradual C0 from end-to-end, *i.e.* the performance measure of Gradual C0's entire pipeline for the case (msec)
- The number of verification conditions statically verified by Gradual C0, and the total number of verification conditions verified by Gradual C0—also saved to the log_data.txt file
- Time stamp of the run saved to the log_data.txt file

Gradual C0 also outputs the verification result, time elapsed for the run, and a specification summary (recording the number of different types of specification constructs specified on the case) to the console. Gradual C0 does not save this output automatically; and so, the author did this manually. She piped the output to a file, which she saved in the data folder corresponding to the run. If there are any failures from Gradual C0—verification failures or failures from undefined behavior in Gradual C0—then only data that can be collected up until the failure point is recorded. For example, if Gradual C0 produces a static verification error for the case and its specification, then only the original case files, its .ir.c0 files, and its .vpr files are saved. The console will only output the error details. We gave a document to the author that contains step-by-step instructions on how to collect the data we are looking for with the aforementioned materials. This ensures data is recorded in a consistent manner across verification sessions. This document is hosted on github: https://github.com/gradual-verification/gvc0-cparser-case-study/tree/main/materials.

The author completed the study in a Linux VM with the current versions of Gradual C0 and Visual Studio installed. The Visual Studio editor was set up to run Gradual C0 in the terminal on TinierCP, had TinierCP loaded as a project in the file explorer, and had the C0 language highlighting extension installed. The VM is running Ubuntu 22.04.3 LTS, has a 13th Gen Intel(R) Core(TM) i7-1365U 1.80 GHz processor with 10 cores, and has 16GB of RAM and 200GB of storage.

5.2.2 Data Analysis Procedure

The author submitted 15 journal entries via the Google Form described in §5.2.1 and each entry contained anywhere from one to eight paragraphs of text per long answer for Q5-Q7. This text thoroughly documents the author's experience, thought process, and comments about using Gradual C0 to gradually verify TinierCP, and so was the focus of our data analysis. The

[2]Note, the performance measures are for a singular run of Gradual C0 on the case.

answers to Q1-Q4 and Q9-Q10 were used for book keeping; and, we ignored the answers to Q8—which asked for the “theme of this entry"—because they devolved into short summaries of Q5’s answers.

We qualitatively coded the author’s answers to Q5-Q7 in three phases using Google Sheets. The raw responses, codes, and outputs from each phase can be found in this case study’s data repository: https://github.com/gradual-verification/gvc0-cparser-case-study/tree/main/analyzed-data. In the first phase, we started with an initial set of codes defined from our research questions (see introduction to Chpt. 5) and assigned each code a distinct color (*e.g.* “static feedback helpful" is a code paired with the dark blue color). Then, we colored phrases, sentences, or paragraphs in each answer according to the code that applies to the text. We also added a few new codes and color pairs as needed during this process. In the second phase, we grouped the color coded text from phase one by similarly coded excerpts (the color coding from phase one facilitated this process). Then, we developed higher-level themes and patterns emerging from the groupings, *e.g.* “run-time assertion failures due to optimistic branching are confusing". As more excerpts were added to a grouping or new groupings were formed, new themes and patterns emerged and existing ones were refined, *e.g.* “run-time assertion failures due to optimistic branching are confusing" was added to a larger grouping named “static verification with branching is hard and Gradual C0 doesn’t help but rather makes it more confusing." Once all excerpts from phase one were grouped under a higher-level theme and all themes were refined sufficiently, we organized the themes and refined them further into the narrative presented in §5.3 in phase three.

5.3 Results

The author gradually verified loop termination for parse, declspec, and parse_typedef—which parses entire programs, types, and typedefs respectively—successfully using Gradual C0. She partially specified seven parser functions and one helper function during this process. Fig. 5.4 summarizes the contents of specifications written by the author (denoted as the final partial specification). Since Gradual C0 defaults missing specifications to `?`, Fig. 5.4 also provides a summary of the specifications on TinierCP before the author wrote any specifications (denoted as default) as a baseline. In the end, the author wrote 120 lines of specification code, 243 additional lines of program code (primarily for inductive lemmas), and modified 35 lines of program code for the gradual verification of TinierCP. She specified 59 folds and unfolds, 21 accessibility predicates, 45 predicates, and 29 booleans expressions. Nearly all function contracts (229/234) and loop invariants (41/41) still contain `?` as the author focused only on verifying loop termination for three loops—complete static specifications for any contract or loop invariant would require many more specifications for orthogonal properties. In contrast, predicate bodies, which were written specifically for expressing and verifying loop termination, are nearly all (5/6) precise.

Even without specifications in the default case, Gradual C0 still verifies that heap accesses in TinierCP are safe (*i.e.* receivers are non-null and each access contains an owned heap location as determined by the *implicit dynamic frames* logic [44]) using static verification where possible and dynamic checking otherwise. Gradual C0 statically verifies 49.2% of heap accesses in TinierCP in 1 minute and 1 second and the rest are guarded with run-time checks. It takes Gradual C0 0.16 seconds to run TinierCP with these checks on its test case (from §5.1.2). After the specification

Partial Spec	LoSC:LoC	Contents of Partial Spec					
		Fold	*Unfold*	*Pre.*	*Post.*	*Pred. Body*	*Loop Inv.*
Default	0:2800	0	0	0/0/0/229/229	0/0/0/229/229	0/0/0/0/0	0/0/0/41/41
Final	120:3043	34	25	5/16/7/231/234	6/19/8/229/234	9/3/4/1/6	1/7/10/41/41

Figure 5.4: Partial specification summaries for select partial specifications of TinierCP. For each partial specification, the table gives the ratio of lines of specification code to lines of program code and the distribution of specification elements for the partial specification. Element counts are formatted as "*Accessibility Predicate/Predicate Instance/Boolean Expression/Imprecision/Total*"

Partial Spec	% Statically Verified	Static Verification Time	Dynamic Verification Time
Default	49.20	1 min 1 sec	0.16 sec
Final	71.23	3 min 7 sec	0.19 sec

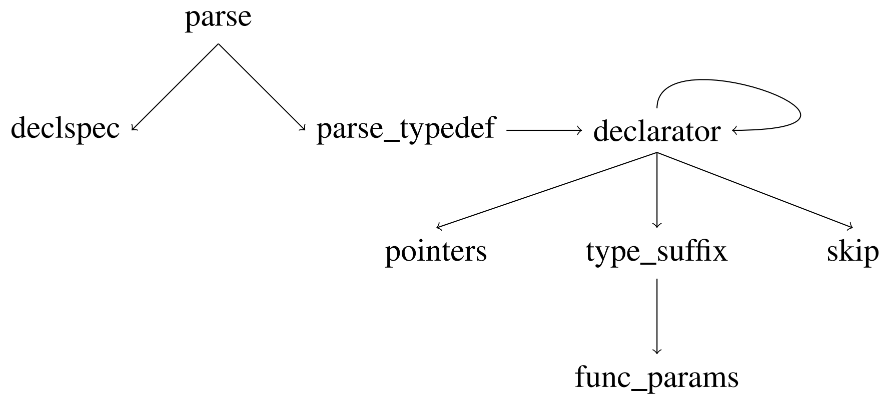
Figure 5.5: Gradual verifier results for select partials specifications of TinierCP. For each partial specification, the table gives the percentage of verification conditions statically verified, the time Gradual C0 spent statically verifying the partial specification in minutes and seconds, and the time Gradual C0 spent dynamically verifying the partial specification in seconds for the test case from §5.1.2

process (in the final case), Gradual C0 statically verifies 71% of heap accesses and new proof obligations for termination in 3 minutes and 7 seconds. The rest are turned into run-time checks and Gradual C0 takes 0.19 seconds to execute TinierCP with these checks on TinierCP's test case. The aforementioned numbers are summarized in Fig. 5.5.

The rest of this section presents themes and supporting evidence from qualitative coding the author's journal entries (§5.2.2). In §5.3.1 we see how the author used a top-down workflow to verify loop termination in TinierCP and how Gradual C0 encourages such a workflow in §5.3.2. In §5.3.3 we discuss how this workflow and feedback from Gradual C0 allowed the author to ensure TinierCP's parser loops terminate and find related bugs in TinierCP's implementation. The author uncovered limitations of Gradual C0 in supporting such a useful workflow, which we discuss in §5.3.4 along with solutions. We end this section by pointing out additional issues the author had with Gradual C0 in §5.3.5 and propose solutions where possible.

5.3.1 The author verified loop termination in TinierCP in a top-down manner focusing only on relevant functions and properties

The author gradually verified termination of parse, declspec, and parse_typedef's loops in a top-down manner. She partially specified the eight functions in Fig. 5.6's call graph during the process. She first wrote fully precise recursive predicates `tokenList` and `tokenListSeg` specifying that a given token list or token list segment is acyclic. Then, she used those predicates to partially specify parse—the entry function into TinierCP—to ensure its loop terminates. In particular, she specified parse's precondition as `? && tokenList(tok)`—which ensures parse only accepts acyclic lists—and parse's loop invariant as `? && tokenListSeg(gv_tok,tok) &&`



parse — *entry function into TinierCP*	pointers — *parses pointers*
declspec — *parses types*	type_suffix — *parses function parameters*
parse_typedef — *parses typedefs*	func_params — *parses function parameters*
declarator — *parses declarators*	skip — *skips a given token in a token list*

Figure 5.6: Graph of function calls for TinierCP functions partially specified by the author

`tokenListSeg(tok,NULL) && (gv_beforeloop == true ? true : gv_tok != tok)` where `gv_tok` and `gv_beforeloop` are program variables introduced for verification and `gv_tok` is always the current token before the loop executes and `gv_beforeloop` is true when execution is before the loop and false otherwise. That is, parse's loop must preserve the acyclic shape of parse's token list and progress the list forward at least one token on every iteration.

The author chose to partially specify declspec next, because it "is always executed in [parse's loop making it] an important function for determining parse's termination." It also has a loop of its own that must terminate when parsing the token list passed to declspec from parse. She used what she learned from specifying parse's loop to similarly specify declspec's contract and loop to preserve acyclicity of its given token list and ensure execution progresses forward in the list. While specifying the loop, the author realized with Gradual C0's help that parse's loop will incorrectly loop forever. Gradual C0 alerted her to a contradiction between declspec's loop invariant and postcondition. After investigating, the author found that the loop invariant and declspec were correct while the postcondition was incorrect and too strong for declspec's code. But, declspec's postcondition was precisely what parse's loop needed to avoid iterating indefinitely leading the author to investigate parse's loop and find an infinite loop bug (see §5.3.3 for more information). The author, then, fixed both parse's loop code and declspec's postcondition; and as a result, "went back to statically verifying more of declspec's loop ... to make sure [there are not] any [more] issues." She, similarly, "weakened [declspec's] loop invariant ... based on [a] realization" made during this process. Once the author was confident in the correctness of declspec and its loop, she moved on to partially specifying parse_typedef, which is called optionally in parse's loop and contains its own loop that must terminate for token lists. She specified `parse_typedef` and its dependencies (declarator, type_suffix, pointers, func_params, and skip) in a similar manner to `parse` and `declspec`. That is, the author:

1. Specified each function's contract as motivated by the needs of its callers. For example,

she wrote “The main determiner of whether the loop in parse_typedef progresses or not is the declarator function, which ideally should always progress the token list if it returns a type with non-null name ... so, I set out to specify and verify declarator for this property in its contract"

2. Specified loop invariants for termination where necessary (*e.g.* parse_typedef)
3. Added additional specifications in functions where the author wanted to increase her confidence in the correctness of the code and her specifications (*e.g.* the execution path in declarator that contains recursive calls to declarator)
4. Refined her specifications in the enclosing function and its callers based on new information gained during the specification and verification process (*e.g.* clauses in declarator’s postcondition were modified slightly after partially verifying its callee skip’s contract)

That is, the author gradually specified and verified TinierCP’s parser loops in a top-down fashion and refined her specifications in a bottom-up fashion as her understanding of the specifications and code improved from the process and feedback from Gradual C0.

5.3.2 Gradual C0 makes selective and top-down verification workflows possible

Gradual C0 is implemented in adherence with the gradual guarantee for gradual verification systems (Chpt. 2, §2.4), which says that Gradual C0 will not flag static or dynamic errors for specifications that are correct but imprecise. Therefore, users of Gradual C0 can specify only the properties and components of code that they care about and get relevant verification feedback—errors caused by missing specifications are suppressed and errors caused by inconsistencies between specifications and code are highlighted. This functionality allowed the author to specify only relevant functions for loop termination supporting a selective, top-down verification workflow.

The author avoided writing ownership specifications to verify heap accesses—an orthogonal property to loop termination—with Gradual C0 and still received useful verification feedback. Fig. 5.6 gives the call graph of parse, declspec, and parse_typedef and their dependencies that the author determined were important for ensuring the correctness of their loops. However, in reality the eight functions in Fig. 5.6 call many more (approx. 39) functions that are not of interest. For example, six of the eight functions (excluding parse and parse_typedef) call equal at least once (declspec calls equal many times), which tests whether or not a given token’s StringList element is equal to another StringList. These calls are also often paired with a call to a str_[name] function, which creates and returns a StringList containing “name". Similarly, parse and parse_typedef call Scope and VarAttr modifiers to manage variable scopes. None of the aforementioned functions affect loop termination for parse, declspec, and parse_typedef as specified by the author.

Unfortunately, in a static verifier (such as Viper [33] or VeriFast [22]), all the functions in Fig. 5.6 and their 39 dependencies need specified for ownership to verify heap accesses in them, otherwise the verifier will only report errors for insufficient permissions to access heap loca-

tions. Worse, is that TinierCP's other 182 functions are also dependencies of the 39 functions and nearly all of them access heap locations as well. So, the other 182 functions also need specified for ownership or even more errors for insufficient permissions to access heap locations will be generated. Applying Viper to TinierCP without any specifications results in 69 such errors across 229 functions and are suppressing many more such errors. Instead, the author used `?` and selectively specified only the seven functions of interest and only for loop termination. She let `?` represent the missing ownership specifications in places where she wrote partial specifications for loop termination, such as on parse's loop and declspec's contract. In functions she didn't specify, she relied on Gradual C0 defaulting missing specifications to `?` to represent ownership specifications. As a result, Gradual C0 verified heap accesses statically where possible and guarded the rest with run-time checks, suppressing static errors for lack of permission access caused by missing ownership specifications. Gradual C0 did not report any permission access violations by TinierCP.

Furthermore, Gradual C0's suppression of errors from missing ownership specifications—and missing auxiliary specifications for loop termination such as folds and unfolds—allowed the author to receive relevant feedback on the correctness of TinierCP with respect to loop termination. We will see next how this feedback gave the author confidence in the correctness of TinierCP's code and her specifications and helped her find a bug in TinierCP's code.

5.3.3 Selective and top-down verification workflows backed by Gradual C0 are useful for assuring loop termination in TinierCP

Gradual C0's suppression of verification errors from missing specifications encouraged the author to verify TinierCP early and often. Thanks to the gradual guarantee, Gradual C0 suppresses verification errors caused by missing specifications; and thus, highlights errors caused by inconsistencies between partial specifications and code. As a result, the author verified her specifications early and often throughout the specification process. She ran Gradual C0 49 times, including after writing each of parse's contract, parse's loop invariant, declspec's contract, declspec's loop invariant, extra specifications in declspec's loop body, parse_typedef's contract and loop invariant, declarator's contract, type_suffix and func_param's contracts, skip's contract, pointers's contract, and extra specifications in declspec's recursive execution path. The author noted that with static verifiers she "probably would go longer without checking these things [knowing she wouldn't] get any verification success or failure feedback until [she has] a more complete spec." This paid off as she found, debugged, and fixed a few different issues in her specifications and a bug in TinierCP's code.

Early and often reporting of relevant static and dynamic verification errors helped the author discover bugs in specifications and TinierCP's code earlier than static verification alone. The author first specified parse's precondition as `tokenList(tok)` and postcondition as `?`, but Gradual C0 reported a run-time error for a failed "TokenKind.kind" field access check to her for this specification. Recall (from §5.1.1), tokens each contain a TokenKind pointer `kind` that contains an integer field `kind`, so the dereference `tok->kind->kind` is possible. It turns out, the `is_EOF(tok)` function called in `parse` contains such a dereference, which is the cause

of the error. Unfortunately, while parse's precondition specifies ownership of all the fields in each token in the list starting with `tok`, the precondition does not specify ownership of the fields in each token's fields. That is, `tokenList(tok)` specifies ownership of `tok->kind` but not `tok->kind->kind`. Gradual C0's run-time system only passes the owned heap locations specified by `tokenList(tok)` (since this precondition is precise) to parse and then to is_EOF(tok) (since is_EOF's precondition is `?`) to check `tok->kind->kind`. So, ownership of `tok->kind->kind` is missing and the access check fails. That is, parse's precondition is too weak and should additionally specify ownership of `tok->kind->kind`. This may be simple, but unfortunately other functions called transitively through parse require ownership of all heap locations accessible through a `tok`. Specifying this is unweildy and not relevant for loop termination, so the author appended `?` to parse's precondition and Gradual C0 successfully verified TinierCP with just parse's contract. Had the author relied on static verification alone, she noted that she would have specified a lot more of parse before verifying her specifications, but Gradual C0's flexibility encouraged her to check them earlier. She also would have needed to specify far more of parse—like its loop—and its callees before a static verifier would alert her to the issue.

Similarly, Gradual C0's optimistic static verifier, which can detect logical inconsistencies even when specifications are incomplete (*e.g.* `? &&` $x > 0 \not\Rrightarrow$ `? &&` $x < 0$ since $x > 0$ contradicts $x < 0$, and so Gradual C0 produces an error here), helped the author discover that her postcondition for declspec was too strong for its implementation. Initially, she partially specified declspec's postcondition as motivated by parse's needs for assuring its loop terminates. That is, declspec's postcondition ensures declspec returns a token list that is acyclic and starts at least one token later in the list passed to declspec. The author also partially specified declspec's own loop for termination ensuring the loop preserves acyclicity of its token list (the one passed to declspec) and the loop's body moves forward in the list. Unfortunately, Gradual C0 reported a static failure for the aforementioned specifications stating the postcondition of declspec might not hold because it cannot prove the `gv_tok != *rest` clause true, which guarantees the token list passed to declspec is progressed. Since Gradual C0 verified declspec's contract successfully before declspec's loop invariant was added and the loop invariant still contains `?`, the author strongly suspected and confirmed the error was caused by a contradiction between the two specifications. It turns out, when the loop never executes, declspec's implementation never progresses its given token list and the loop invariant correctly specifies `gv_tok == tok` in this case. Since `tok` is assigned to `*rest` and `rest` is returned immediately after the loop, `gv_tok != *rest` does not hold for this execution path. So, declspec's postcondition is too strong for its implementation. In a static verifier, the author would need to specify declspec's 170 LoC loop body with over 20 branches in full before it would report the same inconsistency error as Gradual C0.

Recall, declspec's initial postcondition was written precisely to ensure parse's loop terminates. Any weaker postcondition means parse's loop will loop indefinitely for some execution paths—*e.g.* the one where declspec's loop body never executes, which can happen for badly written programs like `intt n;`. So, the author compared TinierCP's parse loop with Chibicc's parse loop and realized the catch all case for un-parsable, badly constructed code is contained in a function that was removed in TinierCP—because the function parses constructs not supported by TinierCP. She fixed this bug by adding a catch all case directly to parse's loop and weakening her postcondition of declspec to match its implementation. Gradual C0 successfully verified this version of TinierCP without error.

Dynamic verification is limited by execution coverage, but selective applications of static verification with Gradual C0 can cover for this weakness. The author discovered the bug in parse's loop using Gradual C0's static verifier by specifying more of TinierCP—declspec's contract and loop invariant. However, she could have discovered the issue earlier—after specifying parse's loop invariant—by using Gradual C0's dynamic verifier on an additional test case containing a badly written C program, such as `intt n;`. For the badly written test case, the loop gets stuck parsing the first bad token and repeatedly loops indefinitely trying to parse it. Fortunately, Gradual C0 checks the loop invariant on every iteration of the loop at run time (since it is not discharged statically), which fails at the first occurrence of the loop getting stuck. Gradual C0 stops execution immediately and notifies the user of the failure. While a simple test case could have uncovered this issue, the eight functions of interest to the author contain a plethora of execution paths, *e.g.* declspec's loop branches over 20 times, which are hard to cover with test cases. Gradual C0's dynamic verifier only alerts users to errors in executed paths, so issues may exist in paths not covered by the author's test case.

To overcome this weakness, the author used Gradual C0 to statically verify properties of interest for execution paths in TinierCP that she deemed as questionable and not covered by TinierCP's existing test case. This strategy was prompted by her success in finding parse's loop bug with selective specifications on declspec and static feedback from Gradual C0. In particular, the author spent a lot of time trying to make sure edge case "`counter > 0`" execution paths in declspec's loop proved `gv_tok != tok` true, which ensures the loop moves forward in its token list. She also "[made] sure the recursive branch of declarator was solid" by incrementally specifying this execution path with Gradual C0 until she was confident in its correctness (*i.e.* when enough important proof obligations are discharged statically by Gradual C0). During this process, she discovered that her initial loop invariant for declspec was too strong for `counter > 0`'s execution paths and adjusted the specification correspondingly. She also noted that "specifying and thinking about how type_suffix contributes to what [she wanted] to prove about declarator helped [her] refine [her] declarator spec a bit ([which] was rougher at first)."

Takeaways. Gradual C0's ability to run-time check proof obligations that cannot be discharged statically when specifications are missing (rather than produce static verification errors) allows users to get early and frequent feedback on the consistency of their code and specifications. The author leveraged this feedback to find, debug, and fix bugs in specifications and code during selective and top-down specification efforts. Unfortunately, dynamic feedback is limited by path coverage, so bad specifications and code may go undetected along paths that are not executed at run time. Users may overcome this weakness (like the author did) by using Gradual C0 to statically verify properties of interest along paths that are questionable and not covered by run-time checking.

```
1  Token *skip(Token *tok, StringList *op)
2  //@requires ? && tokenListSeg(tok,NULL);
3  //@ensures ?;
4  //@ensures tokenListSeg(tok,\result);
5  //@ensures tokenListSeg(\result,NULL);
6  //@ensures tok != \result;
7  {
8    if (!equal(tok, op)) { ... }
9    //@ unfold tokenListSeg(tok,NULL);
10   Token *tokres = tok->next;
11   //@ FOLDS & UNFOLDS
12   return tokres;
13 }
14
```

Figure 5.7: skip code snippet

```
1  while (!break_loop && is_typename(tok, scope))
2  //@loop_inv ? && ...  ;
3  //@loop_inv tokenListSeg(tok,NULL);
4  //@loop_inv ...  ; {
5    ...
6    if (equal(tok, str_typedef()) ||
7        equal(tok, str_static())  ||
8        equal(tok, str_extern())  ||
9        equal(tok, str_inline())) {
10   ...
11     //@ unfold tokenListSeg(tok,NULL);
12     tok = tok->next;
13     //@ FOLDS & UNFOLDS
14   ...
15 }
```

Figure 5.8: declspec code snippet

Figure 5.9: skip and declspec code snippets from TinierCP with the author's specifications

5.3.4 Gradual C0 needs further improvements to support selective and top-down verification workflows

As we've seen, Gradual C0's adherence to the gradual guarantee, which leads to lots of optimism in static verification, is useful for debugging both specifications and code and assuring TinierCP's loops terminate. However, at times the author found such optimism hindering her workflow.

Gradual C0's optimism at function calls made it difficult for the author to reduce run-time checking with more specifications. The author focused primarily on bug finding in code and specifications, but she also occasionally tried to reduce run-time checking in Gradual C0 with additional specifications. Unfortunately, the author gave up on doing this when she realized Gradual C0's optimism around irrelevant function calls made reducing run-time checking difficult and time consuming. Consider, Fig. 5.7 and Fig. 5.8, which contain code snippets from skip and declspec (respectively) with the author's specifications. The author wrote folds and unfolds before and after the variable assignment on line 10 to get Gradual C0 to statically verify skip's postcondition in full (lines 3-6). However, these specifications require `tokenListSeg(tok,NULL)` to be available for the unfold on line 9. The precondition (line 2) provides this predicate, but the call to equal (line 8) consumes and does not give back `tokenListSeg(tok,NULL)` because equal's pre- and postconditions are `?`. Thus, a run-time check for `tokenListSeg(tok,NULL)` is required by Gradual C0 at the unfold (line 9). But, equal only retrieves the StringList field `str` from `tok` and calls StringList equals on `str` and `op`. That is, the token list starting at `tok` passed to equal is not modified by the function or its callees preserving `tokenListSeg(tok,NULL)`. Consequently, the author thought about appending `tokenListSeg(tok,NULL)` to equal's pre- and postconditions, which allows skip to be completely statically verified. However, this pushes the run-time check for `tokenListSeg(tok,NULL)` to the end of equal as equals—with its `?` contract—similarly consumes `tokenListSeg(tok,NULL)`. The author would then need to completely statically specify equals for ownership (indicating the function doesn't access any

token list heap locations just StringList ones) to avoid the run-time check in equal, which is a lot of work for a function orthogonal to the author's concerns. Thus, the author decided to leave equal's contract as `?` and move on. Similarly, the author tried to statically prove `tokenListSeg(tok,NULL)` (Fig. 5.8, line 3) is preserved by declspec's loop to reduce run-time checking. If `tokenListSeg(tok,NULL)` is available before `tok` is re-assigned on line 12, then folds and unfolds (lines 11 and 13) can be used to prove `tokenListSeg(tok,NULL)` holds after the assignment at the end of the loop. But, calls to equal and str_[name] functions are again permanently consuming `tokenListSeg(tok,NULL)` provided by the loop invariant at the top of the loop. So, the author gave up here too.

Gradual C0 should provide a "hold" construct to facilitate run-time check reductions in top-down workflows. Unfortunately, calls to equal and other functions not of interest (*e.g.* helper functions for types and variable scopes) happen all throughout the parser functions in TinierCP. So, unless the author statically specifies them for ownership—an orthogonal property to loop termination—she will not be able to reduce run-time checks from Gradual C0 significantly. However, Gradual C0 could be extended to support a `hold` ϕ `{...}` construct, where ϕ is a partial specification containing only predicates and accessibility predicates, to wrap around function calls. This construct tells Gradual C0 to assert ϕ and then reserve the predicates in ϕ and their heap locations from the gradual verification of the program statements in the lexical scope. The held permissions are merged with the ones from the scope at its end. Then, the author can write `hold tokenListSeg(tok,NULL) {equals(tok->str, op);}` in equal; and Gradual C0 keeps `tokenListSeg(tok,NULL)` and its heap locations in the caller equal rather than passing them to the callee equals. Thus, equals does not consume `tokenListSeg(tok,NULL)` statically nor accepts its heap locations dynamically. Gradual C0 can then prove statically that `tokenListSeg(tok,NULL)` holds at the end of equal and eliminate the run-time check for it there. Since equals does not rely on the heap locations in `tokenListSeg(tok,NULL)`, equals still verifies correctly at run time.

Note, the `hold` construct proposed here is an adaptation of the one from prior work on gradual typestate [18], which reserves an access permission to a variable from a lexical scope and then merges the permission of the variable after the scope with the held permission. Similar to Garcia et al. [18]'s `hold`, we also let the lexical scope do what it wants with the permissions it receives as long as it returns sufficient ones to its caller.

Gradual C0's implicit optimism at function calls and branch points made it difficult for the author to track when proof obligations are discharged statically. Using Gradual C0, the author selectively applied static verification to secure questionable execution paths in TinierCP not covered by her test case. So, it was crucial for her to know whether or not Gradual C0 discharged her proof obligations statically. Unfortunately, Gradual C0 does not provide such information forcing the author to track it herself; and worse, Gradual C0's optimism at function calls and branch points made this process harder.

The author often forgot that unspecified, irrelevant functions—with default `?` contracts—were on her verification path and believed proof obligations were discharged statically when they were not. In reality, information available to prove them statically was consumed by such func-

tions' `?` contracts in Gradual C0 resulting in run-time checks for the proof obligations instead. For example, she wrote "[I] noticed that the two simple fold statements that I wrote in declspec for the loop invariant entry to reduce run-time checking does not actually reduce run-time checks due to the is_typename function used in the loop condition subsuming these predicates thanks to its unspecified pre/postconditions." She also wrote "I saw some unexpected additional run time checks that I thought I'd specified away" in reference to `tokenListSeg(tok,NULL)` being checked at run time rather than statically for type_suffix's precondition in declarator's recursive path. Calls to equal and empty_type earlier in the code caused `tokenListSeg(tok,NULL)` to be checked at run time. The author uncovered these misconceptions by inspecting the .verified.c0 intermediate file produced and saved by Gradual C0 on each run. This file contains TinierCP's code modified to include all required run-time checks as determined by Gradual C0. She used this file at least three different times to track her static verification progress.

Additionally, Gradual C0 treats branch points optimistically (as discussed in more detail in Chpt. 3, §3.3) in adherence with the gradual guarantee (Chpt. 2, §2.4). For example, consider `if (x > 2) //@ assert false; else //@ assert true;` where Gradual C0 only knows `?` prior to statically verifying this if statement. Gradual C0 first splits execution and verifies both the `x > 2` branch and `x <= 2` branch; where, the former branch fails due to asserting `false`, while the latter succeeds due to asserting `true`. Since one of the execution paths fails, a normal static verifier would report verification of the if statement a failure. However, Gradual C0's static verifier optimistically assumes `?` contains `x <= 2` and the failing branch is unreachable code. So, Gradual C0's static verifier successfully verifies the if statement and Gradual C0 guards the if statement with a run-time check for `x <= 2` before the if.

While this behavior makes sense in theory, in practice it makes determining whether or not proof obligations are statically verified harder. The author initially wrote a loop invariant for declspec that was too strong for the loop's "`counter > 0`" execution paths. She was never alerted to this issue, because Gradual C0's optimism at branch points turned the static failure down these execution paths (due to it not preserving declspec's loop invariant) into a run-time check for `!(counter > 0)` that is not covered by her test case. The author discovered the issue by reasoning about the `counter > 0` execution paths herself. She ran into this problem three more times: when she was specifying 1) declspec's loop invariant very early in the process, 2) more of the `counter > 0` paths, and 3) parse_typedef's loop invariant. However, in these cases, Gradual C0 reported a run-time error for the check Gradual C0 injected at the optimistic branch point, which was unfortunately more confusing than helpful. Execution is intended to go down the paths the check is guarding against—hence, the run-time failure—and the true cause of the error—which is a mismatch between specifications and code in the statically failing branch—is not reflected by Gradual C0's reporting. The author thought the first error she encountered of this form was a bug in Gradual C0: "ran into what seems like a bug in Gradual C0... in the true case of the loop invariant of declspec". But, eventually figured out what was going on: "run time check error ... in parse_typedef was generated due to the permissiveness of branching in Gradual C0... I had to look in the .verified.c0 [file] to figure out what is going on and this error is confusing on its own".

Gradual C0 should tell users whether or not their proof obligations have been discharged statically. Gradual C0's optimism from adhering to the gradual guarantee is powerful for assuring

loops terminate in TinierCP by suppressing unwanted static verification errors. However, as we have just seen, it also at times suppresses wanted static information (including some errors). Therefore, Gradual C0's user interface should be extended to provide feedback on whether or not user selected proof obligations down user selected execution paths have been statically verified. This feedback should include reports of static verification errors down the selected paths where applicable. For example, the author could select the `tokenListSeg(tok,NULL)` clause (line 3) in the loop invariant from Fig. 5.8 and the true branch of the if statement (line 6). Then, Gradual C0's user interface would report whether or not the clause is proven statically at the end of each execution path in the if statement's true branch—which would be *yes* for the path shown on lines 11-13 (if the author instead selected the unfold on line 11, then *no* would be reported). The reporting could appear in an IDE on the original source program or in a custom visualization tracking Gradual C0's static verification process (*e.g.* an adaptation of the symbolic execution tree from VeriFast's IDE [23]). That is, this solution preserves Gradual C0's current level of optimism and just selectively reports additional information—beyond verification success or failure and related errors—produced by Gradual C0 during static verification.

5.3.5 General Gradual C0 limitations and improvements

While the author was using Gradual C0 to gradually verify TinierCP, she came across a few bugs in the tool, uncovered limitations in the Gradual C0 for codebases like TinierCP, and described minor issues with Gradual C0's error reporting interface. We present these bugs and limitations and propose solutions where applicable in this section.

Bugs found in Gradual C0. The author ran into three bugs in Gradual C0 while she was using the tool to gradually verify the skip function. For the first bug, she ran into an exception in Gradual C0's frontend module that translates run-time checks into C0 source code. The exception was for an invalid '\result' expression in a run-time check, which was previously witnessed in a closed github issue in Gradual C0's main repository: https://github.com/gradual-verification/gvc0/issues/54. She re-opened the issue and removed skip's pre-condition, which was triggering the exception. This bug appears to be a completeness issue not a soundness one. The second issue was Gradual C0 treating logical conditionals as a priority even across `ensures` clauses. For example, `ensures (x > 2 ? false : true); ensures acc(y->f);` is treated as `x > 2 ? false : (true && acc(y->f))` rather than `(x > 2 ? false:true) && acc(y->f)`. The author intended the latter and was surprised to find out Gradual C0 used the former. She swapped the order of her ensures clauses to solve the issue, and we should fix this in general in Gradual C0. Finally, the author discovered that her `unfold` in skip did not produce a run-time check when it should have. As a result, we uncovered and fixed a single line bug in Gradual C0's backend, so `unfold` correctly produces run-time checks when necessary. The performance data in this thesis was updated to account for this bug fix.

Engineering improvements required for larger codebases. The author ran into a few different limitations of Gradual C0 that are related to using the tool on larger codebases (about 3k LoC vs a few hundred LoC for the benchmarks in Chpt. 4). Since Gradual C0 does not currently support

multi-file verification, we implemented all of TinierCP's modules in a single file called cparser.c0 for the author to verify. This resulted in the author scrolling up and down through approximately 2k LoC, which led the author to open two copies of TinierCP's cparser.c0 file in a split screen to help. To make the gradual verification of codebases with multiple modules easier, Gradual C0 should support multi-file verification. Additionally, we can see from Fig. 5.5 that gradually verifying TinierCP without specifications using Gradual C0—which defaults missing specifications to `?`—results in 0.16 seconds of run-time overhead from checking that heap accesses are safe. While this cost is reasonable for our test case (from §5.1.2), larger C programs (larger token lists) may have significantly increased dynamic verification costs. Future work should investigate and implement strategies to reduce the run-time overhead and memory usage of Gradual C0's run-time checking strategy for verifying heap accesses. Similarly, as more specifications were added to TinierCP, Gradual C0's static verification run-time overhead significantly increased—going from 1 minute 1 second without specifications to 3 minutes 7 seconds with the author's final specifications. Thanks to the modularity of static verification and localization of specification changes in a gradual workflow, Gradual C0 could improve its static verifier's performance by only re-verifying a function and its callers when the function's specifications (or code) have changed from one run to another. This and other optimizations for Gradual C0's static verifier should be explored in future work to improve usability in larger codebases.

Minor user interface changes to improve user experience. Finally, the author suggested a handful of minor user interface changes that would have improved her experience. She wants a VSCode plug-in for highlighting Gradual C0 specific constructs that "distinguishes annotations from comments." Furthermore, Gradual C0 reported a run-time check verification error to the author that states "Field access runtime check failed for struct TokenKind.kind." No location information was attached to this error, causing the author to spend a non-trivial amount of time and effort locating the failing field access in TinierCP. Had line numbers or a stack trace been attached to the error message the origin of the error would be obvious (*e.g.* `tok->kind->kind` in is_EOF). Similarly, Gradual C0 reported an "assert failed" run-time error for a run-time check containing a boolean expression. This time line numbers were given, but they corresponded to line numbers in an intermediate file used by Gradual C0 and not the original TinierCP program. Gradual C0 also did not report any details about the failing expression. Consequently, the author looked in the aforementioned intermediate file to figure out what was going on. Going forward, we should improve Gradual C0's error messaging by attaching line numbers and stack traces that correspond to the original C0 program to the messages, and by giving better textual descriptions of failing run-time checks—as inspired by the thorough error messaging from Rust's borrow checker.

5.4 Discussion

Now, we connect our experience from developing TinierCP (§5.1.1) and what we've learned in §5.3 to our research questions.

Trends, themes, and trade-offs. A few trends emerged from using Gradual C0 to verify TinierCP's parser functions for loop termination (**RQ1**). Gradual C0's ability to suppress static verification errors from missing specifications allows users to specify only the software components and properties they care about. Furthermore, TinierCP's parser functions and their loops call a number of other functions and contain a lot of additional code that have no bearing on loop termination. In our study, this led to the author choosing to only specify and verify properties and code relevant to loop termination and do so in a top-down manner—using specifications on callers to inform specifications in callees. In addition to suppressing static verification errors from missing specifications, Gradual C0 also reports static or dynamic verification errors from inconsistencies between specifications and code—as Gradual C0 is sound. This encouraged the author to verify her specifications early and often throughout her top-down verification process, which uncovered bugs in specifications and code (**RQ4**). Dynamic errors helped her find issues with specifications and code earlier than static verification (**RQ3**), but only when test cases covered problematic execution paths (**RQ5**). The author overcame this weakness by using Gradual C0 to prove specific proof obligations statically down execution paths not covered by run-time checking—trading off human effort in exchange for increased assurance (**RQ2**). Static errors from Gradual C0 were also helpful for finding bugs in specifications and code (**RQ3**).

Limitations of Gradual C0 and C0. One of the reasons Chibicc was chosen as our case was because it is implemented with C code that is friendly to C0—avoiding abstractions and clever tricks like higher-order functions, macros, and unions. Despite this, as we saw in §5.1.1, C0 is even more restrictive than we realized initially by disallowing switches, enums, global variables, breaks and continues, and address of. While the changes to Chibicc's code/algorithms to deal with these limitations involved busy work, their common use throughout Chibicc made this process time consuming. We looked at a number of other C parsers written in C and the use of the aforementioned features, pointer casts and arithmetic, macros, and unions were all common. That is, while C0 has worked as a starting language for building Gradual C0 and positions Gradual C0 well for use in educational studies, Gradual C0 should be extended to support a programming language with more expressive features (**RQ5**).

Additionally, the author witnessed a number of user interface, engineering, and fundamental limitations of Gradual C0 (**RQ5**). While run-time check errors were helpful for alerting the author to issues with her specifications and TinierCP's code, minor improvements to their presentation, such as reporting where they originated from in TinierCP and providing more detailed error messaging about the failure, would make the errors more effective in practice. Furthermore, Gradual C0 does not inform its users about optimistic assumptions made during static verification, because most of the time this is the preferred behavior. However, in select cases Gradual C0 should report whether or not a proof obligation is discharged statically or optimistically. Additionally, Gradual C0's lack of multi-file verification support and slow static verification performance on partially specified TinierCP code limits Gradual C0's ability to scale to large codebases. Supporting multi-file verification in Gradual C0 and applying additional optimization techniques to Gradual C0 are both important engineering endeavors. Finally, Gradual C0's inability to support the gradual verification of arrays and strings eliminated a large number of interesting case studies from consideration and resulted in more time consuming and unsatisfying changes to Chibicc's code than C0's limitations. Gradual verification theory should be

extended to support quantification in specifications,—just as we have for ownership and recursive predicates—so arrays and strings can be verified.

Challenges from TinierCP for Gradual C0 (RQ6). Gradual C0's optimism around function calls coupled with TinierCP's parser functions calling a plethora of other helper functions—that don't impact loop termination—made it difficult for the author to make improvements to run-time checking overhead by statically verifying more proof obligations. She would have had to statically verify these helper functions in full for orthogonal properties related to memory safety to make any impact on run-time performance, which is counter-intuitive to her workflow. Instead, future work should investigate a `hold` construct in Gradual C0 (inspired by work in gradual typing [18]) that allows users to specify permissions that should be held from the gradual verification of specific functional calls.

5.5 Threats to Validity

Reliability. The author of this dissertation not only gradually verified TinierCP with Gradual C0 producing the data for this study, she also analyzed the data and wrote up the results in narrative form. While this helped provide additional context to our analysis and results, this may make it harder for other researchers—who don't have this context—to reproduce parts of our results even when following our coding methodology.

External validity. The generality of the findings in this study is hindered by them being drawn from data produced by one person's experiences, opinions, and decisions, especially a person who is an expert on gradual verification technology. Other verification experts would not have the same level of familiarity with Gradual C0 and how it works to gradually verify code even after using the tool prior to the study. Thus, other experts may make different choices throughout the verification process that would collectively result in different conclusions to this study. Follow-up studies should be conducted with a larger number of experts to confirm, deny, or refine the conclusions of this study.

Chapter 6

Conclusion

Gradual verification is a promising approach to supporting incrementality and enhancing the adoptability of program verification. Users can focus on specifying and verifying the most important properties and components of their systems and get immediate feedback about the consistency of their specifications and the correctness of their code. This dissertation makes significant advancements over early work in gradual verification by extending it to support programs manipulating recursive heap data structures. We overcame several key technical challenges, including the semantics of imprecise formulas in the presence of accessibility predicates and recursive predicates, and consistency between iso-recursive static checking and equi-recursive dynamic checking. We formalize our gradual verification approach and prove it sound and that it adheres to important gradual properties like the gradual guarantee (Chpt. 2). Gradual C0 implements these ideas in a working gradual verifier, which facilitates development of gradual verifiers for other languages and minimizes run-time checks and their overhead (Chpt. 3). Experimental results show that our approach can reduce overhead significantly compared to purely dynamic checking and adheres to speculated performance trends—introducing proof obligations increases performance until a critical mass of specifications are written, at which point run-time overhead decreases with additional specifications (Chpt. 4). Finally, Gradual C0 opened the door for us to explore how gradual verification can be used to assure real codebases through a case study. We found gradual verification useful for allowing us to verify only the code and properties relevant to our task, which was a necessity in real software; and also, uncovered bugs in specifications and code far earlier than if we had used static verification alone.

Future Work. The work presented in this dissertation lays a solid foundation for gradual verification, and also positions us well for important future work. Gradual verification soundly and systematically applies run-time checking at the boundaries between modules, which is a necessity for verifying larger codebases. It also can do this at the boundaries with library code, but Gradual C0 is not designed to support this. In fact, not only is Gradual C0's strategy for checking ownership at run time expensive at times (Chpt. 4), it also requires source code to be available to provide sound guarantees. Our formal framework, Gradual C0, and benchmarking infrastructure provide a helpful basis for exploring more optimal ownership checking strategies that can soundly verify code even when source code is unavailable. Additionally, more work remains to extend gradual verification (and Gradual C0) to the expressiveness of state-of-the-art static pro-

gram verifiers, such as supporting quantification in specifications and a full fledged programming language like Rust. Our formal framework and Gradual C0's modular structure should facilitate both of these endeavors nicely. Finally, we witnessed a number of interesting user trends from using Gradual C0 on real code that should be evaluated, confirmed, refined or added to in future user studies with Gradual C0.

Appendix A

Appendix

A.1 Chpt. 2's Appendix

A.1.1 $\mathrm{SVL}_{\mathrm{RP}}$

Formula Semantics

$$\frac{}{H, \rho \vdash v \Downarrow v}\ \textsc{EVal} \qquad \frac{}{H, \rho \vdash x \Downarrow \rho(x)}\ \textsc{EVar} \qquad \frac{H, \rho \vdash e \Downarrow o \qquad H(o) = \langle C, l \rangle}{H, \rho \vdash e.f \Downarrow l(f)}\ \textsc{EvField}$$

$$\frac{H, \rho \vdash e_1 \Downarrow v_1 \qquad H, \rho \vdash e_2 \Downarrow v_2}{H, \rho \vdash e_1 \oplus e_2 \Downarrow v_1 \oplus v_2}\ \textsc{EvOp} \qquad \frac{H, \rho \vdash e_1 \Downarrow v_1 \qquad H, \rho \vdash e_2 \Downarrow v_2}{H, \rho \vdash e_1 \odot e_2 \Downarrow v_1 \odot v_2}\ \textsc{EvComp}$$

Figure A.1: $\mathrm{SVL}_{\mathrm{RP}}$: Expression dynamic semantics

$$\frac{}{\langle H, \rho, \pi \rangle \vDash_E \texttt{true}} \textsc{EvTrueExpr} \qquad \frac{H, \rho \vdash e_1 \odot e_2 \Downarrow \texttt{true}}{\langle H, \rho, \pi \rangle \vDash_E e_1 \odot e_2} \textsc{EvCompExpr}$$

$$\frac{H, \rho \vdash e \Downarrow o \qquad H, \rho \vdash e\texttt{.}f \Downarrow v \qquad \langle o, f \rangle \in \pi}{\langle H, \rho, \pi \rangle \vDash_E \textbf{acc}(e.f)} \textsc{EvAcc}$$

$$\frac{\langle H, \rho, \pi \rangle \vDash_E \phi_1 \qquad \langle H, \rho, \pi \rangle \vDash_E \phi_2}{\langle H, \rho, \pi \rangle \vDash_E \phi_1 \wedge \phi_2} \textsc{EvAndOp} \qquad \frac{\langle H, \rho, \pi_1 \rangle \vDash_E \phi_1 \qquad \langle H, \rho, \pi_2 \rangle \vDash_E \phi_2}{\langle H, \rho, \pi_1 \uplus \pi_2 \rangle \vDash_E \phi_1 * \phi_2} \textsc{EvSepOp}$$

$$\frac{H, \rho \vdash e_1 \Downarrow v_1 \qquad ... \qquad H, \rho \vdash e_n \Downarrow v_n \qquad \langle H, \rho, \pi \rangle \vDash_E \mathsf{body}_\mu(p)(e_1, ..., e_n)}{\langle H, \rho, \pi \rangle \vDash_E p(e_1, ..., e_n)} \textsc{EvPred}$$

$$\frac{H, \rho \vdash e \Downarrow \texttt{true} \qquad \langle H, \rho, \pi \rangle \vDash_E \phi_T}{\langle H, \rho, \pi \rangle \vDash_E \texttt{if } e \texttt{ then } \phi_T \texttt{ else } \phi_F} \textsc{EvCondTrue}$$

$$\frac{H, \rho \vdash e \Downarrow \texttt{false} \qquad \langle H, \rho, \pi \rangle \vDash_E \phi_F}{\langle H, \rho, \pi \rangle \vDash_E \texttt{if } e \texttt{ then } \phi_T \texttt{ else } \phi_F} \textsc{EvCondFalse}$$

$$\frac{\langle H, \rho, \pi \rangle \vDash_E \phi}{\langle H, \rho, \pi \rangle \vDash_E \textbf{unfolding } p(e_1, ..., e_n) \textbf{ in } \phi} \textsc{EvUnfolding}$$

Figure A.2: SVL_{RP}: Formula evaluation

$$\frac{}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} v} \textsc{FrmVal} \qquad \frac{}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} x} \textsc{FrmVar}$$

$$\frac{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_1 \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_2}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_1 \oplus e_2} \textsc{FrmOp}$$

$$\frac{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_1 \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_2}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_1 \odot e_2} \textsc{FrmComp}$$

$$\frac{\langle H, \rho, \Pi \rangle \vDash_I \textbf{acc}(e.f) \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e.f} \textsc{FrmField} \qquad \frac{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \textbf{acc}(e.f)} \textsc{FrmAcc}$$

$$\frac{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \phi_1 \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \phi_2}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \phi_1 \wedge \phi_2} \textsc{FrmAndOp}$$

$$\frac{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \phi_1 \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \phi_2}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \phi_1 \ast \phi_2} \textsc{FrmSepOp}$$

$$\frac{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_1 \qquad \ldots \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_n}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} p(e_1, \ldots, e_n)} \textsc{FrmPred}$$

$$\frac{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e \qquad H, \rho \vdash e \Downarrow \texttt{true} \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \phi_T}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \texttt{if}\ e\ \texttt{then}\ \phi_T\ \texttt{else}\ \phi_F} \textsc{FrmCondTrue}$$

$$\frac{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e \qquad H, \rho \vdash e \Downarrow \texttt{false} \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \phi_F}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \texttt{if}\ e\ \texttt{then}\ \phi_T\ \texttt{else}\ \phi_F} \textsc{FrmCondFalse}$$

$$\frac{\begin{array}{c}\langle H, \rho, \Pi \rangle \vDash_I p(e_1, \ldots, e_n) \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_1 \qquad \ldots \qquad \langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} e_n \\ \langle H, \rho, \Pi' \rangle \vdash_{\texttt{frmI}} \phi \qquad \Pi' = \Pi \cup \lfloor \mathsf{body}_\mu(p)(e_1, \ldots, e_n) \rfloor_{H,\rho}\end{array}}{\langle H, \rho, \Pi \rangle \vdash_{\texttt{frmI}} \textbf{unfolding}\ p(e_1, \ldots, e_n)\ \textbf{in}\ \phi} \textsc{FrmUnfolding}$$

Figure A.3: SVL_{RP}: Framing

$$
\begin{aligned}
\mathsf{acc}(v) &= \texttt{true} \\
\mathsf{acc}(x) &= \texttt{true} \\
\mathsf{acc}(e_1 \odot e_2) &= \mathsf{acc}(e_1) \land \mathsf{acc}(e_2) \\
\mathsf{acc}(e_1 \oplus e_2) &= \mathsf{acc}(e_1) \land \mathsf{acc}(e_2) \\
\mathsf{acc}(e.f) &= \mathsf{acc}(e) \land \texttt{\textbf{acc}}(e.f)
\end{aligned}
$$

Figure A.4: $\mathsf{acc}(e) : \textsc{Expr} \rightarrow \textsc{Formula}$

$$
\begin{array}{ll}
\mathsf{WLP}(\texttt{skip}, \theta) & = \theta \\
\mathsf{WLP}(s_1\texttt{;}\ s_2, \theta) & = \mathsf{WLP}(s_1, \mathsf{WLP}(s_2, \theta)) \\
\mathsf{WLP}(T\ x, \theta) & = \begin{cases} \theta & \texttt{if } x \notin \texttt{FV}(\theta) \\ \texttt{undefined} & \texttt{otherwise} \end{cases} \\
\mathsf{WLP}(\texttt{if}\ (e)\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}, \theta) & = \max_{\Rightarrow} \{\ \theta' \mid \theta' \Rightarrow \texttt{if } e \texttt{ then } \mathsf{WLP}(s_1, \theta) \texttt{ else } \mathsf{WLP}(s_2, \theta) \quad \wedge \\
& \theta' \Rightarrow \mathsf{acc}(e)\ \} \\
\mathsf{WLP}(x \texttt{ := } e, \theta) & = \max_{\Rightarrow} \{\ \theta' \mid \theta' \Rightarrow \theta[e/x] \quad \wedge \quad \theta' \Rightarrow \mathsf{acc}(e)\ \} \\
\mathsf{WLP}(\texttt{while }(e)\texttt{ inv } \theta_i\ \{\ s\ \}, \theta) & = \max_{\Rightarrow} \{\ \theta' \mid \theta' \Rightarrow \mathsf{acc}(e) \quad \wedge \quad \exists \theta_f\ .\ \theta' \Rightarrow \theta_i * \theta_f \quad \wedge \quad \overline{x_i} \notin \mathsf{FV}(\theta_f) \quad \wedge \\
& \theta_f * (\theta_i * \texttt{(} e \texttt{ = false))}[\overline{x_i/y_i}] \Rightarrow \theta[\overline{x_i/y_i}]\ \} \\
& \texttt{where } \overline{y_i} \texttt{ are variables modified by the loop body } s \\
& \texttt{and } \overline{x_i} \texttt{ are fresh logical variables} \\
\mathsf{WLP}(x\texttt{.}f \texttt{ := } y, \theta) & = \textbf{acc}\texttt{(}x.f\texttt{)} \quad \wedge \quad \theta[y/x.f] \\
\mathsf{WLP}(x \texttt{ := new } C, \theta) & = \max_{\Rightarrow} \{\ \theta' \mid x \notin \mathsf{FV}(\theta') \quad \wedge \\
& \theta' * x \neq \mathit{null} * \overline{x \neq e_i} * \overline{\mathit{acc}(x.f_i)} * \overline{x.f_i = \mathsf{defaultValue}(T_i)} \Rightarrow \theta\ \} \\
& \texttt{where fields(C)} = \overline{T_i\ f_i} \texttt{ and } x \neq e_i \texttt{ is a conjunctive term} \\
& \texttt{in } \theta \\
\mathsf{WLP}(y \texttt{ := } z.m(\overline{x}), \theta) & = \max_{\Rightarrow} \{\ \theta' \mid y \notin \mathsf{FV}(\theta') \quad \wedge \\
& \exists \theta_f\ .\ \theta' \Rightarrow \texttt{(}z \neq \texttt{null)} * \mathsf{mpre}(m)[z/\texttt{this}, \overline{x/\mathsf{mparam}(m)}] * \theta_f \quad \wedge \\
& \theta_f * \mathsf{mpost}(m)[z/\texttt{this}, \overline{x/\textbf{old}\texttt{(}\mathsf{mparam}(m)\texttt{)}}, y/\texttt{result}] \Rightarrow \theta\ \} \\
\mathsf{WLP}(\texttt{assert } \phi_a, \theta) & = \max_{\Rightarrow} \{\ \theta' \mid \theta' \Rightarrow \theta \quad \wedge \quad \theta' \Rightarrow \phi_a\ \} \\
\mathsf{WLP}(\texttt{fold } p\texttt{(}e_1\texttt{,}...\texttt{,}e_n\texttt{)}, \theta) & = \max_{\Rightarrow} \{\ \phi' \mid \phi' * p\texttt{(}e_1\texttt{,}...\texttt{,}e_n\texttt{)} \Rightarrow \theta \quad \wedge \\
& \quad \phi' * p\texttt{(}e_1\texttt{,}...\texttt{,}e_n\texttt{)} \in \textsc{SatFormula} \quad \wedge \\
& \quad \phi' * \mathsf{body}_\mu(p)(e_1, ..., e_n) \in \textsc{SfrmFormula}\ \} * \mathsf{body}_\mu(p)(e_1, ..., e_n) \\
& \texttt{if this result exists and is satisfiable, undefined} \\
& \texttt{otherwise} \\
\mathsf{WLP}(\texttt{unfold } p\texttt{(}e_1\texttt{,}...\texttt{,}e_n\texttt{)}, \theta) & = \max_{\Rightarrow} \{\ \phi' \mid \phi' * \mathsf{body}_\mu(p)(e_1, ..., e_n) \Rightarrow \theta \quad \wedge \\
& \quad \phi' * \mathsf{body}_\mu(p)(e_1, ..., e_n) \in \textsc{SatFormula} \quad \wedge \\
& \quad \phi' * p\texttt{(}e_1\texttt{,}...\texttt{,}e_n\texttt{)} \in \textsc{SfrmFormula}\ \} * p\texttt{(}e_1\texttt{,}...\texttt{,}e_n\texttt{)} \\
& \texttt{if this result exists and is satisfiable, undefined} \\
& \texttt{otherwise}
\end{array}
$$

Figure A.5: SVL_{RP}: Weakest liberal precondition calculus

Dynamic Semantics

$$\frac{}{\langle H, \langle \rho, \pi, \texttt{skip} \rangle \cdot nil \rangle \quad \texttt{final}} \text{ SsSkipFin} \qquad \frac{}{\langle H, \langle \rho, \pi, \texttt{skip;}\ s \rangle \cdot S \rangle \longrightarrow \langle H, \langle \rho, \pi, s \rangle \cdot S \rangle} \text{ SsSkip}$$

$$\frac{}{\langle H, \langle \rho, \pi, T\ x\texttt{;}\ s \rangle \cdot S \rangle \longrightarrow \langle H, \langle \rho, \pi, s \rangle \cdot S \rangle} \text{ SsDeclare}$$

$$\frac{\langle H, \rho, \pi \rangle \vDash_E \phi}{\langle H, \langle \rho, \pi, \texttt{assert}\ \phi\texttt{;}\ s \rangle \cdot S \rangle \longrightarrow \langle H, \langle \rho, \pi, s \rangle \cdot S \rangle} \text{ SsAssert}$$

$$\frac{\langle H, \rho, \pi \rangle \vDash_E \textbf{acc}(x.f) \qquad H, \rho \vdash y \Downarrow v \qquad H' = H[o \mapsto [f \mapsto v]]}{\langle H, \langle \rho, \pi, x\texttt{.}f\ \texttt{:=}\ y\texttt{;}\ s \rangle \cdot S \rangle \longrightarrow \langle H', \langle \rho, \pi, s \rangle \cdot S \rangle} \text{ SsFAssign}$$

$$\frac{\langle H, \rho, \pi \rangle \vDash_E \mathsf{acc}(e) \qquad H, \rho \vdash e \Downarrow v \qquad \rho' = \rho[x \mapsto v]}{\langle H, \langle \rho, \pi, x\ \texttt{:=}\ e\texttt{;}\ s \rangle \cdot S \rangle \longrightarrow \langle H, \langle \rho', \pi, s \rangle \cdot S \rangle} \text{ SsAssign}$$

$$\frac{o \notin \mathsf{dom}(H) \qquad \mathsf{fields}(C) = \overline{T_i\ \ f_i\texttt{;}} \qquad H' = H[o \mapsto [\overline{f_i \mapsto \mathsf{defaultValue}(T_i)}]]}{\langle H, \langle \rho, \pi, x\ \texttt{:=}\ \texttt{new}\ C\texttt{;}\ s \rangle \cdot S \rangle \longrightarrow \langle H', \langle \rho[x \mapsto o], \pi \cup \overline{\langle o, f_i \rangle}, s \rangle \cdot S \rangle} \text{ SsAlloc}$$

$$\frac{\langle H, \rho, \pi \rangle \vDash_E \mathsf{acc}(e) \qquad H, \rho \vdash e \Downarrow \texttt{true}}{\langle H, \langle \rho, \pi, \texttt{if}\ (e)\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}\texttt{;}\ s \rangle \cdot S \rangle \longrightarrow \langle H, \langle \rho, \pi, s_1\texttt{;}\ s \rangle \cdot S \rangle} \text{ SsIfTrue}$$

$$\frac{\langle H, \rho, \pi \rangle \vDash_E \mathsf{acc}(e) \qquad H, \rho \vdash e \Downarrow \texttt{false}}{\langle H, \langle \rho, \pi, \texttt{if}\ (e)\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}\texttt{;}\ s \rangle \cdot S \rangle \longrightarrow \langle H, \langle \rho, \pi, s_2\texttt{;}\ s \rangle \cdot S \rangle} \text{ SsIfFalse}$$

Figure A.6: SVL_{RP}: Small-step semantics

$$\dfrac{\begin{array}{c}\mathsf{method}(m) = T_r\ m\,(\overline{T\ x'})\ \texttt{requires}\ \theta_p\ \texttt{ensures}\ \theta_q\ \{\ r\ \} \qquad H,\rho \vdash z \Downarrow o \qquad \overline{H,\rho \vdash x \Downarrow v}\\ \rho' = [\texttt{this} \mapsto o, \overline{x' \mapsto v}, \overline{\textbf{old}\,(x') \mapsto v}] \qquad \pi' = \langle\langle\ \lfloor\theta_p\rfloor_{H,\rho'}\ \rangle\rangle_H \qquad \pi' \subseteq \pi \qquad \langle H,\rho',\pi'\rangle \vDash_E \theta_p\end{array}}{\langle H, \langle \rho, \pi, y \texttt{ := } z.m(\overline{x})\texttt{;}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho', \pi', r\texttt{; skip}\rangle \cdot \langle \rho, \pi\backslash\pi', y \texttt{ := } z.m(\overline{x})\texttt{;}\ s\rangle \cdot S\rangle}\ \textsc{SsCall}$$

$$\dfrac{\mathsf{mpost}(m) = \theta_q \qquad \langle H,\rho',\pi'\rangle \vDash_E \theta_q \qquad \rho'' = \rho[y \mapsto \rho'(\texttt{result})]}{\langle H, \langle \rho', \pi', \texttt{skip}\rangle \cdot \langle \rho, \pi, y \texttt{ := } z.m(\overline{x})\texttt{;}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho'', \pi \cup \pi', s\rangle \cdot S\rangle}\ \textsc{SsCallFinish}$$

$$\dfrac{\langle H,\rho,\pi\rangle \vDash_E \theta_i \qquad \langle H,\rho,\pi\rangle \vDash_E \mathsf{acc}(e) \qquad H,\rho \vdash e \Downarrow \texttt{false}}{\langle H, \langle \rho, \pi, \texttt{while (}e\texttt{) inv}\ \theta_i\ \{\ r\ \}\texttt{;}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho, \pi, s\rangle \cdot S\rangle}\ \textsc{SsWhileFalse}$$

$$\dfrac{\begin{array}{c}\langle H,\rho,\pi\rangle \vDash_E \theta_i \qquad \langle H,\rho,\pi\rangle \vDash_E \mathsf{acc}(e) \qquad H,\rho \vdash e \Downarrow \texttt{true}\\ \pi' = \langle\langle\ \lfloor\theta_i\rfloor_{H,\rho}\ \rangle\rangle_H\end{array}}{\begin{array}{c}\langle H, \langle \rho, \pi, \texttt{while (}e\texttt{) inv}\ \theta_i\ \{\ r\ \}\texttt{;}\ s\rangle \cdot S\rangle\\ \longrightarrow\\ \langle H, \langle \rho, \pi', r\texttt{; skip}\rangle \cdot \langle \rho, \pi\backslash\pi', \texttt{while (}e\texttt{) inv}\ \theta_i\ \{\ r\ \}\texttt{;}\ s\rangle \cdot S\rangle\end{array}}\ \textsc{SsWhileTrue}$$

$$\dfrac{}{\begin{array}{c}\langle H, \langle \rho', \pi', \texttt{skip}\rangle \cdot \langle \rho, \pi, \texttt{while (}e\texttt{) inv}\ \theta_i\ \{\ r\ \}\texttt{;}\ s\rangle \cdot S\rangle\\ \longrightarrow\\ \langle H, \langle \rho', \pi \cup \pi', \texttt{while (}e\texttt{) inv}\ \theta_i\ \{\ r\ \}\texttt{;}\ s\rangle \cdot S\rangle\end{array}}\ \textsc{SsWhileFinish}$$

$$\dfrac{}{\langle H, \langle \rho, \pi, \texttt{fold}\ p\,(e_1\texttt{,}\ldots\texttt{,}e_n)\texttt{;}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho, \pi, s\rangle \cdot S\rangle}\ \textsc{SsFold}$$

$$\dfrac{}{\langle H, \langle \rho, \pi, \texttt{unfold}\ p\,(e_1\texttt{,}\ldots\texttt{,}e_n)\texttt{;}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho, \pi, s\rangle \cdot S\rangle}\ \textsc{SsUnfold}$$

Figure A.6: SVL_{RP}: Small-step semantics (continued)

Weakest Precondition across stack frames

The formal statement of soundness relies on an extended definition of WLP given in Figure A.7. It is used to validate arbitrary intermediate program states (Def. 2.1.6), and in particular, program states with multiple stack frames. sWLP accepts a stack of statements and postcondition θ and returns a stack of preconditions by recursively picking up the postconditions of methods or loop invariants of loops. sWLP relies on sWLP^{θ_f} to weaken each precondition in the stack except the top-most one. A precondition is weakened by ensuring its accessibility predicates and predicate instances are *disjoint* from those in θ_f. Effectively, θ_f represents the implicit frame of the executing method or loop, so ownership given by θ_f is withdrawn from the call site aligning with SVL_{RP}'s runtime semantics.

For example, imagine a program state with a lower stack frame i having a WLP of `acc(x.f) * (x.f = 3)`. Assume that access to `x.f` was passed up the call stack (*i.e.* it was demanded by the preconditions of called methods or invariants of executing loops), so currently executing statements can *change* the value of `x.f`. As a result, $\langle H, \rho_i, \pi_i\rangle \vDash_E \mathsf{sWLP}_i(s_n \cdot \ldots \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ is violated. We solve this problem by making sure that the stack frame does *not* have a WLP of `acc(x.f) * (x.f = 3)` if it is currently buried under other stack frames that own `x.f`.

$$\mathsf{sWLP}(s \cdot \mathsf{nil}, \theta) = \mathsf{WLP}(s, \theta) \cdot \mathsf{nil}$$

$$\mathsf{sWLP}(s \cdot (y \texttt{ := } z.m(\overline{x})\texttt{; } s') \cdot \overline{s}, \theta) = \mathsf{WLP}(s, \mathsf{mpost}(m)) \cdot \mathsf{sWLP}^{\mathsf{mpre}(m)[z/\texttt{this},\overline{x/\mathsf{mparam}(m)}]}((y \texttt{ := } z.m(\overline{x})\texttt{; } s') \cdot \overline{s}, \theta)$$

$$\mathsf{sWLP}(s \cdot (\texttt{while (}e\texttt{) inv}\ \theta_i \ \{\ r\ \}\texttt{; } s') \cdot \overline{s}, \theta) = \mathsf{WLP}(s, \theta_i) \cdot \mathsf{sWLP}^{\theta_i}((\texttt{while (}e\texttt{) inv}\ \theta_i \ \{\ r\ \}\texttt{; } s') \cdot \overline{s}, \theta)$$

where $\mathsf{sWLP}^{\theta_f}(\overline{s}, \theta) = \min_{\Rightarrow}\{\ \theta'_n \mid \theta_n \Rightarrow \theta_f * \theta'_n\ \} \cdot \theta_{n-1} \cdot ... \cdot \theta_1 \cdot \mathsf{nil}$

and $\theta_n \cdot \theta_{n-1} \cdot ... \cdot \theta_1 \cdot \mathsf{nil} = \mathsf{sWLP}(\overline{s}, \theta)$

Figure A.7: Heap aware weakest liberal precondition across multiple stack frames

A.1.2 GVL$_{\text{RP}}$

Framing

$$
\begin{aligned}
\mathsf{TotalFP}(v, H, \rho) &= \emptyset \\
\mathsf{TotalFP}(x, H, \rho) &= \emptyset \\
\mathsf{TotalFP}(e_1 \odot e_2, H, \rho) &= \mathsf{TotalFP}(e_1, H, \rho) \cup \mathsf{TotalFP}(e_2, H, \rho) \\
\mathsf{TotalFP}(e_1 \oplus e_2, H, \rho) &= \mathsf{TotalFP}(e_1, H, \rho) \cup \mathsf{TotalFP}(e_2, H, \rho) \\
\mathsf{TotalFP}(e.f, H, \rho) &= \mathsf{TotalFP}(e, H, \rho) \cup \{\langle o, f\rangle | H, \rho \vdash e \Downarrow o\} \\
\mathsf{TotalFP}(\mathbf{acc}(e.f), H, \rho) &= \mathsf{TotalFP}(e.f, H, \rho) \\
\mathsf{TotalFP}(\phi_1 \wedge \phi_2, H, \rho) &= \mathsf{TotalFP}(\phi_1, H, \rho) \cup \mathsf{TotalFP}(\phi_2, H, \rho) \\
\mathsf{TotalFP}(\phi_1 * \phi_2, H, \rho) &= \mathsf{TotalFP}(\phi_1, H, \rho) \cup \mathsf{TotalFP}(\phi_2, H, \rho) \\
\mathsf{TotalFP}(p(e_1, ..., e_n), H, \rho) &= \mathsf{TotalFP}(e_1, H, \rho) \cup ... \cup \mathsf{TotalFP}(e_n, H, \rho) \cup \\
&\quad \{\langle p, v_1, ..., v_n\rangle | H, \rho \vdash e_1 \Downarrow v_1, ..., H, \rho \vdash e_n \Downarrow v_n\} \\
\mathsf{TotalFP}(\texttt{if}\ e\ \texttt{then}\ \phi_1\ \texttt{else}\ \phi_2, H, \rho) &= \begin{cases} \mathsf{TotalFP}(e, H, \rho) \cup \mathsf{TotalFP}(\phi_1, H, \rho) & \texttt{if}\ H, \rho \vdash e \Downarrow \texttt{true} \\ \mathsf{TotalFP}(e, H, \rho) \cup \mathsf{TotalFP}(\phi_2, H, \rho) & \texttt{if}\ H, \rho \vdash e \Downarrow \texttt{false} \\ \emptyset & \texttt{otherwise} \end{cases} \\
\mathsf{TotalFP}(\mathbf{unfolding}\ p(e_1, ..., e_n)\ \mathbf{in}\ \phi, H, \rho) &= \mathsf{TotalFP}(p(e_1, ..., e_n), H, \rho) \cup \mathsf{TotalFP}(\phi, H, \rho)
\end{aligned}
$$

Figure A.8: Definition of the TotalFP function.

Lifting functions

$$
\begin{aligned}
&\widetilde{\mathsf{WLP}}(s_1\texttt{;}\ s_2, \widetilde{\phi}) = \widetilde{\mathsf{WLP}}(s_1, \widetilde{\mathsf{WLP}}(s_2, \widetilde{\phi}))\\
&\widetilde{\mathsf{WLP}}(\texttt{if}\ (e)\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}, \widetilde{\phi}) = \alpha(\{\ \max_{\Rightarrow}\{\ \phi' \in \textsc{SatFormula} \mid \phi' \Rightarrow \texttt{if}\ e\ \texttt{then}\ \theta_1\ \texttt{else}\ \theta_2 \ \wedge\\
&\qquad \phi' \Rightarrow \mathsf{acc}(e) \wedge \vdash_{\texttt{frm}} \langle \phi', \mathsf{body}_\Delta{}' \rangle\ \} \mid \theta_1 \in \gamma(\widetilde{\mathsf{WLP}}(s_1, \widetilde{\phi})), \theta_2 \in \gamma(\widetilde{\mathsf{WLP}}(s_2, \widetilde{\phi})),\\
&\qquad \mathsf{body}_\Delta{}' \in \gamma(\mathsf{body}_\mu),\ \vdash_{\texttt{frm}} \langle \theta_1, \mathsf{body}_\Delta{}' \rangle,\ \vdash_{\texttt{frm}} \langle \theta_2, \mathsf{body}_\Delta{}' \rangle\ \})\\
&\widetilde{\mathsf{WLP}}(y\ \texttt{:=}\ z.m(\overline{x}), \widetilde{\phi}) = \alpha(\{\ \max_{\Rightarrow}\{\ \phi' \in \textsc{SatFormula} \mid y \notin \mathsf{FV}(\phi') \wedge \vdash_{\texttt{frm}} \langle \phi', \mathsf{body}_\Delta{}' \rangle \ \wedge\\
&\qquad \exists \phi_f\ .\ \phi' \Rightarrow \texttt{(}z \neq \texttt{null)} * \theta_p[z/\texttt{this}, \overline{x/\mathsf{mparam}(m)}] * \phi_f \ \wedge\\
&\qquad \phi_f * \theta_q[z/\texttt{this}, \overline{x/\texttt{\textbf{old}(}\mathsf{mparam}(m)\texttt{)}}, y/\texttt{result}] \Rightarrow \theta \wedge \vdash_{\texttt{frm}} \langle \phi_f, \mathsf{body}_\Delta{}' \rangle\ \}\\
&\qquad \mid \theta \in \gamma(\widetilde{\phi}),\ \theta_p \in \gamma(\mathsf{mpre}(m)),\ \theta_q \in \gamma(\mathsf{mpost}(m)),\ \mathsf{body}_\Delta{}' \in \gamma(\mathsf{body}_\mu),\\
&\qquad \vdash_{\texttt{frm}} \langle \theta, \mathsf{body}_\Delta{}' \rangle,\ \vdash_{\texttt{frm}} \langle \theta_p, \mathsf{body}_\Delta{}' \rangle,\ \vdash_{\texttt{frm}} \langle \theta_q, \mathsf{body}_\Delta{}' \rangle\ \})\\
&\widetilde{\mathsf{WLP}}(\texttt{while}\ \texttt{(}e\texttt{)}\ \texttt{inv}\ \widetilde{\phi_i}\ \{\ s\ \}, \widetilde{\phi}) = \alpha(\{\ \max_{\Rightarrow}\{\ \phi' \in \textsc{SatFormula} \mid \phi' \Rightarrow \mathsf{acc}(e) \wedge \vdash_{\texttt{frm}} \langle \phi', \mathsf{body}_\Delta{}' \rangle \ \wedge\\
&\qquad \exists \phi_f\ .\ \phi' \Rightarrow \theta_i * \phi_f \ \wedge\ \overline{x_i} \notin \mathsf{FV}(\phi_f) \wedge \vdash_{\texttt{frm}} \langle \phi_f, \mathsf{body}_\Delta{}' \rangle \ \wedge\\
&\qquad \phi_f * (\theta_i * \texttt{(}e\ \texttt{=}\ \texttt{false))}[\overline{x_i/y_i}] \Rightarrow \theta[\overline{x_i/y_i}]\ \}\\
&\qquad \mid \theta \in \gamma(\widetilde{\phi}),\ \theta_i \in \gamma(\widetilde{\phi_i}),\ \mathsf{body}_\Delta{}' \in \gamma(\mathsf{body}_\mu),\ \vdash_{\texttt{frm}} \langle \theta, \mathsf{body}_\Delta{}' \rangle,\ \vdash_{\texttt{frm}} \langle \theta_i, \mathsf{body}_\Delta{}' \rangle\ \})\\
&\qquad \texttt{where}\ \overline{y_i}\ \texttt{are vars modified by the loop body}\ s\ \texttt{and}\ \overline{x_i}\ \texttt{are fresh}\\
&\widetilde{\mathsf{WLP}}(\texttt{fold}\ p\texttt{(}\overline{e}\texttt{)}, \widetilde{\phi}) = \alpha(\{\ \max_{\Rightarrow}\{\ \phi' \in \textsc{SatFormula} \mid \phi' * p\texttt{(}\overline{e}\texttt{)} \Rightarrow \theta \ \wedge\ \phi' * p\texttt{(}\overline{e}\texttt{)} \in \textsc{SatFormula} \wedge\\
&\qquad \vdash_{\texttt{frm}} \langle \phi' * \mathsf{body}_\Delta{}'(p)(\overline{e}), \mathsf{body}_\Delta{}' \rangle\ \} * \mathsf{body}_\Delta{}'(p)(\overline{e}) \in \textsc{SatFormula}\\
&\qquad \mid \theta \in \gamma(\widetilde{\phi}),\ \mathsf{body}_\Delta{}' \in \gamma(\mathsf{body}_\mu),\ \vdash_{\texttt{frm}} \langle \theta, \mathsf{body}_\Delta{}' \rangle\ \})\\
&\widetilde{\mathsf{WLP}}(\texttt{unfold}\ p\texttt{(}\overline{e}\texttt{)}, \widetilde{\phi}) = \alpha(\{\ \max_{\Rightarrow}\{\ \phi' \in \textsc{SatFormula} \mid \phi' * \mathsf{body}_\Delta{}'(p)(\overline{e}) \Rightarrow \theta \ \wedge\\
&\qquad \phi' * \mathsf{body}_\Delta{}'(p)(\overline{e}) \in \textsc{SatFormula} \ \wedge\ \vdash_{\texttt{frm}} \langle \phi' * p\texttt{(}\overline{e}\texttt{)}, \mathsf{body}_\Delta{}' \rangle\ \} * p\texttt{(}\overline{e}\texttt{)} \in \textsc{SatFormula}\\
&\qquad \mid \theta \in \gamma(\widetilde{\phi}),\ \mathsf{body}_\Delta{}' \in \gamma(\mathsf{body}_\mu),\ \vdash_{\texttt{frm}} \langle \theta, \mathsf{body}_\Delta{}' \rangle\ \})\\
&\widetilde{\mathsf{WLP}}(s, \widetilde{\phi}) = \alpha(\{\ \mathsf{WLP}(s, \theta, \mathsf{body}_\Delta{}') \mid \theta \in \gamma(\widetilde{\phi}),\ \mathsf{body}_\Delta{}' \in \gamma(\mathsf{body}_\mu),\ \vdash_{\texttt{frm}} \langle \theta, \mathsf{body}_\Delta{}' \rangle\ \}) \quad \text{otherwise}
\end{aligned}
$$

Figure A.9: $\mathrm{GVL_{RP}}$: Weakest liberal precondition calculus.

Dynamic semantics

$$\frac{}{\langle H, \langle \rho, \pi, \mathtt{skip} \rangle \cdot nil \rangle \quad \mathtt{final}} \ \textsc{SsSkipFin} \qquad \frac{}{\langle H, \langle \rho, \pi, \mathtt{skip;}\ s \rangle \cdot S \rangle \xrightarrow{\sim} \langle H, \langle \rho, \pi, s \rangle \cdot S \rangle} \ \textsc{SsSkip}$$

$$\frac{}{\langle H, \langle \rho, \pi, T\ x\mathtt{;}\ s \rangle \cdot S \rangle \xrightarrow{\sim} \langle H, \langle \rho, \pi, s \rangle \cdot S \rangle} \ \textsc{SsDeclare}$$

$$\frac{\langle H, \rho, \pi \rangle \ \widetilde{\vDash}\ \langle \mathtt{?}\ *\ \phi, \mathsf{body}_\mu \rangle}{\langle H, \langle \rho, \pi, \mathtt{assert}\ \phi\mathtt{;}\ s \rangle \cdot S \rangle \xrightarrow{\sim} \langle H, \langle \rho, \pi, s \rangle \cdot S \rangle} \ \textsc{SsAssert}$$

$$\frac{\langle H, \rho, \pi \rangle \ \widetilde{\nvDash}\ \langle \mathtt{?}\ *\ \phi, \mathsf{body}_\mu \rangle}{\langle H, \langle \rho, \pi, \mathtt{assert}\ \phi\mathtt{;}\ s \rangle \cdot S \rangle \xrightarrow{\sim} \textbf{error}} \ \textsc{SsAssertError}$$

$$\frac{\langle H, \rho, \pi \rangle \vDash_E \mathbf{acc}(x.f) \qquad H, \rho \vdash y \Downarrow v \qquad H' = H[o \mapsto [f \mapsto v]]}{\langle H, \langle \rho, \pi, x\,\mathtt{.}\,f\ \mathtt{:=}\ y\mathtt{;}\ s \rangle \cdot S \rangle \xrightarrow{\sim} \langle H', \langle \rho, \pi, s \rangle \cdot S \rangle} \ \textsc{SsFAssign}$$

$$\frac{\langle H, \rho, \pi \rangle \nvDash_E \mathbf{acc}(x.f)}{\langle H, \langle \rho, \pi, x\,\mathtt{.}\,f\ \mathtt{:=}\ y\mathtt{;}\ s \rangle \cdot S \rangle \xrightarrow{\sim} \textbf{error}} \ \textsc{SsFAssignError}$$

$$\frac{\langle H, \rho, \pi \rangle \vDash_E \mathsf{acc}(e) \qquad H, \rho \vdash e \Downarrow v \qquad \rho' = \rho[x \mapsto v]}{\langle H, \langle \rho, \pi, x\ \mathtt{:=}\ e\mathtt{;}\ s \rangle \cdot S \rangle \xrightarrow{\sim} \langle H, \langle \rho', \pi, s \rangle \cdot S \rangle} \ \textsc{SsAssign}$$

$$\frac{\langle H, \rho, \pi \rangle \nvDash_E \mathsf{acc}(e)}{\langle H, \langle \rho, \pi, x\ \mathtt{:=}\ e\mathtt{;}\ s \rangle \cdot S \rangle \xrightarrow{\sim} \textbf{error}} \ \textsc{SsAssignError}$$

$$\frac{o \notin \mathsf{dom}(H) \qquad \mathsf{fields}(C) = \overline{T_i\ \ f_i\mathtt{;}} \qquad H' = H[o \mapsto [\overline{f_i \mapsto \mathsf{defaultValue}(T_i)}]]}{\langle H, \langle \rho, \pi, x\ \mathtt{:=}\ \mathtt{new}\ C\mathtt{;}\ s \rangle \cdot S \rangle \xrightarrow{\sim} \langle H', \langle \rho[x \mapsto o], \pi \cup \overline{\langle o, f_i \rangle}, s \rangle \cdot S \rangle} \ \textsc{SsAlloc}$$

$$\frac{\langle H, \rho, \pi \rangle \vDash_E \mathsf{acc}(e) \qquad H, \rho \vdash e \Downarrow \mathtt{true}}{\langle H, \langle \rho, \pi, \mathtt{if}\ (e)\ \{\ s_1\ \}\ \mathtt{else}\ \{\ s_2\ \}\mathtt{;}\ s \rangle \cdot S \rangle \xrightarrow{\sim} \langle H, \langle \rho, \pi, s_1\mathtt{;}\ s \rangle \cdot S \rangle} \ \textsc{SsIfTrue}$$

$$\frac{\langle H, \rho, \pi \rangle \vDash_E \mathsf{acc}(e) \qquad H, \rho \vdash e \Downarrow \mathtt{false}}{\langle H, \langle \rho, \pi, \mathtt{if}\ (e)\ \{\ s_1\ \}\ \mathtt{else}\ \{\ s_2\ \}\mathtt{;}\ s \rangle \cdot S \rangle \xrightarrow{\sim} \langle H, \langle \rho, \pi, s_2\mathtt{;}\ s \rangle \cdot S \rangle} \ \textsc{SsIfFalse}$$

$$\frac{\langle H, \rho, \pi \rangle \nvDash_E \mathsf{acc}(e)}{\langle H, \langle \rho, \pi, \mathtt{if}\ (e)\ \{\ s_1\ \}\ \mathtt{else}\ \{\ s_2\ \}\mathtt{;}\ s \rangle \cdot S \rangle \xrightarrow{\sim} \textbf{error}} \ \textsc{SsIfError}$$

Figure A.10: GVL_{RP}: Small-step semantics adjusted from Fig. A.6 for gradual formulas

$$\dfrac{\begin{array}{c}\mathsf{method}(m) = T_r\ m\,(\overline{T\,x'})\ \texttt{requires}\ \widetilde{\phi}_p\ \texttt{ensures}\ \widetilde{\phi}_q\ \{\ r\ \} \\ H,\rho \vdash z \Downarrow o \qquad \overline{H,\rho \vdash x \Downarrow v} \qquad \rho' = [\texttt{this} \mapsto o, \overline{x' \mapsto v}, \overline{\textbf{old}\,(x') \mapsto v}] \\ \pi' = \lfloor \widetilde{\phi}_p \rfloor_{\pi,H,\rho'} \qquad \pi' \subseteq \pi \qquad \langle H, \rho', \pi' \rangle \,\widetilde{\vDash}\, \langle \widetilde{\phi}_p, \mathsf{body}_\mu \rangle\end{array}}{\langle H, \langle \rho, \pi, y := z.m(\overline{x});\ s\rangle \cdot S\rangle \,\widetilde{\longrightarrow}\, \langle H, \langle \rho', \pi', r;\ \texttt{skip}\rangle \cdot \langle \rho, \pi \backslash \pi', y := z.m(\overline{x});\ s\rangle \cdot S\rangle}\ \textsc{SsCall}$$

$$\dfrac{\langle H,\rho,\pi\rangle \,\widetilde{\nvDash}\, \langle \mathsf{mpre}(m)[z/\texttt{this}, \overline{x/\mathsf{mparam}(m)}], \mathsf{body}_\mu\rangle}{\langle H, \langle \rho, \pi, y := z.m(\overline{x});\ s\rangle \cdot S\rangle \,\widetilde{\longrightarrow}\, \textbf{error}}\ \textsc{SsCallError}$$

$$\dfrac{\mathsf{mpost}(m) = \widetilde{\phi}_q \qquad \langle H, \rho', \pi' \rangle \,\widetilde{\vDash}\, \langle \widetilde{\phi}_q, \mathsf{body}_\mu \rangle \qquad \rho'' = \rho[y \mapsto \rho'(\texttt{result})]}{\langle H, \langle \rho', \pi', \texttt{skip}\rangle \cdot \langle \rho, \pi, y := z.m(\overline{x});\ s\rangle \cdot S\rangle \,\widetilde{\longrightarrow}\, \langle H, \langle \rho'', \pi \cup \pi', s\rangle \cdot S\rangle}\ \textsc{SsCallFinish}$$

$$\dfrac{\langle H, \rho', \pi' \rangle \,\widetilde{\nvDash}\, \langle \mathsf{mpost}(m), \mathsf{body}_\mu \rangle}{\langle H, \langle \rho', \pi', \texttt{skip}\rangle \cdot \langle \rho, \pi, y := z.m(\overline{x});\ s\rangle \cdot S\rangle \,\widetilde{\longrightarrow}\, \textbf{error}}\ \textsc{SsCallFinishError}$$

$$\dfrac{\langle H,\rho,\pi\rangle \,\widetilde{\vDash}\, \langle \widetilde{\phi}_i, \mathsf{body}_\mu\rangle \qquad \langle H,\rho,\pi\rangle \vDash_E \mathsf{acc}(e) \qquad H,\rho \vdash e \Downarrow \texttt{false}}{\langle H, \langle \rho, \pi, \texttt{while}\ (e)\ \texttt{inv}\ \widetilde{\phi}_i\ \{\ r\ \};\ s\rangle \cdot S\rangle \,\widetilde{\longrightarrow}\, \langle H, \langle \rho, \pi, s\rangle \cdot S\rangle}\ \textsc{SsWhileFalse}$$

$$\dfrac{\begin{array}{c}\langle H,\rho,\pi\rangle \,\widetilde{\vDash}\, \langle \widetilde{\phi}_i, \mathsf{body}_\mu\rangle \qquad \langle H,\rho,\pi\rangle \vDash_E \mathsf{acc}(e) \qquad H,\rho \vdash e \Downarrow \texttt{true} \\ \pi' = \lfloor \widetilde{\phi}_i \rfloor_{\pi,H,\rho}\end{array}}{\begin{array}{c}\langle H, \langle \rho, \pi, \texttt{while}\ (e)\ \texttt{inv}\ \widetilde{\phi}_i\ \{\ r\ \};\ s\rangle \cdot S\rangle \\ \widetilde{\longrightarrow} \\ \langle H, \langle \rho, \pi', r;\ \texttt{skip}\rangle \cdot \langle \rho, \pi \backslash \pi', \texttt{while}\ (e)\ \texttt{inv}\ \widetilde{\phi}_i\ \{\ r\ \};\ s\rangle \cdot S\rangle\end{array}}\ \textsc{SsWhileTrue}$$

$$\dfrac{\langle H,\rho,\pi\rangle \,\widetilde{\nvDash}\, \langle \widetilde{\phi}_i \wedge \mathsf{acc}(e), \mathsf{body}_\mu\rangle}{\langle H, \langle \rho, \pi, \texttt{while}\ (e)\ \texttt{inv}\ \widetilde{\phi}_i\ \{\ r\ \};\ s\rangle \cdot S\rangle \,\widetilde{\longrightarrow}\, \textbf{error}}\ \textsc{SsWhileError}$$

$$\dfrac{}{\begin{array}{c}\langle H, \langle \rho', \pi', \texttt{skip}\rangle \cdot \langle \rho, \pi, \texttt{while}\ (e)\ \texttt{inv}\ \widetilde{\phi}_i\ \{\ r\ \};\ s\rangle \cdot S\rangle \\ \widetilde{\longrightarrow} \\ \langle H, \langle \rho', \pi \cup \pi', \texttt{while}\ (e)\ \texttt{inv}\ \widetilde{\phi}_i\ \{\ r\ \};\ s\rangle \cdot S\rangle\end{array}}\ \textsc{SsWhileFinish}$$

$$\dfrac{}{\langle H, \langle \rho, \pi, \texttt{fold}\ p\,(e_1, ..., e_n);\ s\rangle \cdot S\rangle \,\widetilde{\longrightarrow}\, \langle H, \langle \rho, \pi, s\rangle \cdot S\rangle}\ \textsc{SsFold}$$

$$\dfrac{}{\langle H, \langle \rho, \pi, \texttt{unfold}\ p\,(e_1, ..., e_n);\ s\rangle \cdot S\rangle \,\widetilde{\longrightarrow}\, \langle H, \langle \rho, \pi, s\rangle \cdot S\rangle}\ \textsc{SsUnfold}$$

Figure A.10: GVL_{RP}: Small-step semantics adjusted from Fig. A.6 for gradual formulas (continued)

A.2 Chpt. 3's Appendix

```
pc-add(π, t) = Let (id, bc, pcs) :: suffix match π
  (id, bc, pcs ∪ {t}) :: suffix
pc-push(π, id, bc) = (id, bc, ∅) :: π
pc-all(π) = foldl(π, ∅, (λ (id_i, bc_i, pcs_i), all_i . all_i ∪ {bc_i} ∪ pcs_i))
```

Figure A.11: Path condition helper functions

```
eval-p(σ, t, Q)          = Q(σ, t)
eval-p(σ, x, Q)          = Q(σ, σ.γ(x))
eval-p(σ1, op(ē), Q)     = eval-p(σ1, ē, (λ σ2, t̄ . Q(σ2, op′(t̄))))
eval-p(σ1, e.f, Q)       = eval-p(σ1, e, (λ σ2, t .
                            if (∃ f(r; δ) ∈ σ2.h . check(σ2.π, r = t)) then
                              Q(σ2, δ)
                            else if (∃ f(r; δ) ∈ σ2.h? . check(σ2.π, r = t)) then
                              Q(σ2, δ)
                            else if (σ2.isImprecise) then
                              if (σ2.R.origin = (_, unfold acc(_), _)) then
                                e_t := translate(σ2, t)
                                R′ := addcheck(σ2.R, e.f, acc(e_t.f))
                              else
                                R′ := σ2.R
                              δ := fresh
                              Q(σ2{ h? := σ2.h? ∪ f(t; δ), π := pc-add(σ2.π, {t ≠ null}), R := R′ }, δ)
                            else failure())
```

Figure A.12: Rules for symbolically executing expressions without introducing run-time checks (except for a special case for unfold)

In eval-p (Fig. A.12), a special case (highlighted in blue) for unfold statements is added that creates run-time checks for field accesses in the unfolded predicate's body. This case ensures soundness when introducing branch condition variables in C0 programs during run-time verification. In our implementation of Gradual C0, these checks are optimized further as they are only produced for branch conditions in the predicate body rather than for the whole body.

$$
\begin{aligned}
&\textsf{eval-c}(\sigma,\ t,\ Q) &&= Q(\sigma,\ t)\\
&\textsf{eval-c}(\sigma,\ x,\ Q) &&= Q(\sigma,\ \sigma.\gamma(x))\\
&\textsf{eval-c}(\sigma_1,\ op(\overline{e}),\ Q) &&= \textsf{eval-c}(\sigma_1,\ \overline{e},\ (\lambda\ \sigma_2,\ \overline{t}\ .\ Q(\sigma_2,\ op'(\overline{t}))))\\
&\textsf{eval-c}(\sigma_1,\ e.f,\ Q) &&= \textsf{eval-c}(\sigma_1,\ e,\ (\lambda\ \sigma_2,\ t\ .\\
& &&\quad \texttt{if}\ \ (\exists\ f(r;\ \delta) \in \sigma_2.h\ .\ \textsf{check}(\sigma_2.\pi,\ r = t))\ \ \texttt{then}\\
& &&\qquad Q(\sigma_2,\ \delta)\\
& &&\quad \texttt{else if}\ \ (\exists\ f(r;\ \delta) \in \sigma_2.h_?\ .\ \textsf{check}(\sigma_2.\pi,\ r = t))\ \ \texttt{then}\\
& &&\qquad Q(\sigma_2,\ \delta)\\
& &&\quad \texttt{else if}\ \ (\sigma_2.\texttt{isImprecise})\ \ \texttt{then}\\
& &&\qquad res,\ _\ := \textsf{assert}(\sigma_2.\texttt{isImprecise},\ \sigma_2.\pi,\ t \neq \texttt{null})\\
& &&\qquad e_t\ := \textsf{translate}(\sigma_2,\ t)\\
& &&\qquad \mathcal{R}'\ := \textsf{addcheck}(\sigma_2.\mathcal{R},\ e.f,\ \textbf{acc}(e_t.f))\\
& &&\qquad res \wedge Q(\sigma_2\{\ \mathcal{R}\ := \mathcal{R}'\ \},\ \textsf{fresh})\\
& &&\quad \texttt{else}\ \textsf{failure}())
\end{aligned}
$$

Figure A.13: Rules for symbolically executing expressions without modifying the optimistic heap and path condition

A.2.1 Diff and Translate

Algorithm 1 Generating minimal checks

1: **function** DIFF(ϕ)
2: $\quad conjuncts \leftarrow CNF(\phi)$
3: $\quad \phi' \leftarrow \emptyset$
4: $\quad$ **for** $c \leftarrow conjuncts$ **do**
5: $\quad\quad$ **if** $!check(c)$ **then**
6: $\quad\quad\quad \phi' \leftarrow \phi' + c$
7: $\quad\quad$ **end if**
8: $\quad$ **end for**
9: $\quad$ **return** ϕ'
10: **end function**

Figure A.14: Algorithm for computing the diff between two symbolic values

The DIFF (Fig. A.14) function finds a minimal run-time check from an optimistically asserted formula containing statically known information. It accomplishes this by first performing a standard transformation to conjunctive normal form (CNF) on the optimistically asserted formula, to extract the maximal number of top level conjuncts. It then attempts to call *check()* on each conjunct; it accumulates each conjunct for which the call does not succeed. The set of conjuncts which could not be statically discharged are returned as the final check.

Algorithm 2 Variable resolution procedure

```
1: function TRANSLATE-VAR(s, v)
2:     store ← ∅
3:     if s.oldStore then
4:         store ← s.oldStore
5:     else
6:         store ← s.store
7:     end if
8:     aliasList ← aliases(v, s.pathConditions + +s.heap + +s.optimisticHeap)
9:     heap ← s.heap + +s.optimisticHeap
10:    outputs ← ∅
11:    for v ← aliasList do
12:        if c ← store.lookup(v) then
13:            outputs ← outputs + c
14:        else
15:            if h ← heap.lookup(v) && c ← store.lookup(h) then
16:                outputs ← outputs + c
17:            end if
18:        end if
19:    end for
20:    return selectLongest(outputs)
21: end function
```

Figure A.15: TRANSLATE's procedure for resolving variables

The TRANSLATE (Fig. A.15) function lifts symbolic values to concrete values. Most symbolic values are directly translated to their concrete counterparts via recursive descent; the exception is variables, whose concrete values must be reconstructed by searching the program state known by the verifier. This is done by retrieving the states of the symbolic store, which contains mappings from concrete variables to symbolic variables, and the heap, which contains field and predicate permissions. When TRANSLATE encounters a symbolic variable, it first retrieves all possible aliasing information from Gradual Viper's state. This includes all variables known to be equivalent to the translation target according to the path condition and the heap. If the translation target or one of its aliases exists as a value in the symbolic store, then the translator finds a key corresponding to it in the store and returns it. Note that multiple valid keys may exist for a particular symbolic variable, because Gradual Viper may have determined that multiple concrete values are equivalent at a particular program point. If the translation target is a field, then only the top level receiver (the variable on which fields are being accessed) or one of its aliases will exist in the store. The fields being accessed are resolved by mapping their corresponding heap entries, or any aliased heap entries, to a value in the symbolic store, and resolving the store entry as described. In particular contexts, TRANSLATE may be asked to translate a precondition for a

method call, or a predicate body for an (un)fold statement. In these cases, an old store attached to the current symbolic state as described in 3.4.7 is retrieved, and its symbolic store and heap are used for translation. This causes variables in a precondition or predicate to be resolved to their concrete values at the call site, or site of unfolding. This enables run-time checks produced via translate to be straightforwardly emitted to the frontend. The portion of translate related to translating variables is shown in Fig. A.15.

A.2.2 Symbolic production of formulas

The rules for produce are given in Fig. 3.11. Essentially, produce takes a formula and snapshot δ (mirroring the structure of the formula) and adds the information in the formula to the symbolic state, which is then returned to the continuation Q. An imprecise formula `? && ` ϕ has its static part ϕ produced into the current state σ alongside $\textsf{second}(\delta)$. Note the snapshot δ for an imprecise formula looks like $(unit, \textsf{second}(\delta))$ where $unit$ is the snapshot for `?` and $\textsf{second}(\delta)$ is the snapshot for ϕ. An imprecise formula also turns σ imprecise to produce the unknown information represented by `?` into σ. For example, if the state is represented by the formula θ, then this rule results in `? &&` θ `&&` ϕ. A symbolic value t is produced into the path condition of the current state σ. Also, the snapshot δ for t must be $unit$, so this fact is also stored in σ's path condition. Then, σ is passed to Q.

The produce rule for expression e, first evaluates e to its symbolic value t using eval-p. Then, t is produced into the path condition of the current state σ_2 using the aforementioned symbolic value rule. Imprecision in the symbolic state can always provide accessibility predicates for fields also in the state. Therefore, when fields in e are added to an imprecise state, heap chunks for those fields do not have to already be in the state, *e.g.* the state `?&&true` becomes `?&&true &&` e. This functionality is permitted by eval-p. Similarly, an imprecise formula always provides accessibility predicates for fields in its static part, *e.g.* the state `true` and produced formula `? &&` e results in the state `? && true &&` e. The goal of produce is not to assert information in the state, but rather add information to the state. So we reduce run-time overhead by ensuring no run-time checks are produced by produce even for verifying field accesses.

The rules for producing field and predicate accessibility predicates into the state σ_1 operate in a very similar manner. Thus, we will focus on the rule for fields only. The field $e.f$ in $\textbf{acc}(e.f)$ first has its receiver e evaluated to t by eval-pc, resulting in σ_2. Then, using the parameter δ a fresh heap chunk $f(t;\ \delta)$ is created and added to σ_2's heap h, which represents $\textbf{acc}(e.f)$ in the state. Note, the disjoint union $\uplus$ ensures $f(t;\ \delta)$ is not already in the heap before adding $f(t;\ \delta)$ in there. If the chunk is in the heap, then verification will fail. Further, $\textbf{acc}(e.f)$ implies $e \neq \texttt{null}$ and so that fact is recorded in σ_2's path condition as $t \neq \texttt{null}$.

When the separating conjunction $\phi_1 \&\& \phi_2$ is produced, ϕ_1 is first produced and then afterwards ϕ_2 is produced into the resulting symbolic state. Note that the snapshot δ is split between the two formulas using $\textsf{first}(\delta)$ and $\textsf{second}(\delta)$. Finally, to produce a conditional, Gradual Viper branches on the symbolic value t for the condition e splitting execution along two different paths. Along one path only the true branch ϕ_1 is produced into the state, and along the other path only the false branch ϕ_2 is produced. Both paths follow the continuation to the end of its execution. More details about branching are provided next, as we describe Gradual Viper's branch function.

The branch function in Fig. 3.12 is used to split the symbolic execution into two paths in a

number of places in our algorithm: during the production or consumption of logical conditionals and during the execution of if statements. One path (Q_t) is taken under the assumption that the parameter t is true, and the other ($Q_{\neg t}$) is taken under the assumption that t is false. For each path, a branch condition corresponding to the assumption made is added to $\sigma.\mathcal{R}$, as highlighted in blue. Additionally, paths may be pruned using check when Gradual Viper knows for certain a path is infeasible (the assumption about t would contradict the current path conditions). Now, normally, if either of the two paths fail verification, then branch marks verification as failed ($\wedge$ the results). This is still true when σ (the current state) is precise. However, when σ is imprecise, branch can be more permissive as highlighted in yellow. If verification fails on one of two paths only (one success, one failure), then branch returns success ($\vee$ the results). In this case, a run-time check (highlighted in blue) is added to $\mathfrak{R}$ to force run-time execution down the success path only. Of course, two failures result in failure and two successes result in success ($\vee$ the results). No run-time checks are produced in these cases, as neither path can be soundly taken or both paths can be soundly taken at run time respectively. Note that Gradual Viper being flexible in the aforementioned way is critical to adhering to the gradual guarantee at branch points.

A.2.3 Symbolic consumption of formulas

$$
\begin{aligned}
\text{consume}(\sigma_1,\ \theta,\ Q) \quad &= \sigma_2 \ := \ \sigma_1\{\ h,\ \pi\ := \text{consolidate}(\sigma_1.h,\ \sigma_1.\pi)\ \}\\
&\quad \text{consume}'(\sigma_2,\ \sigma_2.\mathtt{isImprecise},\ \sigma_2.h_?,\ \sigma_2.h,\ \theta,\ (\lambda\ \sigma_3,\ h'_?,\ h_1,\ \delta_1\ .\\
&\quad\quad Q(\sigma_3\{\ h_?\ := h'_?,\ h\ := h_1\},\ \delta_1)))\\
\text{consume}(\sigma_1,\ ?\ \&\&\ \phi,\ Q) &= \sigma_2 \ := \ \sigma_1\{\ h,\ \pi\ := \text{consolidate}(\sigma_1.h,\ \sigma_1.\pi)\ \}\\
&\quad \text{consume}'(\sigma_2,\ \mathtt{true},\ \sigma_2.h_?,\ \sigma_2.h,\ \phi,\ (\lambda\ \sigma_3,\ h'_?,\ h_1,\ \delta_1\ .\\
&\quad\quad Q(\sigma_3\{\ \mathtt{isImprecise}\ := \mathtt{true},\ h_?\ := \emptyset,\ h\ := \emptyset\ \},\ pair(unit,\ \delta_1))))
\end{aligned}
$$

Handles imprecision | Handles run-time check generation and collection

Figure A.16: Rules for symbolically consuming formulas (1/3)

The goals of consume are 3-fold: 1) given a symbolic state σ and formula $\widetilde{\phi}$ check whether $\widetilde{\phi}$ is established by σ, *i.e.* $\widetilde{\phi}_\sigma \;\widetilde{\Rightarrow}\; \widetilde{\phi}$ where $\widetilde{\phi}_\sigma$ is the formula which represents the state σ, 2) produce and collect run-time checks that are minimally sufficient for σ to establish $\widetilde{\phi}$ soundly, and 3) remove accessibility predicates and predicates that are asserted in $\widetilde{\phi}$ from σ. Note that $\widetilde{\Rightarrow}$ is the consistent implication formally defined in Chpt. 2. The rules for consume are given in Fig. A.16.

The consume function always begins by consolidating information across the given heap $\sigma_1.h$ and path condition $\sigma_1.\pi$. The invariant on the heap $\sigma_1.h$ ensures all heap chunks in $\sigma_1.h$ are separated in memory, *e.g.* $f\,(x;\ \delta_1) \in \sigma_1.h$ and $f\,(y;\ \delta_2) \in \sigma_1.h$ implies $x \neq y$. Similarly, $f\,(x;\ \delta_1) \in \sigma_1.h$ implies $x \neq \mathtt{null}$. Therefore, such information is added to the path condition $\sigma_1.\pi$ during consolidation. Further, consolidate ensures $\sigma_1.h$ and $\sigma_1.\pi$ are consistent, *i.e.* do not

$$
\begin{aligned}
\text{consume'}(\sigma,\ f_?,\ h_?,\ h,\ (e,t),\ Q) &= res,\ \bar{t} := \text{assert}(\sigma.\texttt{isImprecise},\ \sigma.\pi,\ t) \\
&\quad \mathcal{R}' := \text{addcheck}(\sigma.\mathcal{R},\ e,\ \text{translate}(\sigma,\ \bar{t})) \\
&\quad res \wedge Q(\sigma\{\mathcal{R} := \mathcal{R}'\},\ h_?,\ h,\ unit)
\end{aligned}
$$

$$
\begin{aligned}
\text{consume'}(\sigma_1,\ f_?,\ h_?,\ h,\ e,\ Q) &= \text{eval-c}(\sigma_1\{\texttt{isImprecise} := f_?\},\ e,\ (\lambda\ \sigma_2,\ t\ . \\
&\quad \text{consume'}(\sigma_2\{\texttt{isImprecise} := \sigma_1.\texttt{isImprecise}\},\ f_?,\ h_?,\ h,\ (e,t),\ Q)))
\end{aligned}
$$

$$
\begin{aligned}
\text{consume'}(\sigma_1,\ f_?,\ h_?,\ h,\ \textbf{acc}(p(\bar{e})),\ Q) &= \text{eval-c}(\sigma_1\{\texttt{isImprecise} := f_?\},\ \bar{e},\ (\lambda\ \sigma_2,\ \bar{t}\ . \\
&\quad \sigma_3 := \sigma_2\{\texttt{isImprecise} := \sigma_1.\texttt{isImprecise}\} \\
&\quad (h_1,\ \delta_1,\ b_1) := \text{heap-rem-pred}(\sigma_3.\texttt{isImprecise},\ h,\ \sigma_3.\pi,\ p(\bar{t})) \\
&\quad \texttt{if}\ (\sigma_3.\texttt{isImprecise})\ \texttt{then} \\
&\quad\quad (h'_?,\ \delta_2,\ b_2) := \text{heap-rem-pred}(\sigma_3.\texttt{isImprecise},\ h_?,\ \sigma_3.\pi,\ p(\bar{t})) \\
&\quad\quad \texttt{if}\ (b_1 = b_2 = \texttt{false})\ \texttt{then} \\
&\quad\quad\quad \mathcal{R}' := \text{addcheck}(\sigma_3.\mathcal{R},\ \textbf{acc}(p(\bar{e})),\ \textbf{acc}(p(\bar{e}))) \\
&\quad\quad \texttt{else}\ \mathcal{R}' := \sigma_3.\mathcal{R} \\
&\quad\quad Q(\sigma_3\{\mathcal{R} := \mathcal{R}'\},\ h'_?,\ h_1,\ (\texttt{if}\ (b_1)\ \texttt{then}\ \delta_1\ \texttt{else}\ \delta_2)) \\
&\quad \texttt{else if}\ (b_1)\ \texttt{then}\ Q(\sigma_3,\ \sigma_3.h_?,\ h_1,\ \delta_1) \\
&\quad \texttt{else}\ \text{failure}()))
\end{aligned}
$$

$$
\begin{aligned}
\text{consume'}(\sigma_1,\ f_?,\ h_?,\ h,\ \textbf{acc}(e.f),\ Q) &= \text{eval-c}(\sigma_1\{\texttt{isImprecise} := f_?\},\ e,\ (\lambda\ \sigma_2,\ t\ . \\
&\quad \sigma_3 := \sigma_2\{\texttt{isImprecise} := \sigma_1.\texttt{isImprecise}\} \\
&\quad res,\ \bar{t} := \text{assert}(\sigma_3.\texttt{isImprecise},\ \sigma_3.\pi,\ t \neq \texttt{null}) \\
&\quad res\ \wedge\ (\\
&\quad \mathcal{R}' := \text{addcheck}(\sigma_3.\mathcal{R},\ \textbf{acc}(e.f),\ \text{translate}(\sigma_3,\ \bar{t})) \\
&\quad (h_1,\ \delta_1,\ b_1) := \text{heap-rem-acc}(\sigma_3.\texttt{isImprecise},\ h,\ \sigma_3.\pi,\ f(t)) \\
&\quad \texttt{if}\ (\sigma_3.\texttt{isImprecise})\ \texttt{then} \\
&\quad\quad (h'_?,\ \delta_2,\ b_2) := \text{heap-rem-acc}(\sigma_3.\texttt{isImprecise},\ h_?,\ \sigma_3.\pi,\ f(t)) \\
&\quad\quad \texttt{if}\ (b_1 = b_2 = \texttt{false})\ \texttt{then} \\
&\quad\quad\quad \mathcal{R}'' := \text{addcheck}(\mathcal{R}',\ \textbf{acc}(e.f),\ \textbf{acc}(\text{translate}(\sigma_3,\ t).f)) \\
&\quad\quad \texttt{else}\ \mathcal{R}'' := \mathcal{R}' \\
&\quad\quad Q(\sigma_3\{\mathcal{R} := \mathcal{R}''\},\ h'_?,\ h_1,\ (\texttt{if}\ (b_1)\ \texttt{then}\ \delta_1\ \texttt{else}\ \delta_2)) \\
&\quad \texttt{else if}\ (b_1)\ \texttt{then}\ Q(\sigma_3\{\mathcal{R} := \mathcal{R}'\},\ \sigma_3.h_?,\ h_1,\ \delta_1) \\
&\quad \texttt{else}\ \text{failure}())))
\end{aligned}
$$

Handles imprecision | Handles run-time check generation and collection

Figure A.16: Rules for symbolically consuming formulas (2/3)

$$\text{consume'}(\sigma_1,\ f_?,\ h_?,\ h,\ \phi_1 \ \&\&\ \phi_2,\ Q) = \text{consume'}(\sigma_1,\ f_?,\ h_?,\ h,\ \phi_1,\ (\lambda\ \sigma_2,\ h'_?,\ h',\ \delta_1\ .$$
$$\text{consume'}(\sigma_2,\ f_?,\ h'_?,\ h',\ \phi_2,\ (\lambda\ \sigma_3,\ h''_?,\ h'',\ \delta_2\ .$$
$$Q(\sigma_3,\ h''_?,\ h'',\ pair(\delta_1,\delta_2)))))$$

$$\text{consume'}(\sigma_1,\ f_?,\ h_?,\ h,\ e\ ?\ \phi_1\ :\ \phi_2,\ Q) = \text{eval-c}(\sigma_1\{\texttt{isImprecise} := f_?\},\ e,\ (\lambda\ \sigma_2,\ t\ .$$
$$\sigma_3 := \sigma_2\{\texttt{isImprecise} := \sigma_1.\texttt{isImprecise}\}$$
$$\text{branch}(\sigma_3,\ e,\ t,$$
$$(\lambda\ \sigma_4\ .\ \text{consume'}(\sigma_4,\ f_?,\ h_?,\ h,\ \phi_1,\ Q)),$$
$$(\lambda\ \sigma_4\ .\ \text{consume'}(\sigma_4,\ f_?,\ h_?,\ h,\ \phi_2,\ Q)))))$$

Handles imprecision | Handles run-time check generation and collection

Figure A.16: Rules for symbolically consuming formulas (3/3)

contain contradictory information. We use the definition of consolidate from Schwerhoff [39], without repeating it here.

After consolidation, consume calls a helper function consume', which performs the major functionality of consume. Along with the state σ_2 from consolidation, consume' accepts a boolean flag, optimistic heap $\sigma_2.h_?$, regular heap $\sigma_2.h$, the formula to be consumed $\widetilde{\phi}$, and a continuation. The boolean flag sent to consume' controls how σ_2 provides access to fields in $\widetilde{\phi}$. When $\widetilde{\phi}$ is precise (is θ), then σ_2 provides access to fields in θ through heap chunks or imprecision where applicable. Therefore, in this case, the boolean flag is set to $\sigma_2.\texttt{isImprecise}$. However, when $\widetilde{\phi}$ is imprecise (*i.e.* `? && ` ϕ), then the boolean flag is set to `true` so access to fields in $\widetilde{\phi}$ is always justified: first by σ_2 if applicable and second by imprecision in $\widetilde{\phi}$. Copies of the optimistic heap $\sigma_2.h_?$ and regular heap $\sigma_2.h$ are sent to consume' where heap chunks from $\widetilde{\phi}$ are removed from them. If consume' succeeds, then when $\widetilde{\phi}$ is precise execution continues with the residual heap chunks. When $\widetilde{\phi}$ is imprecise execution continues with empty heaps, because $\widetilde{\phi}$ may require and assert any heap chunk in σ_2. Residual heap chunks are instead represented by imprecision, *i.e.* execution continues with an imprecise state. Finally, consume' also sends snapshots collected for removed heap chunks to the continuation.

Rules for consume' can also be found in Fig. A.16. Cases for expressions e, the separating conjunction ϕ_1 `&&` ϕ_2, and logical conditionals e `?` ϕ_1 `:` ϕ_2 are straightforward. Expressions are evaluated to symbolic values that are then consumed with the corresponding rule. In a separating conjunction, ϕ_1 is consumed first, then afterward ϕ_2 is consumed. The rule for logical conditionals evaluates the condition e to a symbolic value, and then uses the branch function to consume ϕ_1 and ϕ_2 along different execution paths. The case for **acc**$(p(\overline{e}))$ is also very similar to the case for **acc**$(e.f)$ that we discuss later in this section.

When a symbolic value t is consumed, the current state σ must establish t, *i.e.* $\sigma \widetilde{\Rrightarrow} t$, or verification fails. The assert function (defined in Fig. A.18) implements this functionality. In particular, assert returns success() when π can statically prove t or when σ is imprecise and t does not contradict constraints in π—here, t is optimistically assumed to be true. Otherwise, assert returns failure(). When assert succeeds, it also returns a set of symbolic values $\overline{t}$ that are

residuals of t that cannot be proved statically by π. If t is proven entirely statically, then assert returns `true`. A run-time check is created for the residuals $\bar{t}$ and is added to σ to be passed to the continuation Q. Note that translate is used to create an expression from $\bar{t}$ that can be evaluated at run time. Further, the location e is the expression that evaluates to t and is passed to consume' alongside t. The heaps $h_?$ and h are passed unmodified to Q alongside the snapshot $unit$.

The consume' rule for accessibility predicates **acc**$(e.f)$, first evaluates the receiver e to t using eval-c, the current state σ_1, and the parameter $f_?$. The parameter $f_?$ is the boolean flag mentioned previously. Assigning $f_?$ to σ_1`.isImprecise` during evaluation allows $f_?$ to control whether or not imprecision verifies field accesses. This occurs in all of the consume' rules where expressions and thus fields are evaluated. After evaluation, the `isImprecise` field is reset resulting in σ_3, and assert is used to ensure the receiver t is non-null. If $t \neq$ `null` is optimistically true, a run-time check for $t \neq$ `null` at location **acc**$(e.f)$ is created and added to $\sigma_3.\mathcal{R}$. Next, heap-rem-acc is used to remove the heap chunks from heap h that overlap with or may potentially overlap with **acc**$(e.f)$ in memory. The heap-rem-acc function is formally defined alongside a similar function for predicates (heap-rem-pred) in the Fig. A.17. If a field chunk is not statically proven to be disjoint from **acc**$(e.f)$, then it is removed. Further, since predicates are opaque, Gradual Viper cannot tell whether or not their predicate bodies overlap with **acc**$(e.f)$. Therefore, predicate chunks are almost always considered to potentially overlap with **acc**$(e.f)$. The only time this is not the case is if they both exist in the heap h, which ensures its heap chunks do not overlap in memory. The heap-rem-acc function also checks that **acc**$(e.f)$ has a corresponding heap chunk in h. If so, its snapshot δ_1 is returned and b_1 is assigned `true`. Otherwise, a fresh snapshot is returned with `false`. If the current state σ_3 is imprecise, then heap chunks are similarly removed from $h_?$ and **acc**$(e.f)$ is checked for existence in $h_?$. If a field chunk for **acc**$(e.f)$ is not found in either heap, then a run-time check is generated for it and passed to the continuation Q alongside the two heaps after removal and **acc**$(e.f)$'s snapshot. Without imprecision, consume' will fail when a field chunk for **acc**$(e.f)$ is not found in h.

A.2.4 Symbolic execution of statements

The exec rules for sequence statements, variable declarations and assignments, allocations, and if statements are pretty much unchanged from Viper. The only difference is that Gradual Viper's versions of eval, produce, branch, and consume (defined previously) are used instead of Viper's. Statements in a sequence are executed one after another, and variable declarations introduce a fresh symbolic value for the variable into the state. Variable assignments evaluate the right-hand side to a symbolic value and update the variable in the symbolic store with the result. Allocations produce fresh heap chunks for fields into the state. Finally, if statements have their condition evaluated and then branch is used to split execution along two paths to symbolically execute the true and false branches.

Symbolic execution of field assignments first evaluates the right-hand side expression e to the symbolic value t with the current state σ_1 and eval. Any field reads in e are either directly or optimistically verified using σ_1. Then, the resulting state σ_2 must establish write access to $x.f$ in consume, *i.e.* $\sigma_2 \widetilde{\Rrightarrow}$ **acc**$(x.f)$. The call to consume also removes the field chunk for **acc**$(x.f)$ from σ_2 (if it is in there) resulting in σ_3. Therefore, the call to produce can safely add a fresh field

$$
\begin{aligned}
&\textsf{heap-rem-pred}(\texttt{isImprecise},\ h,\ \pi,\ p(\overline{t})) = \texttt{if}\ \ \exists\ (p(\overline{r};\ \delta) \in h\ .\ \textsf{check}(\pi,\ \bigwedge \overline{t = r}))\ \ \texttt{then}\\
&\qquad (h \setminus \{p(\overline{r};\ \delta)\},\ \delta,\ \texttt{true})\\
&\qquad \texttt{else}\ \ (\emptyset,\ \textsf{fresh},\ \texttt{false})\\
&\textsf{heap-rem-acc}(\texttt{isImprecise},\ h,\ \pi,\ f(t)) = h' := \textsf{foldl}(h,\ \emptyset,\ (\lambda\ f_{src}(\overline{r};\ \delta),\ h_{dst}\ .\\
&\qquad\quad \texttt{if}\ \ (\neg(|\overline{r}| = 1) \parallel \neg(f = f_{src}) \parallel \neg\textsf{check}(\texttt{isImprecise},\ \pi,\ t = r))\ \texttt{then}\\
&\qquad\qquad h_{dst} \cup f_{src}(\overline{r};\ \delta)\\
&\qquad\quad \texttt{else}\ \ h_{dst}))\\
&\qquad \texttt{if}\ \ \exists\ f(r;\ \delta) \in h\ .\ \textsf{check}(\pi,\ t = r)\ \ \texttt{then}\\
&\qquad\quad (h',\ \delta,\ \texttt{true})\\
&\qquad \texttt{else}\\
&\qquad\quad h' := \textsf{foldl}(h',\ \emptyset,\ (\lambda\ f_{src}(\overline{r};\ \delta),\ h_{dst}\ .\\
&\qquad\qquad \texttt{if}\ \ (f_{src}(\overline{r};\ \delta)\ \ \texttt{is a field chunk})\ \ \texttt{then}\\
&\qquad\qquad\quad h_{dst} \cup f_{src}(\overline{r};\ \delta)\\
&\qquad\qquad \texttt{else}\ \ h_{dst}))\\
&\qquad\quad (h',\ \textsf{fresh},\ \texttt{false})
\end{aligned}
$$

Figure A.17: Heap remove function definitions

$$
\begin{aligned}
&\textsf{check}(\pi,\ t) = \textsf{pc-all}(\pi) \Rightarrow t\\
&\textsf{check}(\texttt{isImprecise},\ \pi,\ t) = \begin{cases} \texttt{true},\ \ \texttt{true} & \texttt{if}\ \textsf{check}(\pi,\ t)\\ \texttt{true},\ \ \textsf{diff}(\textsf{pc-all}(\pi), t) & \texttt{if}\ \ (\texttt{isImprecise} \wedge (\bigwedge \textsf{pc-all}(\pi) \wedge t)_{\textsc{SAT}})\\ \texttt{false},\ \ \emptyset & \texttt{otherwise} \end{cases}\\
&\textsf{assert}(\texttt{isImprecise},\ \pi,\ t) = \begin{cases} \textsf{success}(),\ \ \overline{t} & \texttt{if}\ \ (b = \texttt{true})\ \ \texttt{where}\ \ b,\ \overline{t} := \textsf{check}(\texttt{isImprecise},\ \pi,\ t)\\ \textsf{failure}(),\ \ \emptyset & \texttt{otherwise} \end{cases}
\end{aligned}
$$

Handles imprecision	Handles run-time check generation and collection

Figure A.18: Check and assert function definitions

chunk for **acc** $(x.f)$ alongside $x.f = t$ to σ_3 before it is passed to the continuation Q. Under the hood, run-time checks are collected where required for soundness and passed to Q.

The exec rule for method calls similarly uses eval to evaluate the given args $\overline{e}$ to symbolic values $\overline{t}$, asserts the method's precondition $meth_{pre}$ holds in the current state, consumes the heap chunks in the precondition, and produces the method's postcondition $meth_{post}$ into the continuation. Run-time checks are also collected where necessary (under the hood) and passed to the continuation. Gradual Viper makes an exception when consuming preconditions at method calls (and loop invariants before entering loops), which can be seen in the `if-then` in the method call rule. If Gradual Viper determines the precondition (invariant) is equi-recursively imprecise, then it will conservatively remove all the heap chunks from both symbolic heaps after the consume. This exception ensures the static verification semantics in Gradual Viper lines up with the equi-recursive, dynamic verification semantics encoded by GVC0 in §3.5 such that Gradual C0 is sound. Note that the `origin` field of $\mathcal{R}$ is set to $\overline{z} := m(\overline{e})$ before consuming $meth_{pre}$

$$\mathsf{exec}(\sigma_1,\ s_1;\ s_2,\ Q) = \mathsf{exec}(\sigma_1,\ s_1,\ (\lambda\ \sigma_2\ .\ \mathsf{exec}(\sigma_2,\ s_2,\ Q)))$$

$$\mathsf{exec}(\sigma,\ \mathtt{var}\ x:T,\ Q) = Q(\sigma\{\gamma\ := \mathsf{havoc}(\sigma.\gamma,\ x)\})$$

$$\mathsf{exec}(\sigma_1,\ x\ := e,\ Q) = \mathsf{eval}(\sigma_1,\ e,\ (\lambda\ \sigma_2,\ t\ .\ Q(\sigma_2\{\gamma\ := \sigma_2.\gamma[x \mapsto t]\})))$$

$$\mathsf{exec}(\sigma_1,\ x.f\ := e,\ Q) = \mathsf{eval}(\sigma_1,\ e,\ (\lambda\ \sigma_2,\ t\ .\ \mathsf{consume}(\sigma_2,\ \mathbf{acc}(x.f),\ (\lambda\ \sigma_3,\ _\ .\ \mathsf{produce}(\sigma_3,\ \mathbf{acc}(x.f)\ \&\&\ x.f = t,\ \mathit{pair}(\mathsf{fresh},\ \mathit{unit}),\ Q)))))$$

$$\mathsf{exec}(\sigma,\ x\ := \mathtt{new}(\overline{f}),\ Q) = \mathsf{produce}(\sigma\{\gamma\ := \mathsf{havoc}(\sigma.\gamma,\ x)\},\ \overline{\mathbf{acc}(x.f)},\ \mathsf{fresh},\ Q)$$

$$\begin{aligned}\mathsf{exec}(\sigma_1,\ \overline{z}\ := m(\overline{e}),\ Q) = \ & \mathsf{eval}(\sigma_1,\ \overline{e},\ (\lambda\ \sigma_2,\ \overline{t}\ .\\ & \mathcal{R}'\ := \sigma_2.\mathcal{R}\{\mathtt{origin}\ := (\sigma_2,\ \overline{z}\ := m(\overline{e}),\ \overline{t})\}\\ & \mathsf{consume}(\sigma_2\{\mathcal{R}\ := \mathcal{R}'\},\ \mathit{meth}_{pre}[\overline{\mathit{meth}_{args} \mapsto t}],\ (\lambda\ \sigma_3,\ \delta.\\ & \quad \mathtt{if}\ \ (\mathsf{equi\text{-}imp}(\mathit{meth}_{pre}))\ \ \mathtt{then}\\ & \quad\quad \sigma_4\ := \sigma_3\{\mathtt{isImprecise}\ := \mathtt{true},\ h_?\ := \emptyset,\ h\ := \emptyset,\\ & \quad\quad\quad \gamma\ := \mathsf{havoc}(\sigma_3.\gamma,\ \overline{z})\}\\ & \quad \mathtt{else}\ \ \sigma_4\ := \sigma_3\{\gamma\ := \mathsf{havoc}(\sigma_3.\gamma,\ \overline{z})\}\\ & \quad \mathsf{produce}(\sigma_4,\ \mathit{meth}_{post}[\overline{\mathit{meth}_{args} \mapsto t}][\overline{\mathit{meth}_{ret} \mapsto z}],\ \mathsf{fresh},\\ & \quad\quad (\lambda\ \sigma_5\ .\ Q(\sigma_5\{\mathcal{R}\ := \sigma_5.\mathcal{R}\{\mathtt{origin}\ := \mathtt{none}\}\}))))))\end{aligned}$$

$$\begin{aligned}\mathsf{exec}(\sigma_1,\ \mathtt{assert}\ \phi,\ Q) = \ & \mathsf{consume}(\sigma_1,\ \phi,\ (\lambda\ \sigma_2,\ \delta\ .\\ & \mathsf{well\text{-}formed}(\sigma_2,\ ?\ \&\&\ \phi,\ \delta,\ (\lambda\ \sigma_3\ .\\ & \quad Q(\sigma_1\{\pi\ := \sigma_3.\pi,\ \mathcal{R}\ := \sigma_3.\mathcal{R}\})))))\end{aligned}$$

$$\begin{aligned}\mathsf{exec}(\sigma_1,\ \mathtt{fold}\ \mathbf{acc}(p(\overline{e})),\ Q) = \ & \mathsf{eval}(\sigma_1,\ \overline{e},\ (\lambda\ \sigma_2,\ \overline{t}\ .\\ & \mathcal{R}'\ := \sigma_2.\mathcal{R}\{\mathtt{origin}\ := (\sigma_2,\ \mathtt{fold}\ \mathbf{acc}(p(\overline{e})),\ \overline{t})\}\\ & \mathsf{consume}(\sigma_2\{\mathcal{R}\ := \mathcal{R}'\},\ \mathit{pred}_{body}[\overline{\mathit{pred}_{args} \mapsto t}],\ (\lambda\ \sigma_3,\ \delta.\\ & \quad \mathsf{produce}(\sigma_3\{\mathcal{R}\ := \sigma_3.\mathcal{R}\{\mathtt{origin}\ := \mathtt{none}\}\},\ \mathbf{acc}(p(\overline{t})),\\ & \quad \delta,\ Q)))))\end{aligned}$$

$$\begin{aligned}\mathsf{exec}(\sigma_1,\ \mathtt{unfold}\ \mathbf{acc}(p(\overline{e})),\ Q) = \ & \mathsf{eval}(\sigma_1,\ \overline{e},\ (\lambda\ \sigma_2,\ \overline{t}\ .\\ & \mathcal{R}'\ := \sigma_2.\mathcal{R}\{\mathtt{origin}\ := (\sigma_2,\ \mathtt{unfold}\ \mathbf{acc}(p(\overline{e})),\ \overline{t})\}\\ & \mathsf{consume}(\sigma_2\{\mathcal{R}\ := \mathcal{R}'\},\ \mathbf{acc}(p(\overline{t})),\ (\lambda\ \sigma_3,\ \delta\ .\\ & \quad \mathsf{produce}(\sigma_3,\ \mathit{pred}_{body}[\overline{\mathit{pred}_{args} \mapsto t}],\ \delta,\ (\lambda\ \sigma_4\ .\\ & \quad Q(\sigma_4\{\mathcal{R}\ := \sigma_4.\mathcal{R}\{\mathtt{origin}\ := \mathtt{none}\}\})))))))\end{aligned}$$

$$\begin{aligned}\mathsf{exec}(\sigma_1,\ \mathtt{if}\ (e)\ \{\ \mathit{stmt}_1\ \}\ \mathtt{else}\ \{\ \mathit{stmt}_2\ \},\ Q) = \ & \mathsf{eval}(\sigma_1,\ e,\ (\lambda\ \sigma_2,\ t\ .\\ & \mathsf{branch}(\sigma_2,\ e,\ t,\ (\lambda\ \sigma_3\ .\ \mathsf{exec}(\sigma_3,\ \mathit{stmt}_1,\ Q)),\ (\lambda\ \sigma_3\ .\ \mathsf{exec}(\sigma_3,\ \mathit{stmt}_2,\ Q))))\end{aligned}$$

Legend: ■ Handles imprecision ■ Handles run-time check generation and collection

Figure A.19: Rules for symbolically executing program statements (1/2)

```
exec(σ1, while (e) invariant φ̃ { stmt }, Q) =  γ2 := havoc(σ1.γ, x̄)
        res_body  :=  well-formed (
          σ1{ isImprecise := false, h? := ∅ , h := ∅, γ := γ2,
            R := σ1.R{origin := (σ1, while (e) invariant φ̃ { stmt }, beginning)} },
          φ̃ && e , fresh, (λ σ3{R := σ3.R{origin := none}} .
            exec(σ3, stmt, (λ σ4 .
              eval ( σ4{R := σ4.R{origin := (σ4, while (e) invariant φ̃ { stmt }, end)}} ,
                e, (λ σe, _ .
                  consume ( σ4{R := σ4.R{origin := σe.R.origin, rcs := σe.R.rcs}} ,
                    φ̃ , (λ σ5, _ . ℜ := ℜ ∪ σ5.R.rcs ; success()))))))))
        res_after  :=  eval ( σ1{R := σ1.R{origin := (σ1, while (e) invariant φ̃ { stmt }, before)}} ,
          e, (λ σe, _ . consume (
            σ1{R := σ1.R{origin := σe.R.origin, rcs := σe.R.rcs}} ,
            φ̃ , (λ σ2, _ .
              if  (equi-imp(φ̃))  then
                σ3 := σ2{isImprecise := true, h? := ∅, h := ∅, γ := γ2}
              else  σ3 := σ2{γ := γ2}
              produce (σ3{ σ3.R{origin := (σ3, while (e) invariant φ̃ { stmt }, after)} },
                φ̃ && !e , fresh, Q)))))
        if (σ1.isImprecise) then
          if (¬res_body ∧ res_after) then
            R′ := addcheck(σ1.R, e, ¬e)
            ℜ := ℜ ∪ R′.rcs.last
          (¬res_body ∨ res_after) ∧ (res_body ∨ res_after)
        else
          res_body ∧ res_after
        where x̄ are variables modified by the loop body
```

Handles imprecision	Handles run-time check generation and collection

Figure A.19: Rules for symbolically executing program statements (2/2)

```
equi-imp(? && φ)          = true
equi-imp(φ1 && φ2)        = equi-imp(φ1) ∨ equi-imp(φ2)
equi-imp(e ? φ1 : φ2)     = equi-imp(φ1) ∨ equi-imp(φ2)
equi-imp(acc(p(ē)))       = if  (p ∈ VisitedPreds)  then
                              false
                            else
                              VisitedPreds := VisitedPreds ∪ p
                              b := equi-imp(pred_body)
                              VisitedPreds := VisitedPreds \ p
                              b
equi-imp(_)               = false
```

Figure A.20: Boolean function determining if a gradual formula is equi-recursively imprecise or not

$$\text{well-formed}(\sigma_1,\ \widetilde{\phi},\ \delta,\ Q) = \text{produce}(\sigma_1,\ \widetilde{\phi},\ \delta,\ (\lambda\ \sigma_2\ .\ \text{produce}(\sigma_1\{\ \pi\ :=\ \sigma_2.\pi\ \},\ \widetilde{\phi},\ \delta,\ Q)))$$

Figure A.21: Well-formed formula function definition

and reset to `none` after producing $meth_{post}$. Setting the origin indicates that run-time checks or branch conditions for $meth_{pre}$ or $meth_{post}$ should be attached to the method call statement rather than where they are declared. The origin arguments σ_2 and $\overline{t}$ are used to reverse the substitution $\overline{[meth_{args} \mapsto t]}$ in run-time checks and branch conditions for $meth_{pre}$ and $meth_{post}$. The rule for (un)folding predicates operates the same as for method calls where $meth_{pre}$ is the predicate body (predicate instance) and $meth_{post}$ is the predicate instance (predicate body). The `origin` is set to `fold` **acc**$(p(\overline{e}))$ and `unfold` **acc**$(p(\overline{e}))$ respectively.

In contrast, ϕ in `assert` ϕ maintains a `none` origin field, because ϕ's use and declaration align at the same program location `assert` ϕ. The assert rule relies on consume to assert ϕ holds in the current state σ_1. If the consume succeeds, the state σ_1 is passed to the continuation nearly unmodified. Path condition constraints from ϕ hold in σ_1 either directly or optimistically. Therefore, these constraints are added to σ_1 to avoid producing run-time checks for them in later program statements. Run-time checks from the consume are also passed to the continuation. Note that ϕ is checked for well-formedness here (Fig. A.21). A formula is well-formed if it contains `?` or accessibility predicates that verify access to the formula's fields (*self-framing*). Additionally, the formula cannot contain duplicate accessibility predicates or predicate instances. Finally, well-formed adds the formula's information to the given symbolic state. Here, ϕ does not need to be self-framed, and so it is joined with `?` in the call to well-formed. `?` verifies access to all of ϕ's fields.

Finally, while the while loop rule is the largest rule and looks fairly complex, it just combines ideas from other rules that are discussed in great detail in this section and from the branch rule

described in §A.2.2.

A.2.5 Valid program

A Gradual Viper program is valid if all of its method and predicate declarations are verified successfully as defined in Fig. 3.15. In particular, a method m's declaration is verified first by checking well-formedness of m's precondition $meth_{pre}$ and postcondition $meth_{post}$ using the empty state σ_0 (well-formedness is described in §A.2.4). Note, fresh symbolic values are created and added to σ_0 for m's argument variables $\overline{x}$ and return variables $\overline{y}$. If $meth_{pre}$ and $meth_{post}$ are well-formed, then the body of m ($meth_{body}$) is symbolically executed (§A.2.4) starting with the symbolic state σ_1 containing $meth_{pre}$. Recall, well-formed additionally produces the formula that is being checked into the symbolic state. The symbolic state σ_2 is produced after the symbolic execution of $meth_{body}$. Then, $meth_{post}$ is checked for validity against σ_2, *i.e.* σ_2 must establish $meth_{post}$ (§A.2.3). If $meth_{post}$ is established, then verification succeeds; and as a result, the run-time checks collected during verification are added to $\mathfrak{R}$ (highlighted in blue). A valid predicate p is simply valid if p's body $pred_{body}$ is well-formed. As before, fresh symbolic values are created for p's argument variables $\overline{x}$. Note, no run-time checks are added to $\mathfrak{R}$ here, because well-formedness checks do not produce any run-time checks.

Bibliography

[1] Pieter Agten, Bart Jacobs, and Frank Piessens. Sound modular verification of C code executing in an unverified context. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 581–594, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450333009. doi: 10.1145/2676726.2676972. URL `https://doi.org/10.1145/2676726.2676972`. 1.6.3

[2] Rob Arnold. C0, an imperative programming language for novice computer scientists. Master's thesis, Department of Computer Science, Carnegie Mellon University, 2010. 1.2, 1.4, 3.1, 3.2

[3] Johannes Bader, Jonathan Aldrich, and Éric Tanter. Gradual program verification. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 25–46. Springer, 2018. 1, 1.3.1, 1.6, 2, 2.1.2, 2.1.2, 2.1.5, 2.2.3, 2.2.3, 2.4, 2.4

[4] Josh Berdine, Cristiano Calcagno, and Peter W O'hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures 4*, pages 115–137. Springer, 2006. 1.6.2, 3

[5] Sam Blackshear and Shuvendu K Lahiri. Almost-correct specifications: A modular semantic framework for assigning confidence to warnings. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 209–218, 2013. 1.6.2

[6] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 289–300, 2009. 1.6.2

[7] John Peter Campora, Sheng Chen, and Eric Walkingshaw. Casts and costs: harmonizing safety and performance in gradual typing. 2(ICFP):98:1–98:30, September 2018. 4

[8] Satish Chandra, Stephen J Fink, and Manu Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 363–374, 2009. 1.6.2

[9] Ankush Das, Shuvendu K Lahiri, Akash Lal, and Yi Li. Angelic verification: Precise verification modulo unknowns. In *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I 27*, pages 324–

342. Springer, 2015. 1.6.2

[10] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975. ISSN 0001-0782. doi: 10.1145/360933.360975. URL `http://doi.acm.org/10.1145/360933.360975`. 2.1.3

[11] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975. 1

[12] Isil Dillig, Thomas Dillig, and Alex Aiken. Automated error diagnosis using abductive inference. *ACM SIGPLAN Notices*, 47(6):181–192, 2012. 1.6.2

[13] Dino Distefano and Matthew J Parkinson J. jStar: Towards practical verification for Java. *ACM Sigplan Notices*, 43(10):213–226, 2008. 1.6.2, 3

[14] Jenna DiVincenzo, Ian McCormack, Hemant Gouni, Jacob Gorenburg, Mona Zhang, Conrad Zimmerman, Joshua Sunshine, Éric Tanter, and Jonathan Aldrich. Gradual C0: Symbolic execution for efficient gradual verification, 2022. URL `https://arxiv.org/abs/2210.02428`. 1.2, 1.2

[15] Joseph Eremondi, Éric Tanter, and Ronald Garcia. Approximate normalization for gradual dependent types. *Proc. ACM Program. Lang.*, 3(ICFP), jul 2019. doi: 10.1145/3341692. URL `https://doi.org/10.1145/3341692`. 1.6.1

[16] Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. Collapsible contracts: Fixing a pathology of gradual typing. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018. doi: 10.1145/3276503. URL `https://doi.org/10.1145/3276503`. 1.6.1

[17] Carlo Alberto Furia and Bertrand Meyer. Inferring loop invariants using postconditions. In *Fields of logic and computation*, pages 277–300. Springer, 2010. 1.6.2

[18] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. *ACM Transactions on Programming Languages and Systems*, 36 (4):12:1–12:44, October 2014. 1.2, 1.6.1, 5.3.4, 5.4

[19] Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 429–442, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837670. URL `http://doi.acm.org/10.1145/2837614.2837670`. 1, 1.3.1, 1.6, 1.6.1, 2, 2.1, 2.2, 2.2.3, 2.2.4, 2.2.5, 2.4

[20] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. 23(2): 167–189, June 2010. 3.1

[21] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. 1

[22] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium*, pages 41–55. Springer, 2011. 1, 3, 5.3.2

[23] Bart Jacobs, Jan Smans, and Frank Piessens. The VeriFast program verifier: A tutorial. *Tech. Rep.*, 2014. 5.3.4

[24] Nico Lehmann and Éric Tanter. Gradual refinement types. *SIGPLAN Not.*, 52(1):775–788, jan 2017. ISSN 0362-1340. doi: 10.1145/3093333.3009856. URL `https://doi.org/10.1145/3093333.3009856`. 1.6.1

[25] Nico Lehmann and Éric Tanter. Gradual refinement types. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*, pages 775–788, Paris, France, January 2017. 1.3.1, 1.6.1, 2.2.3

[26] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010. 1, 1.6.2

[27] K Rustan M Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. In *Foundations of Security Analysis and Design V*, pages 195–222. Springer, 2009. 1.6.2

[28] Meven Lennon-Bertrand, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. Gradualizing the calculus of inductive constructions. *ACM Transactions on Programming Languages and Systems*, 44(2), June 2022. doi: https://doi.org/10.1145/3495528. 1.6.1

[29] Paqui Lucio. A tutorial on using Dafny to construct verified software. *arXiv preprint arXiv:1701.04481*, 2017. 1.6.2

[30] Bertrand Meyer. Eiffel: A language and environment for software engineering. *Journal of Systems and Software*, 8(3):199–246, 1988. 1.6.3

[31] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM. ISBN 1-58113-297-2. doi: 10.1145/378239.379017. URL `http://doi.acm.org/10.1145/378239.379017`. 1

[32] Fabian Muehlboeck and Ross Tate. Sound gradual typing is nominally alive and well. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017. doi: 10.1145/3133880. URL `https://doi.org/10.1145/3133880`. 1.6.1, 4

[33] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016. 1, 1.2, 1.4, 3, 3.1, 3.4, 5.3.2

[34] Huu Hai Nguyen, Viktor Kuncak, and Wei-Ngan Chin. Runtime checking for separation logic. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 203–217. Springer, 2008. 1.6.3, 1.6.4

[35] Phúc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. Soft contract verification. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, page 139–152, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328739. doi: 10.1145/2628136.2628156. URL `https://doi.org/10.1145/2628136.2628156`. 1.6.4

[36] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *ACM SIG-*

PLAN Notices, volume 40, pages 247–258. ACM, 2005. 1, 1.3, 1.6, 2

[37] Guillaume Petiot, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. StaDy: Deep integration of static and dynamic analysis in Frama-C. 2014. 1.6.2

[38] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002. 1, 1.6

[39] Malte H Schwerhoff. *Advancing automated, permission-based program verification using symbolic execution*. PhD thesis, ETH Zurich, 2016. 3.4.6, A.2.3

[40] Ilya Sergey and Dave Clarke. Gradual ownership types. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, pages 579–599, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-28868-5. doi: 10.1007/978-3-642-28869-2_29. URL `http://dx.doi.org/10.1007/978-3-642-28869-2_29`. 1.6.1

[41] Jeremy Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2007. 1, 1.6

[42] Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006. 1, 1.6, 3.1, 4

[43] Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015. 1, 1.3.2, 1.6, 1.6.1, 1.6.4, 2, 2.4, 3.1

[44] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *European Conference on Object-Oriented Programming*, pages 148–172. Springer, 2009. 1, 1.3, 1.6, 2, 2.1.1, 3.3.1, 5.3

[45] Alexander J Summers and Sophia Drossopoulou. A formal semantics for isorecursive and equirecursive state abstractions. In *European Conference on Object-Oriented Programming*, pages 129–153. Springer, 2013. 1.3.2, 1.4.1, 2, 2.1, 2.1.2

[46] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? *SIGPLAN Not.*, 51(1):456–468, jan 2016. ISSN 0362-1340. doi: 10.1145/2914770.2837630. URL `https://doi.org/10.1145/2914770.2837630`. 1.2, 4, 4.1, 4.3

[47] Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. Gradual verification of recursive heap data structures. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020. 1.2, 4.1

[48] Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. Gradual verification of recursive heap data structures. Zenodo, October 2020. doi: 10.5281/zenodo.4085932. URL `https://doi.org/10.5281/zenodo.4091690`. 2

[49] Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *European Conference on Object-Oriented Programming*, pages 459–483. Springer, 2011.

1.6.1

[50] Conrad Zimmerman, Jenna DiVincenzo, and Jonathan Aldrich. Sound gradual verification with symbolic execution. *arXiv preprint arXiv:2311.07559*, 2023. 3