Instrumentation Analysis: An Automated Method for Producing Numeric Abstractions of Heap-Manipulating Programs

Stephen Magill

CMU-CS-10-150

November 29, 2010

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Peter Lee, Chair
Stephen Brookes
John Reynolds
Byron Cook, Microsoft Research, Cambridge, UK

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

This research was sponsored by the National Science Council, Taiwan, (ICAST) under the Grant No. NSC96-3114-P-001-002-Y; the Office of Naval Research under grant number N0001401107; the National Science Foundation under grant number CCF0429120; the U.S. Army Research Office under grant number DAAD19-01-1-0485; the Commonwealth of Pennsylvania under grant number FC410004860; and DARPA under grant number F19628-00-C-0003. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Separation Logic, Instrumentation Analysis, Static Analysis, Abstract Interpretation, Termination Proving, Hoare Logic

Abstract

A number of questions regarding programs involving heap-based data structures can be phrased as questions about numeric properties of those structures. A data structure traversal might terminate if the length of some path is eventually zero or a function to remove n elements from a collection may only be safe if the collection has size at least n.

In this thesis, we develop proof methods for reasoning about the connection between heap-manipulating programs and numeric programs. In addition, we develop an automatic method for producing numeric abstractions of heap-manipulating programs. These numeric abstractions are expressed as simple imperative programs over integer variables and have the feature that if a property holds of the numeric program, then it also holds of the original, heap-manipulating program. This is true for both safety and liveness. The abstraction procedure makes use of a shape analysis based on separation logic and has support for user-defined inductive data structures.

We also discuss a number of applications of this technique. Numeric abstractions, once obtained, can be analyzed with a variety of existing verification tools. Termination provers can be used to reason about termination of the numeric abstraction, and thus termination of the original program. Safety checkers can be used to reason about assertion safety. And bound inference tools can be used to obtain bounds on the values of program variables. With small changes to the program source, bounds analysis also allows the computation of symbolic bounds on memory use and computational complexity.

Acknowledgments

I would first like to thank my thesis committee. I am very appreciative of the level of interest they all showed and the amount of time that they committed to meeting during the work's progression and to reading once the document was complete. Their advice and support was invaluable during this process. In particular, I want to thank my advisor, Peter Lee, for always finding time in his (very busy) schedule and for his constant encouragement. Thanks to Byron Cook for giving me the opportunity to spend time with the wonderful group in Cambridge (and thanks to Josh Berdine for the many interesting discussions we had there). Thanks to John Reynolds and Stephen Brookes for many helpful meetings and for their careful reading of the final document.

Thanks also to my collaborators along the way: Edmund Clarke, Aleksandar Nanevski, Yih-Kuen Tsay, Ming-Hsien Tsai, Ashutosh Gupta, Andrey Rybalchenko, Jiri Simsa, Mohammad Raza, Satnam Singh, Viktor Vafeiadis, Josh Berdine, Kevin Donnelly, Tyler Gibson, Neel Krishnaswami, and Sungwoo Park. A special thanks to Aleksandar Nanevski for first suggesting that I work with Separation Logic.

I also want to thank my parents and sister. Special thanks to my father for buying me that first Macintosh (and a progression of computers thereafter). Thanks to my mother for always encouraging me to try new things and to read, read, read. Thanks to my sister for being not only a sibling, but also a wonderful friend.

Finally, I want to thank my wife, Laura. She wasn't around yet at the start, but she more than made up for it at the end, providing constant support and understanding (as well as just the right amount of pressure to finish).

Contents

1 Introduction **1**

1.1 Approach 2

1.2 Contributions 3

1.3 Example 4

2 Preliminaries **13**

2.1 Programs 13

2.1.1 Syntax and Typing 14

2.1.2 Semantics 15

2.2 Separation Logic 25

2.2.1 Effect of Free Variables 27

2.2.2 Defining Inductive Pointer Structures 31

2.3 Semantics of Programs 47

2.3.1 Transition Systems 47

2.3.2 Programs As Transition Systems 48

2.3.3 Transitive Closure of Relations 49

2.3.4 Deadlock and Angelic Non-determinism 49

2.4 Representing C Programs 51

2.4.1 Control Flow 52

2.4.2 Memory Operations 52

2.4.3 Unhandled Features 58

2.5 Generating C Programs 60

3 Abstractions and Program Properties 63

3.1 LTSL 64

3.1.1 Notation 66

3.1.2 Examples 67

3.1.3 Core Connectives 69

3.2 Stuttering Equivalence 72

3.2.1 Mapping Between Stuttering Equivalent Traces 80

3.2.2 Stuttering Containment 86

3.2.3 Programs and Stuttering Equivalence 88

3.3 Stuttering Equivalence and LTSL Properties 94

3.3.1 Syntactic Descriptions of E-invariance 97

3.3.2 Translating Results Obtained By Analyzing Abstractions 101

3.3.3 Example 117

3.4 Stuttering Simulation 119

3.5 Properties of Interest 127

4 Instrumented Programs 129

4.1 Theory 130

4.1.1 Common Cases 133

4.1.2 Properties 144

4.1.3 Derived Rules 148

4.2 Example 154

4.2.1 Alternate Size Measures 158

4.3 Soundness 160

4.4 Numeric Abstractions 170

4.4.1 Projection and Simulation 173

4.4.2 Combining Projection and Instrumentation 176

4.5 Example 178
4.6 Summary 182
4.7 Conclusion 186

5 Instrumentation Analysis 189
5.1 Symbolic State Formulae 190
5.2 Inductive Predicate Specifications 192
5.3 Basic Types 199
5.4 Basic Structure 201
5.4.1 `instrument` 202
5.4.2 `geninstCont` 208
5.4.3 `instPost` 214
5.5 Theorem Proving 220
5.5.1 Entailment 221
5.5.2 `implies` 237
5.5.3 Frame Inference 239
5.5.4 `exposeCellThenInst` 252
5.6 Example 253
5.7 Abstraction 258
5.7.1 Abstraction Patterns 259
5.7.2 Empty Patterns 266
5.7.3 Applying Abstraction Patterns 268
5.7.4 Additional Comments 274
5.8 Example (continued) 276
5.9 Tracking Flow of Control 280
5.10 Translating Branch Conditions 282
5.11 Experimental Results 291
5.11.1 Simple Examples 291
5.11.2 Complex Examples 293

5.11.3 Summary and Challenges 295

6 Related Work 297

6.1 Approaches to Analyzing the Heap 297

6.2 Termination Proving 301

6.3 Program Logics 302

7 Conclusion 307

7.1 Logic of Instrumentation 307

7.2 Analysis Algorithm 309

7.3 Implementation 310

A Guide to Notation 313

A.1 Programs, States, and Transition Systems 313

A.2 Relations 314

A.3 Separation Logic 315

A.4 LTSL 315

B Pseudo-code 317

B.1 Local Functions 319

Bibliography 321

List of Figures

1.1 A function for depth-first traversal of a tree rooted at `root` 6

1.2 Sample execution showing results from the first four iterations of the loop in the `traverse` function from Figure 1.1. 7

1.3 A numeric abstraction of the program in Figure 1.1. 9

1.4 An example showing `slen` and `ssize` used in the program in Figure 1.3. `slen` is the number of nodes in the stack and `ssize` is the sum of the values in the bold circles. The shaded area contains the nodes that contribute to `ssize` and nodes in this area are labeled with the size of the subtree rooted at that node. Empty trees (denoted by nil) have size 0. . . . 10

1.5 An illustration of the notion of `ssize` used to generate the program in Figure 1.6. The shaded area contains the nodes contributing to `ssize`. Empty trees (denoted by nil) have size 1. Non-empty nodes are labeled with the size of the subtree rooted at that node. `ssize` is the sum of the values in the bold circles, plus 1 for the first element in the stack, as nil has size 1 using this notion of size. 10

1.6 A numeric abstraction of the program in Figure 1.1 with the notion of *ssize* and *tsize* given in Figure 1.5. 11

2.1 Syntax of programs. 16

2.2 Semantics of expressions. $\wedge, \vee, \neg$ in the definitions refer to the standard Boolean operations with type *Bool* $\times$ *Bool* $\rightarrow$ *Bool* (for $\wedge$ and $\vee$) and *Bool* $\rightarrow$ *Bool* (for $\neg$). The functions $+, -, \times$ refer to the standard addition, subtraction, and multiplication functions of type $\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$. The $\leq$ relation is the standard "less than or equal to" relation on integers and $=$ is the identity relation on addresses, which relates each address only to itself. 18

2.3 Semantics of commands. $dom(g)$ indicates the domain of function g. The notation $g[x \to v]$ indicates the function that is the same as g, except that x is mapped to v. The notation $h[v_1.f \to v_2]$ indicates the heap that is the same as h except the record at v_1 maps field f to v_2. We write $h - X$ to indicate the function obtained by restricting the domain of h to $dom(h) - X$. 23

2.4 Semantics of continuations. The semantic rule for "$\mathsf{assume}(e)\texttt{;}\,k$" is included for clarity, but officially we consider "$\mathsf{assume}(e)\texttt{;}\,k$" to be an abbreviation for "$\mathsf{branch}\ e \Rightarrow k\ \mathsf{end}$" (which produces the same result as the rule above). 24

2.5 Iteration number one of a loop that creates a singly-linked list. 26

2.6 Syntax of separation logic formulae. 27

2.7 Semantics of separation logic formulae. We have combined the $\exists$ rules for address and integer-valued variables, using a "/" to separate the alternatives. The field names in any record ρ must be distinct. The semantics of expressions, $[\![e]\!]\ s$, is given in Figure 2.2. 28

2.8 The definition of the function $fv(Q)$, which gives the free variables of formula Q. If $Q = e^{\mathrm{b}}$, the free variables are as given in Definition 2. . . . 29

2.9 Translations of C programs with regular control-flow into the syntax presented in Section 2.1. The function "ctrans()" represents a recursive application of these rules. We assume that fresh labels (l_i) are generated and inserted in the C program wherever necessary to apply these rules. Translations for atomic commands are not given, but are discussed in Section 2.4.2. 54

3.1 Syntax of the logic LTSL. 65

3.2 Semantics of LTSL formulae. The notation T_i denotes the suffix of T starting at position i (where the first element has position 0). The satisfaction relation for Q is in Figure 2.7. We write $T \not\models_X \phi$ to indicate that the relation $T \models_X \phi$ does not hold. 66

3.3 Example depicting the sequences, functions, and variables involved in the proof of Lemma 8. 81

3.4 Four examples of stuttering equivalent programs. Each example involves a different continuation at L_0. 90

3.5 Increasingly weaker abstractions of P_5. 92

3.6 Two programs with traces related by $\approx_{=_{\{\mathtt{x},\mathtt{t}\}}}$ 101

3.7 Definition of $\boxed{\exists}$ and $\boxed{\forall}$. 103

3.8 Derivations showing that our definition of $\boxed{\exists}$ is consistent with the rewritings given in Theorem 9. The corresponding derivations for $\boxed{\forall}$ are identical, with the symbols $\boxed{\exists}$ and $\boxed{\forall}$ interchanged. 104

3.9 Pictorial overview of the proof of Theorem 3.9. The picture depicts how we build up T', α, and β. Solid elements of the figure are given. These include $\alpha(i)$, $\beta(i)$, the elements of T and the fact that $T(\alpha(i))\ R\ T'(\beta(i))$. The dashed elements are defined / proved in terms of these givens. Definitions must be provided for $\alpha(i+1)$, $\beta(i+1)$, and the elements of T' from index $\beta(i)$ to $\beta(i+1)$. It must then be proved that $\big(T(\alpha(i+1))\big)\ R\ \big(T'(\beta(i+1))\big)$ and that $\big(T(a)\ R\ T'(b)\big)$ for all a, b such that $\alpha(i) \leq a < \alpha(i+1)$ and $\beta(i) \leq b < \beta(i+1)$. 122

4.1 Rules for establishing that $\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k$, read "under precondition Q, with label invariants Γ, the continuation $\widehat{k}$ is an instrumented version of k with instrumentation variables V." Premises of the form $\{Q\}\ c\ \{Q'\}$ are partial correctness triples and hold iff for all s, h, $(s, h) \models Q$ implies $(\forall (s', h') \in (\llbracket c \rrbracket\ (s, h)).\ (s', h') \models Q')$. Premises of the form $Q \Rightarrow Q'$ hold iff $Q \Rightarrow Q'$ is valid (that is, $(s, h) \models (Q \Rightarrow Q')$ for all s, h). 132

4.2 Rule for proving that $\widehat{P}$ is an instrumented version of P. The function $\mathit{fv}(P)$ gives the set of variables occurring free in P. Since there are no binding constructs in our language, this is just the set of all variables appearing in P. 133

4.3 Derivation showing an instrumented program that performs a deterministic update of a variable representing the length of a linked list. I-A stands for INST-ASSIGN. 135

4.4 Derivation showing that, for the tree traversal program on page 136, the commands given re-establish the invariant $\Gamma(L_1)$. We write I-E as an abbreviation for INST-EXISTS and abbreviate STRENGTHENING as STRENGTHEN. 139

4.5 Derivation corresponding to the insertion of a case split on $e_1 \vee e_2$. The premises that become premises of the derived rule are boxed (the other two premises are tautologies). We abbreviate STRENGTHENING as STR and INST-ASSUME as I-A. The unlabeled rule is an instance of INST-DISJ. 150

4.6 Derivation corresponding to the translation of branch conditions into conditions on instrumentation variables. In the rule labeled $\forall i$, the premise holds for each value of i. The premises that become premises of the derived rule are boxed. We require that they hold for each $i \in \{1, \ldots, n\}$. . . 152

4.7 Derivation of the INST-ASSIGN rule for the case where $x \notin fv(e)$. The formulas and conditions that become premises and side conditions in the derived rule are boxed. The unboxed formulas can always be made to hold, either because they are tautologies or, in the case of $x' \notin fv(Q)$ because we get to choose x' when constructing the derivation. I-A stands for INST-ASSUME, I-E stands for INST-EXISTS. All other rules are instances of STRENGTHENING. 154

4.8 C code implementing a membership query for an ordered binary tree. . . 155

4.9 The program from Figure 4.8 translated into our program notation, with control points numbered. 156

4.10 Instrumented version of the program in Figure 4.9. 157

4.11 Guide to variable names used throughout the proof of Theorem 22. In each case of the proof, our goal is to show that one of the dashed relation lines exists. 164

4.12 Definition of the function $\pi_V(k)$ which projects a continuation onto variables in V. 172

4.13 A summary of the current state of the technical development. 177

4.14 An example program that traverses a circular linked list, conditionally freeing elements. 179

4.15 An instrumented version of the program in Figure 4.14 and the corresponding projection onto the set $\{n\}$. 181

4.16 An instrumentation and projection of the program in Figure 4.14, with instrumentation variables n and z and projection variables n, z, v. 183

4.17 The numeric program from Figure 4.16, but rearranged so that the cases of the second branch are split into separate continuations. 184

5.1 Restricted subset of separation logic formulae. The notation $\vec{x}$ indicates a list of variables $x_1, x_2, \ldots, x_n$ and $\exists\vec{x}.\ Q$ is shorthand for $\exists x_1.\exists x_2.\ldots\exists x_n.\ Q$. 190

5.2 Equivalence relation for symbolic state formulae. 191

5.3 Syntax of inductive specifications as implemented in THOR. The notation '|' is used to indicate the literal character |, and distinguish it from the BNF grammar operator consisting of the same symbol. 193

5.4 Graphical depiction of the doubly-linked list segment predicate. 194

5.5 Types used by the instrumentation algorithm. 200

5.6 A summary of the primary functions involved in the implementation. . . . 202

5.7 Additional functions used by the implementation. These are primarily concerned with reasoning about implications between symbolic state formulae. 203

5.8 Proof system for entailment. Basic rules. 225

5.9 Proof system for entailment. Rules for inductively specified predicates and variables. We write $\vec{\underline{z}} := ?$ to indicate the sequence of commands $\underline{z}_1 := ?$; . . . ; $\underline{z}_n := ?$. 226

5.10 Rules for frame inference that are the same as for entailment. 245

5.11 Rules for frame inference that differ from those for entailment. 247

5.12 Proof for the frame inference query

$$ls(\underline{n}; x, \mathsf{nil}) \wedge x \neq \mathsf{nil} \underset{\mathbf{S}}{\Longrightarrow}_{f_k} x \mapsto \Box \mathbin{/\!\!/} \Gamma \vdash \widehat{k}$$

We use $\Gamma_1, \widehat{k}_1$ to refer to the results from the left branch and $\Gamma_2, \widehat{k}_2$ to refer to the result from the right branch. 256

5.13 Main rewrite rules for abstraction. We use the notation $\vec{\underline{x}} := ?$ to indicate $\underline{x}_1 := ?$; . . . ; $\underline{x}_n := ?$. 272

5.14 The full instrumentation of the singly-linked list example. 278

5.15 A simplified version of the instrumentation given in Figure 5.14. 279

5.16 An instrumentation of the singly-linked list example that tracks flow of control using a variable $\underline{p}$. 281

5.17 The numeric program corresponding to the program in Figure 5.16. . . . 283

5.18 Proof for the given frame inference query. Below each rule name we show the value that Π_a has in the conclusion of that rule. 289

5.19 The numeric program corresponding to the program from page 284 after perform branch condition annotation. The original branch conditions are given in square brackets. 290

List of Tables

5.2 Experimental results. Time is in seconds. T_{NA} represents the time required to produce the numeric abstraction. T_{BLAST}, T_{ARMC}, and $T_{\text{ARMC-LIVE}}$ represent the time taken to verify the numeric abstraction by BLAST, ARMC, and ARMC-LIVE respectively. 292

5.3 Heap bounds and lines of code. 295

Chapter 1

Introduction

Current static analysis tools can check a wide variety of both safety and liveness properties for programs involving integer variables. Tools such as BLAST [Henzinger et al., 2002], SLAM [Ball et al., 2001], ARMC [Podelski and Rybalchenko, 2007], ASTRÉE [Cousot et al., 2005], SPEED [Gulwani et al., 2009] and TERMINATOR [Cook et al., 2006] all focus on this class of programs. Some of these also have support for pointers, but the heap reasoning is generally kept as simple as possible for the given problem domain.

Difficulty occurs when we try to integrate these methods with very precise methods for heap analysis. Such combinations generally involve a large increase in complexity, both in terms of the verification problem and in the implementation. In this thesis, we offer a solution to this problem in the form of an automatic analysis method that proves program properties by converting a heap-manipulating program into a numeric program that can then be analyzed by analysis tools that only support integer-valued variables.

The numeric program may include additional variables, called *instrumentation variables*, which are not present in the input program. These variables track numeric properties of heap-based data structures, such as the height of a tree, the maximal element in a list of integers, or the length of a path between two points in a data structure. Safety and liveness of the numeric program can be analyzed and the results carried over to the original heap-manipulating program. Bounds on variables are also preserved, which, when combined with additional instrumentation, allows us to use the numeric program to calculate bounds on execution time and memory usage.

1.1 Approach

The approach taken by this thesis is to prove properties of heap programs by reducing them to numeric programs using a static analysis based on separation logic. As such, there are two main questions to address: "Why use separation logic?" and "Why generate numeric programs?"

Why Separation Logic? Work such as [Magill et al., 2006, Distefano et al., 2006, Chang et al., 2007, Calcagno et al., 2009, Yang et al., 2008] has firmly established separation logic as a viable basis for automated program analysis. Its suitability stems from its focus on *local reasoning* [O'Hearn et al., 2001], which means that when performing analysis of a piece of code, we need only consider memory used by that code, rather than the global heap. This allows us to break the verification problem into several smaller subproblems and enables results to be re-used in different contexts, all of which helps improve scalability of analyses based on separation logic.

In addition, the inductive predicates used by separation logic to define data structures can be viewed as specifying the connection between the concrete pointer structures manipulated by a program and more abstract properties of these structures. We leverage this ability of separation logic in our static analysis to establish a link between concrete pointer structures and associated size measures. Such measures include obvious counts, such as "the size of the list starting at *x*" as well as less obvious metrics, such as "the number of nodes in the tree at *root* which are to the left of the path from *root* to *curr*." These measures are critical when proving termination and other liveness properties, as well as being useful for safety properties.

Why Numeric Programs? Given that there are techniques that prove termination of pointer programs directly [Brotherston et al., 2008b, Berdine et al., 2006, Loginov et al., 2006b], one might wonder why it is useful to introduce the added complication of translating pointer programs to numeric programs and then proving termination of these numeric programs. One answer is that, in many ways, using numeric programs as an intermediate form actually simplies the program analysis. Termination proving itself is a complex process of computing transitive closures and inferring ranking functions [Podelski and Rybalchenko, 2004, Cook et al., 2006]. By making the generation of numeric programs the end goal of the shape analysis, we insulate it from the complexities of termination proving (and shape analysis already has plenty of complexity itself). Furthermore, by studying what we can prove while still separating heap analysis from numeric analysis, we are able to investigate the interplay between the fundamentally structural notion of heap and fundamentally arithmetic termination arguments.

Finally, because the technique of generating numeric programs makes use of termination analysis in a "black box" fashion, we can benefit immediately from advances in termination proving without requiring any changes to the work described and implemented in this thesis. Given that there is a large and active community doing termination research [Bradley et al., 2005b,a, Cook et al., 2009b, 2008, Giesl et al., 2006], this is a major benefit of our approach. This same argument applies to other applications of this technique, such as computing bounds or proving safety properties. Furthermore, a significant advantage of this approach is the fact that the same numeric abstraction can be used to produce safety proofs, termination proofs, and bounds on variable values. This significantly reduces the amount of work that must be done to prove multiple properties of a program.

1.2 Contributions

The contributions of this thesis are as follows:

1. We develop a theory of *instrumented programs* as a means of relating heap-manipulating programs and numeric abstractions. Instrumented programs use sep-

aration logic annotations to connect the commands in the numeric abstraction with the states of the original program.

2. A static analysis that automates the generation of numeric abstractions. This aspect of the work involves the specification of a proof system for separation logic assertions, a strategy for proof search in this system, and the definition of symbolic execution and abstraction rules for separation logic formulas involving inductive predicates. These components are all augmented with rules for generating numeric commands that describe how data structure manipulations change numeric properties of data structures. These commands form the building blocks from which the numeric abstraction is constructed.

3. An implementation of the static analysis described above that supports the analysis of C programs. It accepts user-specified inductive data structure definitions and thus allows support for new data structures to be added fairly easily. Experimental results involving a number of examples and various data structures are given. Our experiments also consider multiple program properties, including safety, termination, and memory bounds.

1.3 Example

We conclude this section with an example that concretely demonstrates our approach. Consider the function `traverse` in Figure 1.1. This C-style code performs a left-to-right, depth-first traversal of the tree at `root`. It does this by maintaining a stack of nodes to be processed. The stack is a linked-list with nodes of type `TreeList` and initially contains a single node with a pointer to the root of the tree. On each iteration, the top element of the stack is removed and its children are added. Empty trees are discarded and when the entire stack is empty, execution terminates.

There are a number of properties one might want to prove about this code. First, we might like to show that it terminates on all valid inputs. We might also be interested in obtaining a bound on the amount of memory allocated by the procedure. Both these questions are really questions about numeric properties of the code. In the case of termination, we want to demonstrate that some ranking function decreases during each iteration. For a bound on the number of memory cells used, we can imagine adding a variable mem_usage to the program, which is initially zero and increments each time memory is allocated and decrements each time it is freed. We might be interested in obtaining a bound on mem_usage in terms of the size of the input tree.

In this example, answering either of these questions requires some reasoning about the shape and size properties of heap-allocated data structures. What we show in this thesis, and demonstrate in our experiments, is that the shape reasoning can be separated from the numeric reasoning by constructing a numeric program that explicitly tracks changes in data structure sizes. A graphical view of the steps in the algorithm is given in Figure 1.2. The figure also shows the values of the *slen* and *ssize* size measures, which we will describe shortly.

A numeric program for this example is given in Figure 1.3. This program can be constructed from the original using the rules in Chapter 4 and an equivalent, though larger program can be constructed automatically by the analysis implementation discussed in Chapter 5. In each case, the variables in the numeric program correspond to size properties of the data structures involved.

Informally, tsize_root is the number of nodes in the tree at the top of the stack, the variable slen tracks the number of nodes in the stack, and ssize is the number of nodes in the trees linked to by nodes in the stack, as depicted in Figure 1.4. The main integer variables ssize and slen are updated by means of a number of temporary variables. These updates are sometimes non-deterministic. For example, in the while loop in traverse, we remove the first element of the stack and, if it links to a non-empty tree, we replace it with two nodes that link to that tree's children. Thus, in the numeric program we must represent how removing an element from the stack changes the values slen and ssize. In the case of slen we know that the length simply decreases by one. For ssize, however, the effect of removing an element is not deterministic. The most we can conclude is that ssize can be broken into tsize, the size of the tree linked to by the element we just removed, and ssize_tail, the size of the remaining portion of the

```
struct Tree {
  Tree left;
  Tree right;
}
struct TreeList {
  Tree tree;
  TreeList next;
}

TreeList push(Tree r, TreeList next) {
  TreeList t;
  t = malloc();
  t->tree = r;
  t->next = next;
  return t;
}

void traverse(Tree root) {
  TreeList stack, tail;

  stack = push(root,0);
  while(stack != 0) {
    tail = stack->next;
    if(stack->tree == 0) { // remove empty trees
      free(stack);
      stack = tail;
    }
    else { // process non-empty trees
      tail = push(stack->tree->right,tail);
      tail = push(stack->tree->left,tail);
      free(stack);
      stack = tail;
    }
  }
}
```

Figure 1.1: A function for depth-first traversal of a tree rooted at `root`

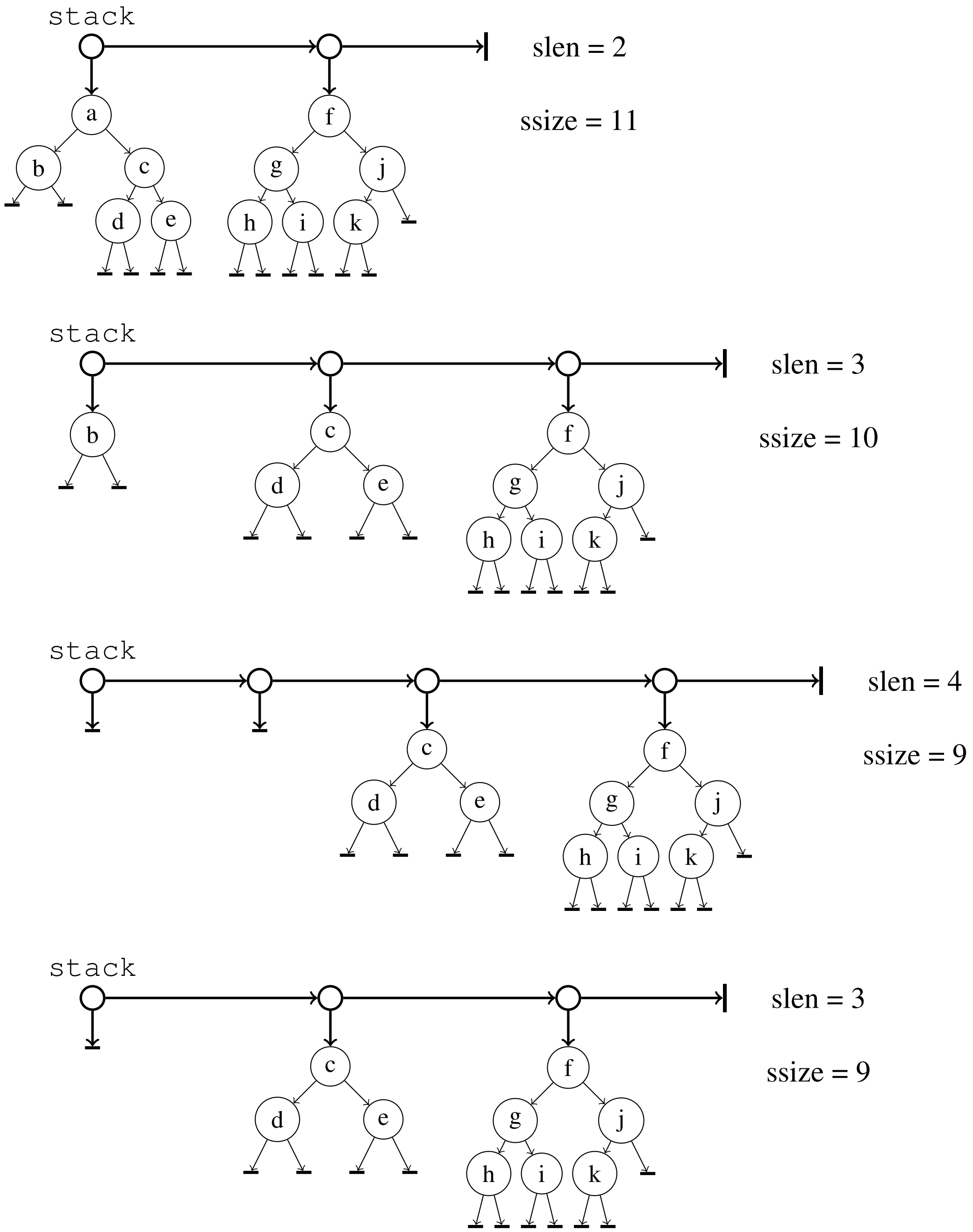


Figure 1.2: Sample execution showing results from the first four iterations of the loop in the `traverse` function from Figure 1.1.

stack. This is accomplished by the non-deterministic assignment on line 6 coupled with the assume statements at lines 7 and 8. A similar situation occurs on line 12, when we record the relationship between `tsize` and the sizes of its left and right children (`tsize_l` and `tsize_r`, respectively).

While `assume` statements are not part of standard C, they are accepted by many verification tools, allowing us to pass the code in Figure 1.3 directly to ARMC or TERMINATOR in order to check termination. In this case, the termination argument involves a lexicographic order on `ssize` and `slen`. By producing numeric abstractions such as that given in Figure 1.3, we allow ourselves and our program analysis tool to concentrate on the shape analysis problem, while leaving details of lexicographic rankings or disjunctive well-foundedness [Podelski and Rybalchenko, 2004] to other tools.

We can also ask bounds analysis tools as described in [Gulwani et al., 2009] and [Cook et al., 2009a] for a bound on the length of the stack. In this case, the stack can grow to size $\texttt{tsize\_root} + 1$ if the tree is maximally unbalanced. The theory presented in Chapter 4 also allows us to obtain a numeric program that demonstrates the expected logarithmic bound on stack length for balanced trees. However, the shape analysis used by our tool to compute numeric programs does not yet support reasoning about tree balance, so such proofs still involve a manual component.

Alternate Abstractions It is often the case that there are different notions of data structure size. The measures used in Figure 1.3 are fairly natural in the sense that the number of allocated heap cells reachable through the stack is the sum of `slen` and `ssize`. If we abandon this correspondence, we can obtain the simpler numeric abstraction given in Figure 1.6. In this case we have only one main size variable, `ssize`, which tracks the sum of the sizes of the subtrees reachable through the stack. However, we alter the notion of tree size such that empty trees have size equal to one, as depicted in Figure 1.5. This simplifies the termination argument, as there is now only a single count, `ssize`, which decreases during every iteration. However, we lose the ability to talk about the length of the stack when computing bounds and we lose the close connection between our counts and the number of allocated heap cells.

```
void traverse(int tsize_root) {
  assume(tsize_root >= 0);
  slen = 1;
  ssize = tsize_root;
  while(slen > 0) {
    tsize = ?; ssize_tail = ?;
    assume(tsize >= 0 && ssize_tail >= 0);
    assume(ssize == tsize + ssize_tail);

    if(tsize == 0) // remove empty trees
      slen--;
    else {         // process non-empty trees
      tsize_l = ?; tsize_r = ?;
      assume(tsize_l >= 0 && tsize_r >= 0);
      assume(tsize == tsize_l + tsize_r + 1);
      ssize = tsize_l + tsize_r + ssize_tail;
      slen++;
    }
  }
}
```

Figure 1.3: A numeric abstraction of the program in Figure 1.1.

The technique described in this thesis has the flexibility to allow either approach to numeric abstraction, and the implementation is not tied to any fixed notion of size. Instead, we allow the user to specify the definition of size they have in mind when running the tool. The numeric abstraction corresponding to the input C program is then automatically generated for that notion of size.

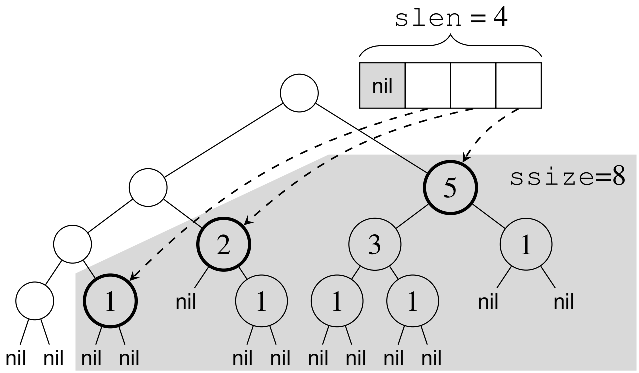


Figure 1.4: An example showing `slen` and `ssize` used in the program in Figure 1.3. `slen` is the number of nodes in the stack and `ssize` is the sum of the values in the bold circles. The shaded area contains the nodes that contribute to `ssize` and nodes in this area are labeled with the size of the subtree rooted at that node. Empty trees (denoted by nil) have size 0.

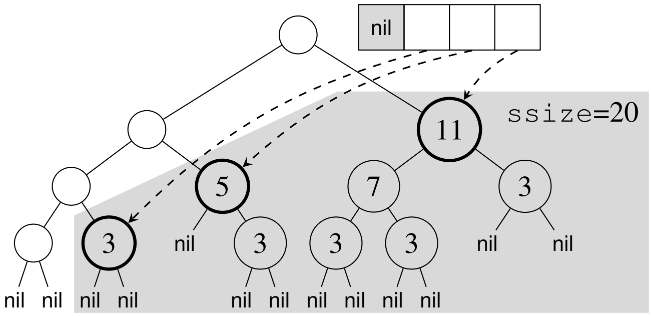


Figure 1.5: An illustration of the notion of `ssize` used to generate the program in Figure 1.6. The shaded area contains the nodes contributing to `ssize`. Empty trees (denoted by nil) have size 1. Non-empty nodes are labeled with the size of the subtree rooted at that node. `ssize` is the sum of the values in the bold circles, plus 1 for the first element in the stack, as nil has size 1 using this notion of size.

```
void traverse(int tsize_root) {
  assume(tsize_root > 0);
  ssize = tsize_root;
  while(ssize > 0) {
    tsize = ?; ssize_tail = ?;
    assume(tsize > 0 && ssize_tail >= 0);
    assume(ssize == tsize + ssize_tail);

    if(tsize == 1) // remove empty trees
      ssize = ssize_tail;
    else {         // process non-empty trees
      tsize_l = ?; tsize_r = ?;
      assume(tsize_l > 0 && tsize_r > 0);
      assume(tsize == tsize_l + tsize_r + 1);
      ssize = tsize_l + tsize_r + ssize_tail;
    }
  }
}
```

Figure 1.6: A numeric abstraction of the program in Figure 1.1 with the notion of *ssize* and *tsize* given in Figure 1.5.

Chapter 2

Preliminaries

In this chapter we present the basic definitions on which we will build the theory of instrumented programs and numeric abstractions that is the topic of this thesis. In Section 2.1, we present the syntax and semantics of the programming language we consider. Section 2.2 gives the syntax and semantics of the version of separation logic we use. Section 2.2.2 gives the syntax and semantics we adopt for inductive predicates in separation logic. And finally, Section 2.4 describes how we can translate C programs into the language defined in this chapter.

Notation A summary of the notation used in the thesis is given as Appendix A. This notation is described in detail in this and subsequent chapters.

2.1 Programs

Since our final goal is to analyze C-language programs, we consider an imperative programing language with unstructured flow of control (also referred to as a *goto language*). Because of the non-returning nature of gotos, the language is presented as a language of continuations. This serves as a convenient intermediate language for C since the C language contains a goto statement and all other control-flow constructs can be reduced to branches and gotos. We give examples of such reductions in Section 2.4.

The language is strongly typed, which deviates from C. We make this choice because it allows us to focus on issues of memory safety, assertion safety, and termination while ignoring issues such as pointer arithmetic and casts.

2.1.1 Syntax and Typing

Figure 2.1 gives the syntax for programs. A program P is a list of labeled continuations, which can also be viewed as a partial mapping from labels to continuations (and we will often use function syntax for P, writing $P(l)$ for the continuation labeled with l in program P). The first label l_0 is taken to be the starting point of execution and l_0 will be referred to as the *initial location*. We write $initloc(P)$ for the initial location of program P. The set L of labels is assumed to be infinite.

A continuation is a branching structure consisting of conditional branches and commands that update the state. At the leaves of each continuation, we have either a goto or an indication that execution should halt or abort. We write ϵ for the empty list of branch cases and omit it when writing branching continuations. For example, we write $\mathsf{branch}\ \mathsf{true} \Rightarrow k\ \mathsf{end}$ instead of $\mathsf{branch}\ \mathsf{true} \Rightarrow k, \epsilon\ \mathsf{end}$. We list $\mathsf{assume}(e)\mathsf{;}\,k$ as a continuation, but this is actually definable as $\mathsf{branch}\ e \Rightarrow k\ \mathsf{end}$—a fact we return to in Section 2.3.4.

Commands include the standard commands for variable assignment, heap lookup, heap mutation, memory allocation, and deallocation. The commands range over variables drawn from the infinite set *Vars* and field names drawn from the infinite set *Fields*.

We will write $k \in \mathit{subterms}(P)$ if k is a sub-term of some continuation in the range of P. A program P is considered *well-formed* iff $\{l \mid \mathsf{goto}\ l \in \mathit{subterms}(P)\} \subseteq dom(P)$, where $dom(P)$ is the domain of P (the set of labels prefixing continuations in P). This ensures that all jumps are to locations defined by P. We will restrict ourselves to well-formed programs for the rest of this thesis.

Variables and expressions are typed, with the types drawn from the set $\{\mathrm{a}, \mathrm{i}, \mathrm{b}\}$ (representing addresses, integers, and Booleans, respectively). We assume that the set *Vars* can be partitioned into two infinite subsets $\mathit{Vars}_{\mathrm{a}}$ and $\mathit{Vars}_{\mathrm{i}}$. We do not include variables of type b in our syntax or states. We write x^{a} to denote an element of $\mathit{Vars}_{\mathrm{a}}$ and x^{i} for an element of $\mathit{Vars}_{\mathrm{i}}$. We use τ to stand for either a or i. Often, types can be inferred from the context and, in such cases, we will omit them.

We take a similar approach to typing of record fields. We assume the set *Fields* can be partitioned into two infinite subsets $\mathit{Fields}_{\mathrm{a}}$ and $\mathit{Fields}_{\mathrm{i}}$ and write f^{a} for elements of $\mathit{Fields}_{\mathrm{a}}$ and f^{i} for elements of $\mathit{Fields}_{\mathrm{i}}$.

We make a distinction between integer values and values representing addresses as a means of ruling out pointer arithmetic. Pointer arithmetic could be handled by moving to a lower-level memory model, where addresses are integers and records are represented by contiguous groups of memory cells. However, our analysis algorithm does not support pointer arithmetic, so we chose to rule it out from the beginning.

2.1.2 Semantics

The semantics is given in terms of transitions between states. Each non-terminal state includes a *store* paired with a *heap*. Formally, a store is a mapping from variables to their values, which are either integers or addresses. We require that this mapping respects types and indicate this by using the notation $\rightarrow_\tau$ to denote the function space. A function f is in $\mathit{Vars} \rightarrow_\tau \mathit{Values}$ iff $f \in \mathit{Vars} \rightarrow \mathit{Values}$ and variables in $\mathit{Vars}_{\mathrm{i}}$ are mapped by f to integers while variables in $\mathit{Vars}_{\mathrm{a}}$ are mapped to addresses. We assume that $\mathbb{Z}$ and *Addr* are disjoint and that *Addr* is an infinite set. We use the meta-variable v to represent a value and s to represent a store.

$$v \in \mathit{Values} \overset{\mathrm{def}}{=} \mathbb{Z} \cup \mathit{Addr}$$
$$s \in \mathit{Stores} \overset{\mathrm{def}}{=} \mathit{Vars} \rightarrow_\tau \mathit{Values}$$

The set of addresses contains a distinguished element ***nil*** which is not in the domain of any heap. The heap is a finite partial function from non-***nil*** addresses to *records*, which

Syntax of Programs

Types	τ	$\in$	$\{\mathrm{a}, \mathrm{i}\}$
Variables	x^τ	$\in$	Vars_τ
Fields	f^τ	$\in$	Fields_τ
Labels	l	$\in$	L
Integers	n	$\in$	$\mathbb{Z}$
Integer Expressions	e^{i}	$::=$	$x^{\mathrm{i}} \mid n \mid e_1^{\mathrm{i}} + e_2^{\mathrm{i}} \mid e_1^{\mathrm{i}} - e_2^{\mathrm{i}} \mid e_1^{\mathrm{i}} \times e_2^{\mathrm{i}}$
Address Expressions	e^{a}	$::=$	$x^{\mathrm{a}} \mid \mathsf{nil}$
Boolean Expressions	e^{b}	$::=$	$\mathsf{true} \mid \mathsf{false} \mid e_1^{\mathrm{a}} = e_2^{\mathrm{a}} \mid e_1^{\mathrm{i}} \leq e_2^{\mathrm{i}} \mid e_1^{\mathrm{b}} \wedge e_2^{\mathrm{b}} \mid e_1^{\mathrm{b}} \vee e_2^{\mathrm{b}} \mid \neg e^{\mathrm{b}}$
Commands	c	$::=$	$x^\tau := e^\tau \mid x^\tau := ?^\tau \mid x_1^\tau := x_2^{\mathrm{a}}.f^\tau \mid x^{\mathrm{a}}.f^\tau := e^\tau \mid$
			$x^{\mathrm{a}} := \mathsf{alloc}(f_1^{\tau_1}, \ldots, f_n^{\tau_n}) \mid \mathsf{free}\ x^{\mathrm{a}} \mid \mathsf{skip}$
Branch Cases	β	$::=$	$e^{\mathrm{b}} \Rightarrow k, \beta \mid \epsilon$
Continuations	k	$::=$	$c\,\mathtt{;}\,k \mid \mathsf{halt} \mid \mathsf{abort} \mid \mathsf{goto}\ l \mid \mathsf{branch}\ \beta\ \mathsf{end} \mid \mathsf{assume}(e^{\mathrm{b}})\,\mathtt{;}\,k$
Programs	P	$::=$	$l_0 : k_0;\ \ldots;\ l_n : k_n$

Figure 2.1: Syntax of programs.

are finite partial functions from fields to values of the appropriate type. We use the metavariable h to represent an element of *Heaps*.

$$\mathit{Records} \overset{\text{def}}{=} \mathit{Fields} \overset{\text{fin}}{\rightharpoonup}_\tau \mathit{Values}$$

$$h \in \mathit{Heaps} \overset{\text{def}}{=} (\mathit{Addr} - \{\boldsymbol{nil}\}) \overset{\text{fin}}{\rightharpoonup} \mathit{Records}$$

As with stores, the functions that serve as the denotation of records must respect types. Unlike stores, they need not be defined on all elements of the domain (different heap cells may contain different sets of fields). We refer to an (s, h) pair as a *memory state*.

$$\mathit{Memory\ States} \quad (s, h) \quad \in \quad \mathit{Stores} \times \mathit{Heaps}$$

We also include an $\mathbf{error}$ state representing the result of an erroneous computation such as an attempt to dereference unallocated memory.

The semantics of expressions is given in Figure 2.2. In addition to the sets *Addr* and $\mathbb{Z}$, that were defined previously, the semantics of expressions makes use of a set *Bool* of Boolean values, defined as $\mathit{Bool} = \{\textbf{\textit{true}},\textbf{\textit{false}}\}$. We note the following theorem, which relates the meaning of expressions to their types and ensures that our interpretation of expressions is well-defined.

Theorem 1.

$$\forall s, e^{\mathrm{a}}.\ \llbracket e^{\mathrm{a}} \rrbracket\, s \in \mathit{Addr} \tag{2.1}$$

$$\forall s, e^{\mathrm{i}}.\ \llbracket e^{\mathrm{i}} \rrbracket\, s \in \mathbb{Z} \tag{2.2}$$

$$\forall s, e^{\mathrm{b}}.\ \llbracket e^{\mathrm{b}} \rrbracket\, s \in \mathit{Bool} \tag{2.3}$$

Proof. The proof is by induction on the structure of the expression language and each case follows directly from the expression semantics and the requirement that stores are well-typed. □

Another property of expressions is that only the portion of the store involving the variables that appear in the expression affects its value. This is captured by the following lemma.

Definition 1. *Let $s =_V s'$ hold iff $\forall x.\ x \in V \Rightarrow s(x) = s'(x)$.*

Definition 2. *Let $\mathit{fv}(e)$ be the function that returns the set of variables occurring free in e. Since there are no binding constructs in the expression language, this is just the set of all variables appearing in e.*

Lemma 1. *If $s =_V s'$ and $\mathit{fv}(e) \subseteq V$ then $\llbracket e \rrbracket\, s = \llbracket e \rrbracket\, s'$.*

Proof. The proof is by induction on the expression e. The inductive cases are straightforward. To take an example, consider the case $e_1 + e_2$. We assume $s =_V s'$ and $\mathit{fv}(e_1 + e_2) \subseteq V$. The second assumption implies $\mathit{fv}(e_1) \subseteq V$ and $\mathit{fv}(e_2) \subseteq V$. This allows us to apply the induction hypothesis and conclude that $\llbracket e_1 \rrbracket\, s = \llbracket e_1 \rrbracket\, s'$ and $\llbracket e_2 \rrbracket\, s = \llbracket e_2 \rrbracket\, s'$. It then follows that $\llbracket e_1 \rrbracket\, s + \llbracket e_2 \rrbracket\, s = \llbracket e_1 \rrbracket\, s' + \llbracket e_2 \rrbracket\, s'$, which, by the definition of $\llbracket e_1 + e_2 \rrbracket$ implies that $\llbracket e_1 + e_2 \rrbracket\, s = \llbracket e_1 + e_2 \rrbracket\, s'$.

Semantics of Expressions

$$\begin{aligned}
\llbracket n \rrbracket\, s &= n \\
\llbracket x^{\tau} \rrbracket\, s &= s(x^{\tau}) \\
\llbracket \mathsf{nil} \rrbracket\, s &= \textbf{\textit{nil}} \\
\llbracket \mathsf{true} \rrbracket\, s &= \textbf{\textit{true}} \\
\llbracket \mathsf{false} \rrbracket\, s &= \textbf{\textit{false}} \\
\llbracket \neg e^{\mathrm{b}} \rrbracket\, s &= \neg(\llbracket e^{\mathrm{b}} \rrbracket\, s) \\
\llbracket e^{\mathrm{i}}_1 + e^{\mathrm{i}}_2 \rrbracket\, s &= (\llbracket e^{\mathrm{i}}_1 \rrbracket\, s) + (\llbracket e^{\mathrm{i}}_2 \rrbracket\, s)
\end{aligned}$$

$$\begin{aligned}
\llbracket e^{\mathrm{i}}_1 - e^{\mathrm{i}}_2 \rrbracket\, s &= (\llbracket e^{\mathrm{i}}_1 \rrbracket\, s) - (\llbracket e^{\mathrm{i}}_2 \rrbracket\, s) \\
\llbracket e^{\mathrm{i}}_1 \times e^{\mathrm{i}}_2 \rrbracket\, s &= (\llbracket e^{\mathrm{i}}_1 \rrbracket\, s) \times (\llbracket e^{\mathrm{i}}_2 \rrbracket\, s) \\
\llbracket e^{\mathrm{a}}_1 = e^{\mathrm{a}}_2 \rrbracket\, s &= (\llbracket e^{\mathrm{a}}_1 \rrbracket\, s) = (\llbracket e^{\mathrm{a}}_2 \rrbracket\, s) \\
\llbracket e^{\mathrm{i}}_1 \leq e^{\mathrm{i}}_2 \rrbracket\, s &= (\llbracket e^{\mathrm{i}}_1 \rrbracket\, s) \leq (\llbracket e^{\mathrm{i}}_2 \rrbracket\, s) \\
\llbracket e^{\mathrm{b}}_1 \wedge e^{\mathrm{b}}_2 \rrbracket\, s &= (\llbracket e^{\mathrm{b}}_1 \rrbracket\, s) \wedge (\llbracket e^{\mathrm{b}}_2 \rrbracket\, s) \\
\llbracket e^{\mathrm{b}}_1 \vee e^{\mathrm{b}}_2 \rrbracket\, s &= (\llbracket e^{\mathrm{b}}_1 \rrbracket\, s) \vee (\llbracket e^{\mathrm{b}}_2 \rrbracket\, s)
\end{aligned}$$

Figure 2.2: Semantics of expressions. $\wedge, \vee, \neg$ in the definitions refer to the standard Boolean operations with type *Bool* × *Bool* → *Bool* (for $\wedge$ and $\vee$) and *Bool* → *Bool* (for $\neg$). The functions $+, -, \times$ refer to the standard addition, subtraction, and multiplication functions of type $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$. The $\leq$ relation is the standard "less than or equal to" relation on integers and $=$ is the identity relation on addresses, which relates each address only to itself.

The base cases for the constants are immediate, as the store does not affect their semantics at all. This covers $n, \mathsf{nil}, \mathsf{true}$, and false. We are left with the variable case. If $e = x$ then $\llbracket e \rrbracket\, s = s(x)$, so we must show $s(x) = s'(x)$. The definition of $s =_V s'$ gives us $x \in V \Rightarrow s(x) = s'(x)$, so it suffices to show $x \in V$. This follows directly from our assumption that $\mathit{fv}(x) \subseteq V$ and the fact that $\mathit{fv}(x) = \{x\}$. □

The semantics of commands is given in Figure 2.3. The command $x := e$ is a standard assignment statement, $x := ?$ is non-deterministic assignment, $x_1 := x_2.f$ reads a value from a heap cell, and $x.f := e$ writes a value into a heap cell. Attempts to read from or write to a non-existent record field result in a run-time error, represented by **error**. The command $x := \mathsf{alloc}(f_1, \ldots, f_n)$ allocates a new heap cell with fields $f_1, \ldots, f_n$. The fields are initially mapped to non-deterministically chosen values of the correct type. The field names provided must all be distinct. The command $\mathsf{free}\ x$ disposes of the heap cell at x. We permit the call $\mathsf{free}\ \mathsf{nil}$, which has the effect of a no-op. We do this to match the semantics of the "free" function call in the C programming language, which will be the source language we ultimate target with our analysis.

We claim that the type of $[\![c]\!]$ is *Stores* × *Heaps* → $2^{((\mathit{Stores}\times\mathit{Heaps})\cup\{\mathbf{error}\})}$. To verify this, we must check that, in all rules, the store and heap are updated in a manner consistent with the types. In all cases, this follows immediately from the well-typedness of the initial store and heap and Theorem 1.

One property of commands is that only the heap and the portion of the store corresponding to the variables used by the command affects execution. This is captured by the following Lemma.

Definition 3. *Let $fv(c)$ indicate the set of free variables occurring in command c. Since there are no binders in the syntax for commands, this is the set of all variables occurring in c.*

Lemma 2. *If $s_1 =_V s_2$ and $fv(c) \subseteq V$ then for all h, s_1', h' the following holds*

$$\Big((s_1', h') \in \big([\![c]\!]\,(s_1, h)\big)\Big) \Rightarrow \Big(\exists s_2'.\, (s_2', h') \in \big([\![c]\!]\,(s_2, h)\big) \wedge (s_1' =_V s_2')\Big)$$

This states that if V is a set containing the free variables of command c, and two stores agree on the values of variables in V, then an evaluation of c from either of the two stores has a matching evaluation starting from the other store (matching in the sense that the post-states agree on the values of variables in V).

Proof. The proof proceeds by case analysis on the command c in question and most cases follow directly from the definition of $[\![c]\!]$ and Lemma 1. Note that according to the semantics in Figure 2.3, we have

$$\forall c, s, h.\ \big(\mathbf{error} \in [\![c]\!]\,(s, h)\big) \Leftrightarrow \big([\![c]\!]\,(s, h) = \{\mathbf{error}\}\big)$$

To see why this holds, note that the only commands that can result in **error** are those of the form $x_1 := x_2.f$ or $x.f := e$ or free x. Examining the semantics for these commands reveals that the error case results in the singleton set $\{\mathbf{error}\}$. Thus, the fact that we have $(s_1', h') \in \big([\![c]\!]\,(s_1, h)\big)$ as a hypothesis implies that $\mathbf{error} \notin \big([\![c]\!]\,(s_1, h)\big)$.

CASE $x.f := e$: Since $\mathbf{error} \notin \big([\![x.f := e]\!]\,(s_1, h)\big)$, we have the following

$$s_1(x) \in dom(h) \wedge f \in dom(h(s_1(x)))$$

We have $s_1 =_V s_2$ as an assumption and $x \in V$ from our assumption that $\mathit{fv}(x.f := e) \subseteq V$. This then gives us $s_1(x) = s_2(x)$ and allows us to derive

$$s_2(x) \in \mathit{dom}(h) \wedge f \in \mathit{dom}(h(s_2(x)))$$

This implies that $[\![x.f := e]\!]\,(s_2, h)$ does not result in an error. Thus, we have

$$[\![x.f := e]\!]\,(s_1, h) = \{(s_1, h[(s_1(x)).f \to ([\![e]\!]\,s_1)])\}$$

and

$$[\![x.f := e]\!]\,(s_2, h) = \{(s_2, h[(s_2(x)).f \to ([\![e]\!]\,s_2)])\}$$

We must show $s_1 =_V s_2$, which we already have from our assumptions. We also must show the following.

$$\Big(h[(s_1(x)).f \to ([\![e]\!]\,s_1)]\Big) = \Big(h[(s_2(x)).f \to ([\![e]\!]\,s_2)]\Big)$$

Since $x \in V$, we have that $s_1(x) = s_2(x)$. Thus, the above reduces to showing that

$$([\![e]\!]\,s_1) = ([\![e]\!]\,s_2)$$

which follows from Lemma 1.

CASE $x_1 := x_2.f$: Again, we have from our assumptions that $x_1 := x_2.f$ does not result in $\mathbf{error}$. From $s_1 =_V s_2$ and $\mathit{fv}(x_1 := x_2.f) \subseteq V$, we have that $s_1(x_1) = s_2(x_1)$ and $s_1(x_2) = s_2(x_2)$. This gives us the following.

$$[\![x_1 := x_2.f]\!]\,(s_1, h) = \{(s_1[x_1 \to (h(s_1(x_2)))\ f], h)\}$$

and

$$[\![x_1 := x_2.f]\!]\,(s_2, h) = \{(s_2[x_1 \to (h(s_2(x_2)))\ f], h)\}$$

We must show

$$\Big(s_1[x_1 \to (h(s_1(x_2)))\ f]\Big) =_V \Big(s_2[x_1 \to (h(s_2(x_2)))\ f]\Big)$$

We have that $x_1 \in V$ and $s_1 =_V s_2$, so the above will hold if we can show

$$\Big(h(s_1(x_2))\Big) = \Big(h(s_2(x_2))\Big)$$

This holds if $s_1(x_2) = s_2(x_2)$ which follows from $x_2 \in V$ and $s_1 =_V s_2$.

CASE $\mathsf{free}\ x$: As before, we have $s_1 =_V s_2$ and $fv(\mathsf{free}\ x) \subseteq V$, which implies $x \in V$ and thus $s_1(x) = s_2(x)$. Since $[\![\mathsf{free}\ x]\!]\,(s_1, h) \neq \{\mathbf{error}\}$, we have $s_1(x) \in \big(dom(h) \cup \{\mathsf{nil}\}\big)$. This combined with $s_1(x) = s_2(x)$ gives us $s_2(x) \in \big(dom(h) \cup \{\mathsf{nil}\}\big)$. Since $[\![\mathsf{free}\ x]\!]\,(s_1, h) = (s_1, h - \{s_1(x)\})$, and $[\![\mathsf{free}\ x]\!]\,(s_2, h) = (s_2, h - \{s_2(x)\})$, we must show $s_1 =_V s_2$, which we already have, and $(h - \{s_1(x)\}) = (h - \{s_2(x)\})$, which follows from $s_1(x) = s_2(x)$.

CASE $x := ?$: We have

$$[\![x := ?]\!]\,(s_1, h) = \{(s_1', h) \mid s_1' = s[x \to v]\}$$

where v is chosen from the appropriate domain (either *Addr* or $\mathbb{Z}$). For s_2 we have

$$[\![x := ?]\!]\,(s_2, h) = \{(s_2', h) \mid s_2' = s[x \to v]\}$$

Suppose $(s_1', h) \in [\![x := ?]\!]\,(s_1, h)$. We must show

$$\exists s_2'.\ (s_2', h) \in ([\![x := ?]\!]\,(s_2, h)) \wedge s_1' =_V s_2'$$

We choose $s_2' = s_2[x \to s_1'(x)]$. Clearly this is in $[\![x := ?]\!]\,(s_2, h)$. To see that $s_1' =_V s_2'$, we must show that $s_2'(x) = s_1'(x)$, which is immediate from the definition of s_2'. Agreement of s_2' and s_1' on the rest of V follows from the assumption that $s_1 =_V s_2$.

CASE $x := \mathsf{alloc}(f_1, \ldots, f_n)$: The semantics of this command chooses an address v not in $dom(h)$ and assign v to x in the post-state. Since we are evaluating $x := \mathsf{alloc}(f_1, \ldots, f_n)$ under the same heap but a different store, we have that v is also a valid choice of address when determining $[\![x := \mathsf{alloc}(f_1, \ldots, f_n)]\!]\,(s_2, h)$. It remains to show that $s_1[x \to v] =_V s_2[x \to v]$, which follows from $s =_V s'$.

CASE $x := e$: We have

$$[\![x := e]\!]\,(s_1, h) = \{(s_1[x \to [\![e]\!]\,s_1], h)\}$$

and

$$[\![x := e]\!]\,(s_2, h) = \{(s_2[x \to [\![e]\!]\,s_2], h)\}$$

We must show

$$s_1[x \to [\![e]\!]\, s_1] =_V s_2[x \to [\![e]\!]\, s_2]$$

Since we have $s_1 =_V s_2$, it suffices to show that $[\![e]\!]\, s_1 = [\![e]\!]\, s_2$. This is established by Lemma 1. □

We also have a similar property for commands that result in an error.

Lemma 3. *If* $s_1 =_V s_2$ *and* $fv(c) \subseteq V$ *then*

$$\mathbf{error} \in \big([\![c]\!]\,(s_1, h)\big) \Rightarrow \mathbf{error} \in \big([\![c]\!]\,(s_2, h)\big)$$

Proof. The proof proceeds by case analysis on the command c. There are only three commands that can result in **error**. These are $x_1 := x_2.f$ and $x.f := e$ and free x.

CASE $x_1 := x_2.f$: If **error** $\in \big([\![x_1 := x_2.f]\!]\,(s_1, h)\big)$ then, according to the semantics of commands (Figure 2.3), either $s_1(x_2) \notin dom(h)$ or $f \notin dom(h(s_1(x_2)))$. Suppose $s_1(x_2) \notin dom(h)$. Then since $s_1 =_V s_2$ and $x_2 \in V$ we have $s_1(x_2) = s_2(x_2)$ and thus $s_2(x_2) \notin dom(h)$. If $f \notin dom(h(s_1(x_2)))$, then again we note that $x_2 \in V$ and thus $s_1(x_2) = s_2(x_2)$, which gives us $f \notin dom(h(s_2(x_2)))$.

CASE $x.f := e$: This is similar to the case above. We have either $s_1(x) \notin dom(h)$ or $f \notin dom(h(s_1(x)))$. We have $x \in V$ and $s_1 =_V s_2$, which yields $s_1(x) = s_2(x)$, which gives us that either $s_2(x) \notin dom(h)$ or $f \notin dom(h(s_2(x)))$.

CASE free x: In this case we have $s_1(x) \notin \big(dom(h) \cup \{\mathsf{nil}\}\big)$. Again $s_1(x) = s_2(x)$ and so $s_2(x) \notin \big(dom(h) \cup \{\mathsf{nil}\}\big)$ □

Figure 2.4 gives the transition semantics of continuations. There are three types of execution states: intermediate states, in which the continuation is still executing; terminal states, which indicate that execution has stopped; and goto states, which indicate that the end of this continuation has been reached but execution has not stopped and should continue from another continuation. Intermediate states have the form $\langle k, (s, h)\rangle$ where k is the current continuation and (s, h) is the current store and heap. Terminal states either have the form $\mathbf{final}(s, h)$, which indicates that the program has terminated in the memory

SEMANTICS OF COMMANDS

$$\begin{array}{rcll}
[\![\mathsf{skip}]\!]\,(s,h) & = & \{(s,h)\} & \\
[\![x^{\tau} := e^{\tau}]\!]\,(s,h) & = & \{(s[x^{\tau} \to [\![e^{\tau}]\!]\,s],h)\} & \\
[\![x^{\mathrm{a}} := ?^{\mathrm{a}}]\!]\,(s,h) & = & \{(s',h) \mid s' = s[x^{\mathrm{a}} \to v] \wedge v \in \mathit{Addr}\} & \\
[\![x^{\mathrm{i}} := ?^{\mathrm{i}}]\!]\,(s,h) & = & \{(s',h) \mid s' = s[x^{\mathrm{i}} \to v] \wedge v \in \mathbb{Z}\} & \\
[\![x_1^{\tau} := x_2^{\mathrm{a}}.f^{\tau}]\!]\,(s,h) & = & \{(s[x_1^{\tau} \to (h(s(x_2^{\mathrm{a}})))\ f^{\tau}],h)\} & \text{if } s(x_2^{\mathrm{a}}) \in \mathit{dom}(h) \\
 & & & \ \wedge\ f^{\tau} \in \mathit{dom}(h(s(x_2^{\mathrm{a}}))) \\
 & & \{\mathbf{error}\} & \text{otherwise} \\
[\![x^{\mathrm{a}}.f^{\tau} := e^{\tau}]\!]\,(s,h) & = & \{(s,h[(s(x^{\mathrm{a}})).f^{\tau} \to ([\![e^{\tau}]\!]\,s)])\} & \text{if } s(x^{\mathrm{a}}) \in \mathit{dom}(h) \\
 & & & \ \wedge\ f^{\tau} \in \mathit{dom}(h(s(x^{\mathrm{a}}))) \\
 & & \{\mathbf{error}\} & \text{otherwise}
\end{array}$$

$$\begin{array}{l}
[\![x^{\mathrm{a}} := \mathsf{alloc}(f_1^{\tau_1},\ldots,f_n^{\tau_n})]\!]\,(s,h) = \\
\qquad\qquad \{(s',h') \mid v \in \mathit{dom}(h') \text{ and } \mathit{dom}(h'(v)) = \{f_1^{\tau_1},\ldots,f_n^{\tau_n}\} \\
\qquad\qquad\quad \text{and } h' - \{v\} = h \\
\qquad\qquad\quad \text{and } s' = s[x^{\mathrm{a}} \to v] \text{ and } v \in \mathit{Addr} \\
\qquad\qquad\quad \text{and } h'(v)(f_i^{\tau_i}) \in \mathbb{Z} \text{ if } \tau_i = \mathrm{i} \\
\qquad\qquad\quad \text{and } h'(v)(f_i^{\tau_i}) \in \mathit{Addr} \text{ if } \tau_i = \mathrm{a}\}
\end{array}$$

$$\begin{array}{rcll}
[\![\mathsf{free}\ x^{\mathrm{a}}]\!]\,(s,h) & = & \{(s,h - \{s(x^{\mathrm{a}})\})\} & \text{if } s(x^{\mathrm{a}}) \in \big(\mathit{dom}(h) \cup \{\mathsf{nil}\}\big) \\
 & & \{\mathbf{error}\} & \text{otherwise}
\end{array}$$

Figure 2.3: Semantics of commands. $dom(g)$ indicates the domain of function g. The notation $g[x \to v]$ indicates the function that is the same as g, except that x is mapped to v. The notation $h[v_1.f \to v_2]$ indicates the heap that is the same as h except the record at v_1 maps field f to v_2. We write $h - X$ to indicate the function obtained by restricting the domain of h to $dom(h) - X$.

Semantics of Continuations

$$\frac{(s', h') \in [\![c]\!]\,(s,h)}{\langle (c\,\texttt{;}\,k), (s,h)\rangle \rightsquigarrow \langle k, (s', h')\rangle} \qquad \frac{\mathbf{error} \in [\![c]\!]\,(s,h)}{\langle (c\,\texttt{;}\,k), (s,h)\rangle \rightsquigarrow \mathbf{error}}$$

$$\frac{[\![e_i]\!]\,s = \textbf{\textit{true}}}{\langle \textsf{branch}\ \ldots, e_i \Rightarrow k_i, \ldots\ \textsf{end}, (s,h)\rangle \rightsquigarrow \langle k_i, (s,h)\rangle} \qquad \frac{}{\langle \textsf{halt}, (s,h)\rangle \rightsquigarrow \mathbf{final}(s,h)}$$

$$\frac{}{\langle (\textsf{goto}\ l), (s,h)\rangle \rightsquigarrow \mathbf{goto}(l, (s,h))} \qquad \frac{}{\langle \textsf{abort}, (s,h)\rangle \rightsquigarrow \mathbf{error}}$$

$$\boxed{\frac{[\![e]\!]\,s = \textbf{\textit{true}}}{\langle (\textsf{assume}(e)\texttt{;}\ k), (s,h)\rangle \rightsquigarrow \langle k, (s,h)\rangle}}$$

Figure 2.4: Semantics of continuations. The semantic rule for "$\textsf{assume}(e)\texttt{;}\,k$" is included for clarity, but officially we consider "$\textsf{assume}(e)\texttt{;}\,k$" to be an abbreviation for "$\textsf{branch}\ e \Rightarrow k\ \textsf{end}$" (which produces the same result as the rule above).

state (s, h) or $\mathbf{error}$, which indicates that the program has terminated in the error state. Goto states have the form $\mathbf{goto}(l, (s, h))$ and indicate that execution should continue from label l in memory state (s, h) (the role of goto states is further described in Section 2.3, Definition 13). We use the meta-variable γ to represent an execution state and the meta-variable G to represent the set of all execution states.

Execution States (G) $\quad \gamma \quad ::= \quad \langle k, (s,h)\rangle \mid \mathbf{final}(s,h) \mid \mathbf{goto}(l, (s,h)) \mid \mathbf{error}$

We will sometimes simply use the word *state* when it is clear from context whether we are referring to an execution state or a memory state.

Note that in the semantics for branches given in figure 2.4, a non-deterministic choice is made among all branches whose condition is satisfied. There is no transition from a state in which we are evaluating a branch and none of the conditions hold. We will say more about how this property of the continuation semantics affects our program semantics in the next section when we discuss execution traces. Here we will simply note that, in the source programs we consider, all branches will be *total* in the sense that the disjunction of their conditions is equivalent to true. Thus, any execution state associated with a branch in the source program can always make a transition.

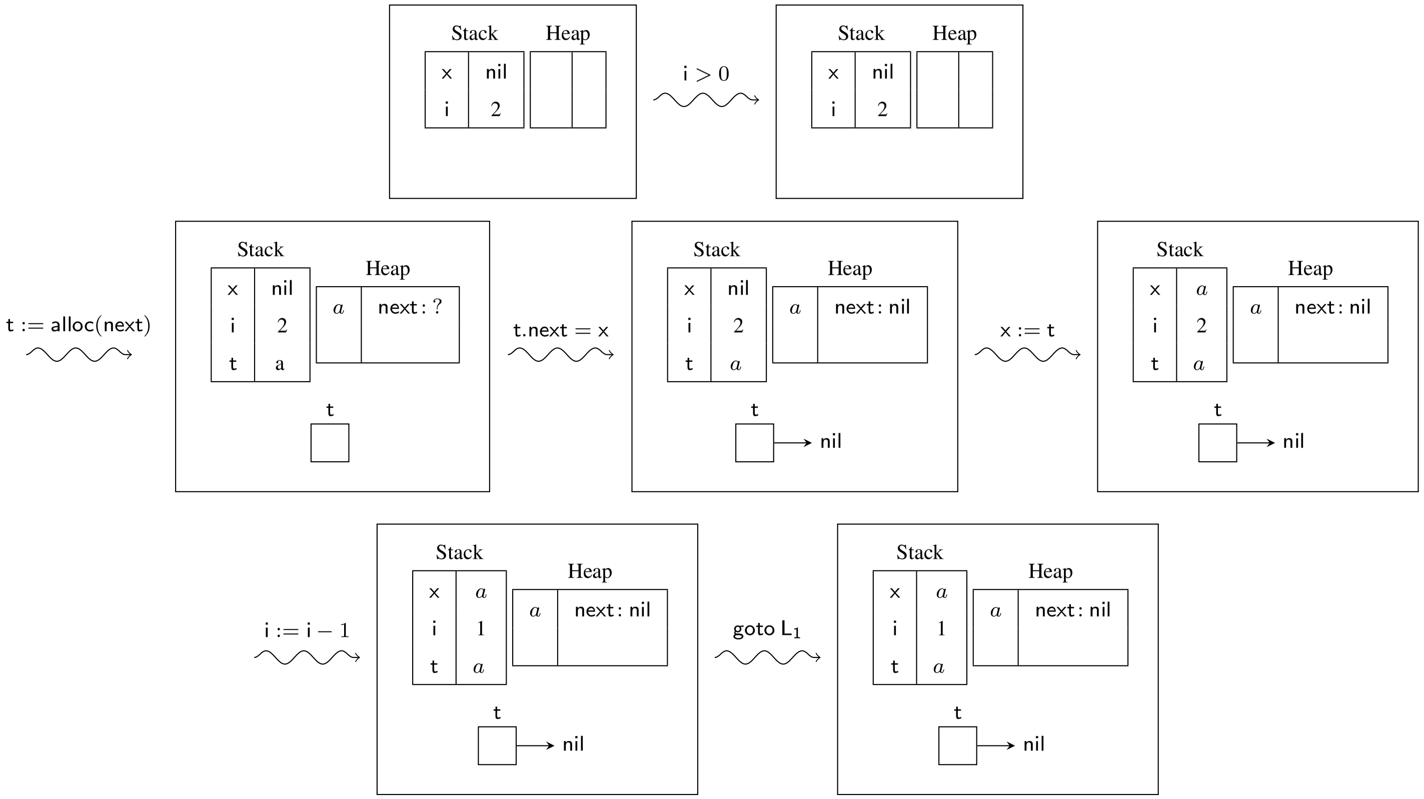
Figure 2.5 gives an example of the semantics of continuations. The arrows are labeled with the commands corresponding to the transitions. Transitions labeled with Boolean conditions ($i > 0$ in the first transition) correspond to the selection of the branch labeled with that condition.

2.2 Separation Logic

Note that all non-error states contain a store and a heap. We will use formulas in *separation logic* [Reynolds, 2002] to represent sets of store-heap pairs. The syntax for formulae is given in Figure 2.6 and describes a fragment of separation logic specialized to our heap model. The expressions (e) are those defined in Figure 2.1. $\mathcal{P}$ is a set of identifiers that are used to refer to inductively-defined predicates, which we discuss in Section 2.2.2.

The semantics of formulae is given in Figure 2.7. The semantics is given as a relation of the form $(s, h) \models_X Q$, where s is a store, h is a heap, Q is a separation logic formula and X is a partial mapping from inductive predicate names to the predicates' denotations (which are functions yielding sets of heaps). The relation $(s, h) \models_X Q$ is only defined when $dom(X)$ contains all predicate names appearing in Q. We describe inductive predicates in detail in the next section and focus on the other cases here. If $(s, h) \models_X Q$ holds for all s, h, we denote this as $\models_X Q$.

The formula $\mathbf{emp}$ describes the empty heap. The formula $x \mapsto [\mathsf{f_1} : e_1, \ ..., \ \mathsf{f_n} : e_n]$ describes a singleton heap where x points to a record whose $\mathsf{f_1}$ field contains the value of e_1 and so on (as with the syntax for branches, we omit the ϵ that terminates the field list when writing records). The field names $f_1, \ldots, f_n$ must be distinct. A store, heap pair (s, h) satisfies $Q_1 * Q_2$ iff h is a union of domain-disjoint heaps h_1 and h_2 such that (s, h_1) satisfies Q_1 and (s, h_2) satisfies Q_2. The binary operators $\wedge$ (conjunction), $\vee$ (disjunction), and $\Rightarrow$ (implication) have their usual semantics. For the binary operators,



```
L₁ : branch i > 0 ⇒ t := alloc(next); t.next = x;
                        x := t; i := i − 1; goto L₁,
            i = 0 ⇒ halt end
```

Figure 2.5: Iteration number one of a loop that creates a singly-linked list.

SYNTAX OF SEPARATION LOGIC FORMULAE

$$\begin{array}{rlll}
\textit{Inductive Predicates} & p^{\vec{\tau}}, r^{\vec{\tau}} & \in & \mathcal{P}^{\vec{\tau}} \\
\textit{Records} & \rho & ::= & \epsilon \mid f^{\tau} : e^{\tau},\ \rho \\
\textit{Spatial Predicates} & \Xi & ::= & \mathbf{emp} \mid e^{\mathrm{a}} \mapsto [\rho] \mid p^{\vec{\tau}}(\vec{e}^{\vec{\tau}}) \\
\textit{Separation Logic Formulae} & Q & ::= & e^{\mathrm{b}} \mid \Xi \mid Q_1 * Q_2 \mid Q_1 \wedge Q_2 \mid Q_1 \vee Q_2 \mid \\
 & & & Q_1 \Rightarrow Q_2 \mid \exists x^{\tau}.\, Q \mid \forall x^{\tau}.\, Q
\end{array}$$

Figure 2.6: Syntax of separation logic formulae.

the order of precedence, from strongest to weakest is: $\mapsto, *, \wedge, \vee, \Rightarrow$. The operators $\wedge, \vee$, and $*$ are associative, so order of operations among sequences of formulae joined by the same one of these operators at the same level does not matter.

We write $\vec{\tau}$ to represent the sequence of types $\tau_1\tau_2 \ldots \tau_n$. Meta-variables $p^{\vec{\tau}}$ and $r^{\vec{\tau}}$ represent the names of inductive predicates. The superscript $\vec{\tau}$ encodes both the number and types of the arguments the predicate expects. For example, p^{iaa} is a predicate that takes an integer-valued argument followed by two address-valued arguments. We write $\mathcal{P}^{\vec{\tau}}$ for the set of all predicates of type $\vec{\tau}$. If $\vec{\tau} = \tau_1 \ldots \tau_n$, we write $\vec{x}^{\,\vec{\tau}}$ to denote a list of variables $x_1^{\tau_1}, \ldots, x_n^{\tau_n}$. Similarly, we write $\vec{e}^{\,\vec{\tau}}$ to represent the list of expressions $e_1^{\tau_1}, \ldots, e_n^{\tau_n}$. We discuss inductive predicates further in the next section.

2.2.1 Effect of Free Variables

The free variables of a separation logic formula Q are defined in Figure 2.8. We have a result for separation logic formulae similar to Lemma 1, which involved expressions.

Lemma 4. *If $s =_V s'$ and $fv(Q) \subseteq V$ then for all X, h, we have $(s, h) \models_X Q$ if and only if $(s', h) \models_X Q$.*

Proof. The proof is by induction on the structure of Q.

SEMANTICS OF SEPARATION LOGIC FORMULAE

$$
\begin{array}{lcl}
\llbracket f^{\tau} : e^{\tau}, \rho \rrbracket\, s & = & \{(f^{\tau}, \llbracket e^{\tau} \rrbracket\, s)\} \cup (\llbracket \rho \rrbracket\, s) \\
\llbracket \epsilon \rrbracket\, s & = & \{\}
\end{array}
$$

$$
\begin{array}{lcl}
\llbracket e_1^{\tau_1}, \ldots, e_n^{\tau_n} \rrbracket\, s & = & (\llbracket e_1^{\tau_1} \rrbracket\, s, \ldots, \llbracket e_n^{\tau_n} \rrbracket\, s)
\end{array}
$$

$$
\begin{array}{lcl}
(s,h) \models_X e^{\mathrm{b}} & \Leftrightarrow & \llbracket e^{\mathrm{b}} \rrbracket\, s = \textbf{\textit{true}} \\
(s,h) \models_X \mathbf{emp} & \Leftrightarrow & h = \{\} \\
(s,h) \models_X e^{\mathrm{a}} \mapsto [\rho] & \Leftrightarrow & h = \{(\llbracket e^{\mathrm{a}} \rrbracket\, s, \llbracket \rho \rrbracket\, s)\} \\
(s,h) \models_X p^{\vec{\tau}}(\vec{e}^{\,\vec{\tau}}) & \Leftrightarrow & h \in \big(X(p^{\vec{\tau}})(\llbracket \vec{e}^{\,\vec{\tau}} \rrbracket\, s)\big) \\
(s,h) \models_X Q_1 \wedge Q_2 & \Leftrightarrow & (s,h) \models_X Q_1 \text{ and } (s,h) \models_X Q_2 \\
(s,h) \models_X Q_1 \vee Q_2 & \Leftrightarrow & (s,h) \models_X Q_1 \text{ or } (s,h) \models_X Q_2 \\
(s,h) \models_X Q_1 \Rightarrow Q_2 & \Leftrightarrow & (s,h) \models_X Q_1 \text{ implies } (s,h) \models_X Q_2 \\
(s,h) \models_X Q_1 * Q_2 & \Leftrightarrow & \text{There exist } h_1, h_2 \text{ such that} \\
 & & \mathit{dom}(h_1) \cap \mathit{dom}(h_2) = \emptyset \text{ and } h = h_1 \cup h_2 \text{ and} \\
 & & (s,h_1) \models_X Q_1 \text{ and } (s,h_2) \models_X Q_2 \\
(s,h) \models_X \exists x^{\mathrm{a/i}}.\, Q & \Leftrightarrow & \text{there exists } v \in \mathit{Addr}/\mathbb{Z} \text{ such that} \\
 & & (s[x^{\mathrm{a/i}} \to v], h) \models_X Q \\
(s,h) \models_X \forall x^{\mathrm{a/i}}.\, Q & \Leftrightarrow & \text{for all } v \in \mathit{Addr}/\mathbb{Z} \text{ we have} \\
 & & (s[x^{\mathrm{a/i}} \to v], h) \models_X Q
\end{array}
$$

$$
\models_X Q \;\Leftrightarrow\; \forall s,h.\, \big((s,h) \models_X Q\big)
$$

Figure 2.7: Semantics of separation logic formulae. We have combined the $\exists$ rules for address and integer-valued variables, using a "/" to separate the alternatives. The field names in any record ρ must be distinct. The semantics of expressions, $\llbracket e \rrbracket\, s$, is given in Figure 2.2.

$$\begin{array}{rcl@{\qquad}rcl}
\mathit{fv}(f^\tau : e^\tau,\ \rho) & = & \mathit{fv}(e) \cup \mathit{fv}(\rho) & \mathit{fv}(Q_1 * Q_2) & = & \mathit{fv}(Q_1) \cup \mathit{fv}(Q_2) \\
\mathit{fv}(\epsilon) & = & \{\} & \mathit{fv}(Q_1 \wedge Q_2) & = & \mathit{fv}(Q_1) \cup \mathit{fv}(Q_2) \\
 & & & \mathit{fv}(Q_1 \vee Q_2) & = & \mathit{fv}(Q_1) \cup \mathit{fv}(Q_2) \\
\mathit{fv}(\mathbf{emp}) & = & \{\} & \mathit{fv}(Q_1 \Rightarrow Q_2) & = & \mathit{fv}(Q_1) \cup \mathit{fv}(Q_2) \\
\mathit{fv}(e^{\mathrm{a}} \mapsto [\rho]) & = & \mathit{fv}(e^{\mathrm{a}}) \cup \mathit{fv}(\rho) & \mathit{fv}(\exists x^\tau.\, Q) & = & \mathit{fv}(Q) - \{x^\tau\} \\
\mathit{fv}(p^{\vec{\tau}}(e_1^{\tau_1} \ldots e_n^{\tau_n})) & = & \mathit{fv}(e_1^{\tau_1}) \cup \ldots \cup \mathit{fv}(e_n^{\tau_n}) & \mathit{fv}(\forall x^\tau.\, Q) & = & \mathit{fv}(Q) - \{x^\tau\}
\end{array}$$

Figure 2.8: The definition of the function $\mathit{fv}(Q)$, which gives the free variables of formula Q. If $Q = e^{\mathrm{b}}$, the free variables are as given in Definition 2.

CASE $Q = e^{\mathrm{b}}$: In this case, the definition of $\models_X$ from Figure 2.7 tells us that $(s, h) \models_X Q$ iff $[\![e^{\mathrm{b}}]\!]\, s = \textbf{\textit{true}}$. By Lemma 1 we then have that $[\![e^{\mathrm{b}}]\!]\, s = \textbf{\textit{true}}$ iff $[\![e^{\mathrm{b}}]\!]\, s' = \textbf{\textit{true}}$. This implies $(s, h) \models_X Q$ iff $(s', h) \models_X Q$.

CASE $Q = \mathbf{emp}$: In this case, $(s, h) \models_X \mathbf{emp}$ iff $h = \{\}$. Since s is not involved in the definition of the semantics of $\mathbf{emp}$, we easily have $(s, h) \models_X \mathbf{emp}$ iff $(s', h) \models_X \mathbf{emp}$.

CASE $Q = e^{\mathrm{a}} \mapsto [\rho]$: We first prove the following lemma:

$$\forall \rho, s, s'.\ (s =_V s') \wedge (\mathit{fv}(\rho) \subseteq V) \Rightarrow ([\![\rho]\!]\, s = [\![\rho]\!]\, s')$$

This is proved by structural induction on ρ. There are two cases. If $\rho = \epsilon$ then $[\![\rho]\!]\, s = \{\}$ and $[\![\rho]\!]\, s' = \{\}$, implying $[\![\rho]\!]\, s = [\![\rho]\!]\, s'$. If $\rho = f^\tau : e^\tau, \rho'$ then we have $[\![\rho]\!]\, s = \{(f^\tau, [\![e^\tau]\!]\, s)\} \cup ([\![\rho']\!]\, s)$. By the induction hypothesis we have $[\![\rho']\!]\, s = [\![\rho']\!]\, s'$. Since $\mathit{fv}(Q) \subseteq V$ we have that $\mathit{fv}(e^\tau) \subseteq V$ and thus by Lemma 1 we have $[\![e^\tau]\!]\, s = [\![e^\tau]\!]\, s'$. Combining these we have the following.

$$\{(f^\tau, [\![e^\tau]\!]\, s)\} \cup ([\![\rho']\!]\, s) = \{(f^\tau, [\![e^\tau]\!]\, s')\} \cup ([\![\rho']\!]\, s')$$

This is equivalent to $[\![\rho]\!]\, s = [\![\rho]\!]\, s'$, which is our goal.

Having proved the result for record expressions ρ, we can now turn back to Q. Since $\mathit{fv}(Q) \subseteq V$ and $Q = e^{\mathrm{a}} \mapsto [\rho]$, we have, as a consequence of Definition 2.2.1 that $\mathit{fv}(e^{\mathrm{a}}) \subseteq V$ and $\mathit{fv}(\rho) \subseteq V$. Thus, by Lemma 1 and by our intermediate lemma above, we have $[\![e^{\mathrm{a}}]\!]\, s = [\![e^{\mathrm{a}}]\!]\, s'$ and $[\![\rho]\!]\, s = [\![\rho]\!]\, s'$. This implies

$$\{([\![e^{\mathrm{a}}]\!]\, s, [\![\rho]\!] s)\} = \{([\![e^{\mathrm{a}}]\!]\, s', [\![\rho]\!] s')\}$$

which implies $(s, h) \models_X Q \Leftrightarrow (s', h) \models_X Q$ by the definition of $\models_X$ given in Figure 2.7.

CASE $Q = p^{\vec{\tau}}(\vec{e}^{\vec{\tau}})$: We first consider the forward implication. We assume $(s, h) \models p^{\vec{\tau}}(\vec{e}^{\vec{\tau}})$ and show $(s', h) \models p^{\vec{\tau}}(\vec{e}^{\vec{\tau}})$. We have from our semantics that $(s, h) \models p^{\vec{\tau}}(\vec{e}^{\vec{\tau}})$ implies $h \in \big(X(p)([\![\vec{e}^{\vec{\tau}}]\!]\, s)\big)$. Since $\mathit{fv}(\vec{e}^{\vec{\tau}}) \subseteq V$ we have by Lemma 1 that $[\![\vec{e}^{\vec{\tau}}]\!]\, s = [\![\vec{e}^{\vec{\tau}}]\!]\, s'$. This implies

$$\big(X(p)([\![\vec{e}^{\vec{\tau}}]\!]\, s)\big) = \big(X(p)([\![\vec{e}^{\vec{\tau}}]\!]\, s')\big)$$

Since we have $h \in \big(X(p)([\![\vec{e}^{\vec{\tau}}]\!]\, s)\big)$ this lets us conclude $h \in \big(X(p)([\![\vec{e}^{\vec{\tau}}]\!]\, s')\big)$ which implies $(s', h) \models Q$. The backward implication is the same with s and s' reversed.

CASE $Q = Q_1 * Q_2$: We have $(s, h) \models_X Q_1 * Q_2$ iff there exist h_1, h_2 such that $\mathit{dom}(h_1) \cap \mathit{dom}(h_2) = \emptyset$ and $h = h_1 \cap h_2$ and $(s, h_1) \models_X Q_1$ and $(s, h_2) \models_X Q_2$. That $\mathit{fv}(Q) \subseteq V$ implies $\mathit{fv}(Q_1) \subseteq V$ and $\mathit{fv}(Q_2) \subseteq V$. We can then apply the induction hypothesis, which gives us that $(s, h_1) \models_X Q_1$ iff $(s', h_1) \models_X Q_1$ and similarly for Q_2. This implies our result.

CASE $Q = Q_1 \wedge Q_2$: We have $(s, h) \models_X Q_1 \wedge Q_2$ iff $(s, h) \models_X Q_1$ and $(s, h) \models_X Q_2$. Again, $\mathit{fv}(Q) \subseteq V$ implies $\mathit{fv}(Q_1) \subseteq V$ and $\mathit{fv}(Q_2) \subseteq V$, allowing us to apply the inductive hypothesis and obtain $(s, h) \models_X Q_1$ iff $(s', h) \models_X Q_1$ (and similarly for $(s', h) \models_X Q_2$). This implies our result.

CASE $Q = Q_1 \vee Q_2$: This case is very similar to the $*$ and $\wedge$ cases. We have $(s, h) \models_X Q_1 \vee Q_2$ iff $(s, h) \models_X Q_1$ or $(s, h) \models_X Q_2$. In either case, we have $\mathit{fv}(Q_i) \subseteq V$ and apply our inductive hypothesis to obtain $(s, h) \models_X Q_i$ iff $(s', h) \models_X Q_i$, which lets us conclude that $(s, h) \models_X Q$ iff $(s', h) \models_X Q$.

CASE $Q = (Q_1 \Rightarrow Q_2)$: We will consider the forward direction first and show that $(s, h) \models_X (Q_1 \Rightarrow Q_2)$ implies $(s', h) \models_X (Q_1 \Rightarrow Q_2)$. Suppose $(s, h) \models_X (Q_1 \Rightarrow Q_2)$. Then by the definition of $\models_X$ given in Figure 2.7 we have $(s, h) \models_X Q_1$ implies $(s, h) \models_X Q_2$. Now, suppose $(s', h) \models_X Q_1$. Since $\mathit{fv}(Q) = \mathit{fv}(Q_1) \cup \mathit{fv}(Q_2)$ and

$\mathit{fv}(Q) \subseteq V$, we have $\mathit{fv}(Q_1) \subseteq V$ and $\mathit{fv}(Q_2) \subseteq V$. This lets us apply our inductive hypothesis, obtaining $(s,h) \models_X Q_1$. This implies $(s,h) \models_X Q_2$ by our assumption, which, applying the inductive hypothesis again, gives us $(s',h) \models_X Q_2$. Thus, we have shown that $(s',h) \models_X Q_1$ implies $(s',h) \models_X Q_2$, which lets us conclude $(s',h) \models_X (Q_1 \Rightarrow Q_2)$. The proof of the backwards direction is the same, with s and s' interchanged.

CASE $Q = \exists x.\, Q'$: We consider the forward direction first. The relation $(s,h) \models_X \exists x.\, Q'$ implies there exists a v such that $(s[x \to v],h) \models_X Q'$. Consider the store $s'[x \to v]$. Since $s =_V s'$, we have $s[x \to v] =_{V \cup \{x\}} s'[x \to v]$. We have that $\mathit{fv}(Q) = \mathit{fv}(Q') - \{x\}$ and $\mathit{fv}(Q) \subseteq V$ which implies $\mathit{fv}(Q') \subseteq V \cup \{x\}$. We can then apply our inductive hypothesis to $(s[x \to v],h) \models_X Q'$, obtaining $(s'[x \to v],h) \models_X Q'$. This implies $(s',h) \models_X \exists x.\, Q'$. The backward direction is the same, with s and s' interchanged.

CASE $Q = \forall x.\, Q'$: We consider the forward direction first. The relation $(s,h) \models_X \forall x.\, Q'$ implies that for all v we have $(s[x \to v],h) \models_X Q'$. Consider an arbitrary v'. Instantiating v above with v' we have $(s[x \to v'],h) \models_X Q'$. Since $s =_V s'$, we have $s[x \to v] =_{V \cup \{x\}} s'[x \to v]$. We have that $\mathit{fv}(Q) = \mathit{fv}(Q') - \{x\}$ and $\mathit{fv}(Q) \subseteq V$ which implies $\mathit{fv}(Q') \subseteq V \cup \{x\}$. We can then apply our inductive hypothesis to $(s[x \to v'],h) \models_X Q'$, obtaining $(s'[x \to v'],h) \models_X Q'$. Since v' was arbitrary, we conclude that for all v' we have $(s'[x \to v'],h) \models_X Q'$, which implies $(s',h) \models_X \forall x.\, Q'$. The backward direction is the same, with s and s' interchanged. □

2.2.2 Defining Inductive Pointer Structures

We follow an approach similar to Brotherston [2007] in our treatment of inductively-defined predicates. Pointer structures in our system are described inductively using definitions of the following form.

$$\textit{Definition List} \quad \mathcal{D} \quad ::= \quad \epsilon \mid \left(p^{\vec{\tau}}(\vec{x}^{\,\vec{\tau}}) \equiv Q\right) :: \mathcal{D}$$

The symbol ϵ represents an empty sequence of definitions. $\mathcal{D}$ then specifies a set of mutually inductive predicates. We require for each definition $p^{\vec{\tau}}(\vec{x}^{\,\vec{\tau}}) \equiv Q$ that all variables in $\vec{x}^{\,\vec{\tau}}$ are distinct, that $\mathit{fv}(Q) \subseteq \vec{x}$, and that all predicates $p^{\vec{\tau}}$ occurring to the left of $\equiv$ in

$\mathcal{D}$ are distinct. We also do not allow implication or universal quantification to appear in Q (and recall that Q also cannot contain negated spatial predicates according to the grammar in Figure 2.6).

As the constraints on type and arity of predicates and type and length of argument vectors are standard and generally clear from context, we will henceforth write predicates and vectors without mentioning arity or length except when necessary for clarity. For example, we will write $p(\vec{x})$ to represent $p^{\vec{\tau}}(\vec{x}^{\,\vec{\tau}})$ for some $\vec{\tau}$ implicitly given by context.

We will write $(p(\vec{x}) \equiv Q) \in \mathcal{D}$ when the definition $p(\vec{x}) \equiv Q$ appears in $\mathcal{D}$. We require that if $(p(\vec{x}) \equiv Q) \in \mathcal{D}$ and the predicate instance $p'(\vec{e}^{\,\vec{\tau}})$ appears in Q then $(p'(\vec{y}^{\,\vec{\tau}}) \equiv Q') \in \mathcal{D}$ for some $\vec{y}^{\,\vec{\tau}}$ and Q'. This ensures that all predicates referenced in the inductive definitions are defined. We write $dom(\mathcal{D})$ to refer to the set of predicates being defined by $\mathcal{D}$. This is defined inductively as follows.

$$\begin{aligned} dom((p(\vec{x}) \equiv Q) :: \mathcal{D}) &= \{p\} \cup dom(\mathcal{D}) \\ dom(\epsilon) &= \emptyset \end{aligned}$$

As an example of an inductive definition, consider the following definition of a doubly-linked list segment with length n starting at heap cell *first* and ending at *last*. The parameter $prev$ records the value of the prev field of the first cell in this list and $next$ records the value in the next field of the last cell.

$$\begin{aligned} &\mathrm{dll}(n, prev, \mathit{first}, last, next) \equiv \\ &\qquad \mathbf{emp} \wedge n = 0 \wedge \mathit{first} = next \wedge last = prev \\ &\quad \vee (\exists z.\ (\mathit{first} \mapsto [\mathsf{prev} : prev, \mathsf{next} : z]) \ast \\ &\qquad\qquad\qquad \mathrm{dll}(n-1, \mathit{first}, z, last, next)) \wedge n > 0 \end{aligned}$$

The disjunction indicates that there are two possible cases for a list segment with length n. Either $n = 0$ and the list is empty, or $n > 0$ and there is an allocated heap cell at the head of the list and a separate tail of length $n - 1$.

The semantics of inductive predicates is defined in terms of iterated expansion. We begin with the following definition.

Definition 4. *Let $o(\tau)$ be the function defined such that $o(\mathrm{a}) = \mathit{Addr}$ and $o(\mathrm{i}) = \mathbb{Z}$. We extend o to vectors, letting $o(\tau_1 \dots \tau_n) = o(\tau_1) \times \dots \times o(\tau_n)$.*

We then view an inductively-defined predicate of arity $\vec{\tau}$ as a function of type $o(\vec{\tau}) \to 2^{\mathit{Heaps}}$, which maps values for the parameters to the set of heaps that satisfy the predicate. We will call such a function an *interpretation function* and define this as follows.

Definition 5. *If N is a set of predicate names, the set of* ***interpretation functions*** *Δ_N is defined as follows.*

$$\Delta_N \overset{\text{def}}{=} \bigcup_{p^{\vec{\tau}} \in N} \left(\{p^{\vec{\tau}}\} \to \left(o(\vec{\tau}) \to 2^{\mathit{Heaps}} \right) \right)$$

In the type above, we use a union over functions with a singleton domain $\{p^{\vec{\tau}}\}$ to indicate that the range of the function depends on the type of $\vec{\tau}$ of the argument $p^{\vec{\tau}}$. Note that $dom(\Delta_N) = N$.

The meaning of a list of inductively defined predicates $\mathcal{D}$ is then an element of the set $\Delta_{dom(\mathcal{D})}$. We devote the remainder of the section to discussing appropriate elements of $\Delta_{dom(\mathcal{D})}$ to take as the semantics of $\mathcal{D}$.

Fixed-Point Semantics

Let $\mathcal{D}$ be the following list of inductive definitions

$$(p_1(\vec{x}_1) \equiv Q_1) :: \dots :: (p_n(\vec{x}_n) \equiv Q_n)$$

with the arity of p_i equal to $\vec{\tau}_i$. Let X be an element of $\Delta_{dom(\mathcal{D})}$. We will write $s[\vec{x} \to \vec{v}]$ for the store s' such that $s'(y) = v_i$ if $y \equiv x_i$ for some i and $s'(y) = s(y)$ otherwise. We use lambda notation to denote functions at the meta-level and write $\lambda\vec{v}.\ t$ as an abbreviation for $\lambda v_1.\ \lambda v_2.\ \dots \lambda v_n.\ t$ where t is some term in the meta-language. As always, we require that the types of the $\vec{x}$ and the domains from which the $\vec{v}$ are drawn match, so that if x_i has type a then $v_i \in \mathit{Addr}$ (and similarly for i and $\mathbb{Z}$). Let $\omega_{\mathcal{D}}$ be a function of type

$\Delta_{dom(\mathcal{D})} \to \Delta_{dom(\mathcal{D})}$ defined as follows.

$$\omega_{\mathcal{D}}(X) = \bigcup_{(p(\vec{x}) \equiv Q) \in \mathcal{D}} \big\{(p, Y) \mid Y = \lambda \vec{v}.\ \{h \mid \exists s.\ (s[\vec{x} \to \vec{v}], h) \models_X Q\}\big\}$$

Intuitively, this operator corresponds to taking X as the current approximation of the meaning of the definitions in $\mathcal{D}$, and adding the heaps that are satisfied when we expand the definitions once.

A fixed-point of $\omega_{\mathcal{D}}$ is any $X \in \Delta_{dom(\mathcal{D})}$ such that $\omega_{\mathcal{D}}(X) = X$. Any fixed-point of $\omega_{\mathcal{D}}$ may be taken as the meaning for a set of inductive definitions without introducing inconsistency into the system. The tool that we discuss in Chapter 5 makes no assumptions about which fixed-point has been chosen, and thus its conclusions are sound for all fixed-points. In order to formalize this, we introduce the following definition of satisfaction with respect to a set of inductive definitions.

Definition 6. *Let $\mathcal{D}$ be a set of inductive predicate definitions. Then we define satisfaction of Q with respect to $\mathcal{D}$ as follows.*

$$(s, h) \models^{\mathcal{D}} Q \ \textit{ iff } \ (s, h) \models_X Q \textit{ for all } X \in \Delta_{dom(\mathcal{D})} \textit{ such that } \omega_{\mathcal{D}}(X) = X$$

This will be the definition of satisfaction that we will use throughout the thesis as it most closely captures the behavior of our static analysis tool. However, it is important to ensure that the universal quantification in the definition above is not vacuously satisfied. If there are no fixed-points for $\omega_{\mathcal{D}}$, then $(s, h) \models^{\mathcal{D}} Q$ is trivially satisfied for all s, h, Q, i.e. the logic becomes inconsistent. We turn now to this issue, showing that $\omega_{\mathcal{D}}$ does in fact always have fixed-points. Furthermore, these fixed-points are partially ordered and there is always a *least* fixed-point with respect to this ordering.

Least Fixed-Points

We first prove the following lemma, which states that if the denotations of predicates given by X' include more states than those given by X, then satisfaction with respect to X implies satisfaction with respect to X'. The fact that implication is not allowed in inductive predicate definitions is crucial for this lemma.

Lemma 5. *Suppose $X \in \Delta_N$ and $X' \in \Delta_N$ for some N. Then*

$$\forall p, \vec{v}.\ (p \in N) \Rightarrow X(p)(\vec{v}) \subseteq X'(p)(\vec{v}) \tag{2.4}$$

implies

$$\forall s, h.\ \big((s, h) \models_X Q\big) \Rightarrow \big((s, h) \models_{X'} Q\big)$$

Proof. The proof is by induction on the structure of Q.

CASE Base Cases Not Involving Inductive Predicates: The base cases not involving inductive predicates are $Q = e^{\mathrm{b}}$, $Q = \mathbf{emp}$, and $Q = e^{\mathrm{a}} \mapsto [\rho]$. In each case, the satisfaction relation does not depend on the predicate meanings provided. For example, suppose $Q = e^{\mathrm{b}}$. Then we have $(s, h) \models_X e^{\mathrm{b}}$ which implies $[\![e^{\mathrm{b}}]\!]\, s = \textbf{\textit{true}}$. This then implies $(s, h) \models_{X'} e^{\mathrm{b}}$, which is our goal.

CASE Inductive Cases: Since we have disallowed implication in the body of inductive definitions, the inductive cases all follow directly from the inductive hypothesis. To give an example, suppose $Q = \exists x^{\mathrm{a}}.\ Q'$. Then we have $(s, h) \models_X \exists x^{\mathrm{a}}.\ Q'$ and must show $(s, h) \models_{X'} \exists x^{\mathrm{a}}.\ Q'$. According to the definition of satisfaction (Figure 2.7) our assumption implies that for some $v \in \mathit{Addr}$ we have $(s[x^{\mathrm{a}} \to v], h) \models_X Q$. Our inductive hypothesis then gives us $(s[x^{\mathrm{a}} \to v], h) \models_{X'} Q$. Thus, we have $(s[x^{\mathrm{a}} \to v], h) \models_{X'} Q$ for some $v \in \mathit{Addr}$ which implies $(s, h) \models_{X'} \exists x^{\mathrm{a}}.\ Q'$.

CASE Inductive Predicates: This is the only non-trivial case. We have $Q = p(\vec{e})$. According to the semantics in Figure 2.7 we have that $(s, h) \models_X p(\vec{e})$ implies $h \in \big(X(p)(s(\vec{e}))\big)$. As we have assumed that $(s, h) \models_X Q$ is only defined when the predicate names appearing in Q are in the domain of X, we also have that $p \in \mathit{dom}(X)$ which implies $p \in N$. We can now apply assumption (2.4) to obtain $h \in \big(X'(p)(s(\vec{e}))\big)$. This implies $(s, h) \models_{X'} p(\vec{e})$, which is our goal. □

We next show that the following lemma holds of our definition of $\omega_{\mathcal{D}}$. This will serve as the basis for establishing a monotonicity property.

Lemma 6. *Suppose $X \in \Delta_{\mathit{dom}(\mathcal{D})}$ and $X' \in \Delta_{\mathit{dom}(\mathcal{D})}$. Then*

$$\forall p, \vec{v}.\ (p \in \mathit{dom}(X)) \Rightarrow X(p)(\vec{v}) \subseteq X'(p)(\vec{v}) \tag{2.5}$$

implies

$$\forall p, \vec{v}.\ (p \in dom(\mathcal{D})) \Rightarrow \omega_{\mathcal{D}}(X)(p)(\vec{v}) \subseteq \omega_{\mathcal{D}}(X')(p)(\vec{v})$$

Proof. Assume $X \in \Delta_{dom(\mathcal{D})}$ and $X' \in \Delta_{dom(\mathcal{D})}$ and suppose we have

$$\forall p, \vec{v}.\ (p \in dom(X)) \Rightarrow X(p)(\vec{v}) \subseteq X'(p)(\vec{v})$$

Let p be an arbitrary predicate name in $dom(\mathcal{D})$ and $\vec{v}$ be a list of values. We must show

$$\omega_{\mathcal{D}}(X)(p)(\vec{v}) \subseteq \omega_{\mathcal{D}}(X')(p)(\vec{v}) \tag{2.6}$$

Expanding the definitions of $\omega_{\mathcal{D}}(X)(p)(\vec{v})$ and $\omega_{\mathcal{D}}(X')(p)(\vec{v})$ we obtain the following, where Q is the body of the definition of p (that is, $(p(\vec{x}) \equiv Q) \in \mathcal{D}$ for some $\vec{x}$).

$$\begin{aligned}\omega_{\mathcal{D}}(X)(p)(\vec{v}) &= \big\{h \bigm| \exists s.\ (s[\vec{x} \to \vec{v}], h) \models_X Q\big\}\\ \omega_{\mathcal{D}}(X')(p)(\vec{v}) &= \big\{h \bigm| \exists s.\ (s[\vec{x} \to \vec{v}], h) \models_{X'} Q\big\}\end{aligned}$$

Given these definitions, equation (2.6) is equivalent to the following.

$$\big\{h \bigm| \exists s.\ (s[\vec{x} \to \vec{v}], h) \models_X Q\big\} \subseteq \big\{h \bigm| \exists s.\ (s[\vec{x} \to \vec{v}], h) \models_{X'} Q\big\}$$

This holds if and only if the following holds for all h.

$$\big(\exists s.\ (s[\vec{x} \to \vec{v}], h) \models_X Q\big) \Rightarrow \big(\exists s.\ (s[\vec{x} \to \vec{v}], h) \models_{X'} Q\big)$$

This follows from Lemma 5. We have $(s[\vec{x} \to \vec{v}], h) \models_X Q$ for some s. By Lemma 5 and our assumption (2.5), we have

$$\forall s, h.\ \big((s, h) \models_X Q\big) \Rightarrow \big((s, h) \models_{X'} Q\big)$$

Applying the above with $s[\vec{x} \to \vec{v}]$ substituted for s then gives us $(s[\vec{x} \to \vec{v}], h) \models_{X'} Q$ which implies our goal of $\exists s.\ (s[\vec{x} \to \vec{v}], h) \models_{X'} Q$. □

A corollary of this lemma is that $\omega_{\mathcal{D}}$ is monotone with respect to $\sqsubseteq$, an ordering on functions defined as follows.

Definition 7. *Let X_1 and X_2 be elements in Δ_N for some N. Then we define the ordering $\sqsubseteq$ as follows.*

$$X_1 \sqsubseteq X_2 \textit{ iff } \forall p, \vec{v}.\ (p \in N) \Rightarrow X_1(p)(\vec{v}) \subseteq X_2(p)(\vec{v})$$

The set of names N will always be clear from context, so we do not include it in the notation for the order $\sqsubseteq$.

The monotonicity result is then the following.

Theorem 2. *If $X \in \Delta_{dom(\mathcal{D})}$ and $X' \in \Delta_{dom(\mathcal{D})}$ and $X \sqsubseteq X'$ then $\omega_{\mathcal{D}}(X) \sqsubseteq \omega_{\mathcal{D}}(X')$.*

Proof. We must show the following.

$$\forall p, \vec{v}.\ (p \in dom(\mathcal{D})) \Rightarrow \omega_{\mathcal{D}}(X)(p)(\vec{v}) \subseteq \omega_{\mathcal{D}}(X')(p)(\vec{v})$$

Our assumption that $X \sqsubseteq X'$ gives us the following.

$$\forall p, \vec{v}.\ (p \in dom(\mathcal{D})) \Rightarrow X(p)(\vec{v}) \subseteq X'(p)(\vec{v})$$

Applying Lemma 6 then yields our goal. □

Next, we define the following operation on sets of functions X_i.

Definition 8. *For any set $\{X_0, X_1, \ldots\}$ of functions in Δ_N, let $\bigsqcup_i X_i$ be defined as follows.*

$$\bigsqcup_i X_i = \left\{(p, \lambda\vec{v}.\ \bigcup_i X_i(p)(\vec{v})) \mid p \in N\right\}$$

This operation gives the supremum of the set $\{X_0, X_1, \ldots\}$.

Theorem 3. *$\bigsqcup_i X_i$ is the supremum of the set $\{X_0, X_1, \ldots\}$ with respect to the order $\sqsubseteq$.*

Proof. We must show that $\forall i.\ X_i \sqsubseteq \bigsqcup_i X_i$ and

$$\forall X.\ (\forall i.\ X_i \sqsubseteq X) \Rightarrow \bigsqcup_i X_i \sqsubseteq X$$

or informally, that $\bigsqcup_i X_i$ is an upper bound and that it is the least upper bound.

Upper Bound We first show $\forall i.\ X_i \sqsubseteq \bigsqcup_i X_i$. Choose some X_j. We must show that $X_j \sqsubseteq \bigsqcup_i X_i$. This holds if $\forall p, \vec{v}.\ (p \in N) \Rightarrow X_j(p)(\vec{v}) \subseteq (\bigsqcup_i X_i)(p)(\vec{v})$. Expanding the definition of $\bigsqcup_i X_i$ and applying the function, we have to show the following.

$$\forall p, \vec{v}.\ (p \in N) \Rightarrow \left(X_j(p)(\vec{v}) \subseteq \bigcup_i (X_i(p)(\vec{v}))\right)$$

This holds since $\bigcup_i X_i(p)(\vec{v})$ contains $X_j(p)(\vec{v})$ (there is some i in this union such that $i = j$ which guarantees the inclusion).

Least Upper Bound We now show the following.

$$\forall X.\ (\forall i.\ X_i \sqsubseteq X) \Rightarrow \bigsqcup_i X_i \sqsubseteq X$$

We consider some X such that $(\forall i.\ X_i \sqsubseteq X)$ and show $\bigsqcup_i X_i \sqsubseteq X$. We must show the following.

$$\forall p, \vec{v}.\ (p \in N) \Rightarrow (\bigsqcup_i X_i)(p)(\vec{v}) \subseteq X(p)(\vec{v})) \tag{2.7}$$

Our assumption $(\forall i.\ X_i \sqsubseteq X)$ implies the following.

$$\forall p, \vec{v}.\ (p \in N) \Rightarrow \forall i.\ X_i(p)(\vec{v}) \subseteq X(p)(\vec{v}) \tag{2.8}$$

Expanding the definition of $\bigsqcup_i(X_i)$ in (2.7) and reducing the function application, we find that we must show

$$\forall p, \vec{v}.\ (p \in N) \Rightarrow \bigcup_i (X_i(p)(\vec{v})) \subseteq X(p)(\vec{v}))$$

This follows from (2.8) and the fact that $\bigcup_i (X_i(p)(\vec{v}))$ is the supremum of the set $\{X_1(p)(\vec{v}), X_2(p)(\vec{v}), \ldots\}$. □

That $\omega_{\mathcal{D}}$ is monotone with respect to $\sqsubseteq$ and $\bigsqcup$ is the supremum with respect to $\sqsubseteq$ implies that $\omega_{\mathcal{D}}$ has a least fixed-point.

Theorem 4. *$\omega_{\mathcal{D}}$ has a least fixed-point.*

Proof. We first note that Theorem 3 implies that the lattice of interpretation functions X is complete. The current theorem then follows from Lemma 2 and application of the Tarski fixed-point theorem. □

Continuity Let $\bot = \{(p, \lambda\vec{x}.\ \emptyset) \mid p \in dom(\mathcal{D})\}$. Not only does $\omega_{\mathcal{D}}$ have a least fixed-point, but this fixed-point is the least upper bound of the increasing chain $\omega_{\mathcal{D}}^0, \omega_{\mathcal{D}}^1, \ldots$, where $\omega_{\mathcal{D}}^i$ for $i \in \mathbb{N}$ is defined as follows.

$$\begin{array}{rcl} \omega_{\mathcal{D}}^0 & = & \bot \\ \omega_{\mathcal{D}}^{i+1} & = & \omega_{\mathcal{D}}(\omega_{\mathcal{D}}^i) \end{array}$$

This is captured by the following theorems. These all rely on the fact that universal quantification is not permitted in inductive predicate definitions.

Theorem 5. *$\omega_{\mathcal{D}}$ is continuous.*

Proof. We have shown that $\bigsqcup$ is the least upper-bound. We must show that $\omega_{\mathcal{D}}$ preserves least upper-bounds of directed sets (the definition of Scott continuity). Consider a set $\mathbf{X}$ of functions in $\Delta_{dom(\mathcal{D})}$ such that for all i, j, if $X_i \in \mathbf{X}$ and $X_j \in \mathbf{X}$ then $\exists X_k.\ X_k \in \mathbf{X} \wedge X_i \sqsubseteq X_k \wedge X_j \sqsubseteq X_k$ (that is, $\mathbf{X}$ is a directed set). We must show that $\omega_{\mathcal{D}}(\bigsqcup \mathbf{X}) = \bigsqcup(\omega_{\mathcal{D}}(\mathbf{X}))$ where $\omega_{\mathcal{D}}(\mathbf{X}) = \{\omega_{\mathcal{D}}(X) \mid X \in \mathbf{X}\}$. Expanding the definition of $\omega_{\mathcal{D}}$, we have the following for the left side of the equality.

$$\bigcup_{(p(\vec{x}) \equiv Q) \in \mathcal{D}} \big\{(p, Y) \mid Y = \lambda\vec{v}.\ \{h \mid \exists s.\ (s[\vec{x} \to \vec{v}], h) \models_{\bigsqcup \mathbf{X}} Q\}\big\}$$

The right side becomes the following

$$\bigsqcup \left\{ \bigcup_{(p(\vec{x}) \equiv Q) \in \mathcal{D}} \big\{(p, Y) \mid Y = \lambda\vec{v}.\ \{h \mid \exists s.\ (s[\vec{x} \to \vec{v}], h) \models_X Q\}\big\} \;\middle|\; X \in \mathbf{X} \right\}$$

Applying the definition of $\bigsqcup$ (Definition 7), the right side expands to the following.

$$\bigcup_{(p(\vec{x}) \equiv Q) \in \mathcal{D}} \big\{(p, Y) \mid Y = \lambda\vec{v}.\ \bigcup_i \{h \mid \big(\exists s.\ (s[\vec{x} \to \vec{v}], h) \models_{X_i} Q\big) \wedge X_i \in \mathbf{X}\}\big\}$$

Continuity will then be implied if we can show the following for all Q of our restricted form (formulas not containing implication or universal quantification).

$$\Big(\{h \mid \exists s.\ (s[\vec{x} \to \vec{v}], h) \models_{\bigsqcup \mathbf{X}} Q\}\Big) = \Big(\bigcup_i \{h \mid \big(\exists s.\ (s[\vec{x} \to \vec{v}], h) \models_{X_i} Q\big) \wedge X_i \in \mathbf{X}\}\Big)$$

Since an element is in the set on the left of the equality exactly when it is in some set being unioned on the right, we have that the statement above holds if and only if we have the following for all h.

$$\Big(\exists s.\ (s[\vec{x} \to \vec{v}], h) \models_{\bigsqcup \mathbf{X}} Q\Big) \Leftrightarrow \Big(\exists X_i \in \mathbf{X}.\ \big(\exists s.\ (s[\vec{x} \to \vec{v}], h) \models_{X_i} Q\big)\Big)$$

The right-to-left direction of the implication follows immediately from Lemma 5 and the fact that for all $X_i \in \mathbf{X}$ we have $X_i \sqsubseteq \bigsqcup \mathbf{X}$.

We show the left-to-right direction by showing the following, stronger statement by induction on the structure of Q.

$$\begin{aligned} &\forall s.\ \Big((s[\vec{x} \to \vec{v}], h) \models_{\bigsqcup \mathbf{X}} Q\Big) \Rightarrow \\ &\qquad\qquad \exists s'.\ (s =_{\mathit{fv}(Q)} s') \wedge \Big(\exists X_i \in \mathbf{X}.\ \big((s'[\vec{x} \to \vec{v}], h) \models_{X_i} Q\big)\Big) \end{aligned}$$

CASE Base Cases Not Involving Inductive Predicates: The base cases not involving inductive predicates are $Q = e^{\mathrm{b}}$, $Q = \mathbf{emp}$, and $Q = e^{\mathrm{a}} \mapsto [\rho]$. In each case, the satisfaction relation does not depend on the predicate meanings provided. For example, suppose $Q = e^{\mathrm{b}}$. Then we have $(s[\vec{x} \to \vec{v}], h) \models_{\bigsqcup \mathbf{X}} e^{\mathrm{b}}$, which is true if and only if $[\![e^{\mathrm{b}}]\!]\,(s[\vec{x} \to \vec{v}]) = \textbf{\textit{true}}$. This implies $(s[\vec{x} \to \vec{v}], h) \models_{X_i} e^{\mathrm{b}}$ for all X_i, thus implying our goal (we trivially have $s =_{\mathit{fv}(Q)} s$, which is the other potion of the goal formula).

CASE $Q = Q_1 * Q_2$: We assume that we have the following.

$$(s[\vec{x} \to \vec{v}], h) \models_{\bigsqcup \mathbf{X}} Q_1 * Q_2$$

The semantics of $\models_{\bigsqcup \mathbf{X}}$ then implies that there exist heaps h_1 and h_2 such that $\mathit{dom}(h_1) \cap \mathit{dom}(h_2) = \emptyset$ and $h = h_1 \cup h_2$ and $(s[\vec{x} \to \vec{v}], h_1) \models_{\bigsqcup \mathbf{X}} Q_1$ and $(s[\vec{x} \to \vec{v}], h_2) \models_{\bigsqcup \mathbf{X}} Q_2$. Our inductive hypothesis then gives us the following

$$\exists s'.\ (s =_{\mathit{fv}(Q_1)} s') \wedge \exists X_i \in \mathbf{X}.\ \big((s'[\vec{x} \to \vec{v}], h_1) \models_{X_i} Q_1\big)$$

and

$$\exists s''.\ (s =_{fv(Q_2)} s'') \land \exists X_j \in \mathbf{X}.\ \big((s''[\vec{x} \to \vec{v}], h_2) \models_{X_j} Q_2\big)$$

Let s' and s'' be as above. Since $s =_{fv(Q_1)} s'$ and $s =_{fv(Q_2)} s''$ we can apply Lemma 4 to the formulas above to obtain

$$\exists X_i \in \mathbf{X}.\ \big((s[\vec{x} \to \vec{v}], h_1) \models_{X_i} Q_1\big)$$

and

$$\exists X_j \in \mathbf{X}.\ \big((s[\vec{x} \to \vec{v}], h_2) \models_{X_j} Q_2\big)$$

Let X_i and X_j be the functions whose existence is stated in the formulas above. Then the assumption that $\mathbf{X}$ is directed implies that there is some X_k such that $X_k \in \mathbf{X}$ and $X_i \sqsubseteq X_k$ and $X_j \sqsubseteq X_k$. Lemma 5 then gives us $(s[\vec{x} \to \vec{v}], h_1) \models_{X_k} Q_1$ and $(s[\vec{x} \to \vec{v}], h_1) \models_{X_k} Q_1$. We can then combine these and apply the definition of $\models_{X_k}$ (Figure 2.7) to conclude the following, which is the second conjunct of our goal.

$$\exists X_k \in \mathbf{X}.\ \big((s[\vec{x} \to \vec{v}], h_2) \models_{X_k} Q_1 * Q_2\big)$$

The first conjunct of the goal is $s =_{fv(Q)} s$, which is immediate.

CASE $Q = Q_1 \land Q_2$ and $Q = Q_1 \lor Q_2$: These cases are very similar to the case above. For $Q_1 \land Q_2$, we have the assumption below.

$$(s[\vec{x} \to \vec{v}], h) \models_{\bigsqcup \mathbf{X}} Q_1 \land Q_2$$

Applying the definition of $\models_{\bigsqcup \mathbf{X}}$ gives us $(s[\vec{x} \to \vec{v}], h) \models_{\bigsqcup \mathbf{X}} Q_1$ and $(s[\vec{x} \to \vec{v}], h) \models_{\bigsqcup \mathbf{X}} Q_2$. Applying the inductive hypothesis yields $(s'[\vec{x} \to \vec{v}], h) \models_{X_i} Q_1$ and $(s''[\vec{x} \to \vec{v}], h) \models_{X_j} Q_1$ where $s =_{fv(Q_1)} s'$ and $s =_{fv(Q_2)} s''$. Applying Lemma 4 yields $(s[\vec{x} \to \vec{v}], h) \models_{X_i} Q_1$ and $(s[\vec{x} \to \vec{v}], h) \models_{X_j} Q_2$. Let X_k be the upper bound of X_i and X_j. We then have $(s[\vec{x} \to \vec{v}], h) \models_{X_k} Q_1$ and $(s[\vec{x} \to \vec{v}], h) \models_{X_k} Q_2$, which implies $(s[\vec{x} \to \vec{v}], h) \models_{X_k} Q_1 \land Q_2$, which is our goal.

For $Q_1 \lor Q_2$ the proof is similar except that we only have one of $(s[\vec{x} \to \vec{v}], h) \models_{\bigsqcup \mathbf{X}} Q_1$ or $(s[\vec{x} \to \vec{v}], h) \models_{\bigsqcup \mathbf{X}} Q_2$. Without loss of generality, suppose it is $(s[\vec{x} \to \vec{v}], h) \models_{\bigsqcup \mathbf{X}} Q_1$ that holds. We then apply the inductive hypothesis, obtaining $(s'[\vec{x} \to \vec{v}], h) \models_{X_i} Q_1$ and

$s =_{fv(Q_1)} s'$. Let s'' be defined such that $s''(x) = s'(x)$ if $x \in fv(Q_1)$ and $s''(x) = s(x)$ otherwise. Consider some $y \in fv(Q_1 \vee Q_2)$. There are two cases. If $y \in fv(Q_1)$, then we have $s''(y) = s'(y)$ and, due to $s' =_{fv(Q_1)} s$, we also have $s''(y) = s(y)$. If $y \notin fv(Q_1)$ then we have $s''(y) = s(y)$ by the definition of s''. Thus we have shown $s =_{fv(Q)} s''$. By Lemma 4 we also have $(s''[\vec{x} \to \vec{v}], h) \models_{X_i} Q_1$. Thus we have shown our goal.

CASE $Q = \exists y.\ Q_1$: We first assume that y is distinct from all elements of $\vec{x}$. This can always be made to hold via α-conversion. We have from the semantics of existential quantification that there is some v_y such that $((s[\vec{x} \to \vec{v}])[y \to v_y], h) \models_{\bigsqcup \mathbf{X}} Q_1$. As y is distinct from all elements of $\vec{x}$, we have that $(s[\vec{x} \to \vec{v}])[y \to v_y] = (s[y \to v_y])[\vec{x} \to \vec{v}]$. We can then apply our inductive hypothesis with $s = s[y \to v_y]$. This yields $\exists s'.\ (s'[\vec{x} \to \vec{v}], h) \models_{X_i} Q_1$ for some X_i and $s =_{fv(Q_1)} s'$. By the case for existentials in the semantics of $\models_{X_i}$, this then implies $\exists s'.\ (s'[\vec{x} \to \vec{v}], h) \models_{X_i} \exists y.\ Q_1$, which is the second conjunct of our goal. The first conjunct, $s =_{fv(Q)} s'$, is implied by our assumption $s =_{fv(Q_1)} s'$ and the fact that $fv(Q_1) \supseteq fv(Q)$.

CASE $Q = p(\vec{e})$: In this case, we have $(s[\vec{x} \to \vec{v}], h) \models_{\bigsqcup \mathbf{X}} p(\vec{y})$. The semantics for $\models_{\bigsqcup \mathbf{X}}$ from Figure 2.7 then gives us

$$h \in \left(\bigsqcup \mathbf{X}(p)(\llbracket e \rrbracket\ s[\vec{x} \to \vec{v}]\right)$$

Applying the definition of $\bigsqcup$, this implies the following, where $X_i \in \mathbf{X}$.

$$h \in \bigcup_i \left(X_i(p)(\llbracket e \rrbracket\ s[\vec{x} \to \vec{v}])\right)$$

This implies that there is some $X_j \in \mathbf{X}$ such that $h \in X_j(p)(\llbracket e \rrbracket\ s[\vec{x} \to \vec{v}])$. Again applying the semantics from Figure 2.7, we obtain

$$(s[\vec{x} \to \vec{v}], h) \models_{X_j} p(\vec{e})$$

We clearly have $s =_{fv(Q)} s$, so introducing an existential on s then gives us our goal. □

Theorem 6. *Let $\bot_N = \{(p, \lambda\vec{x}.\ \emptyset) \mid p \in N\}$. Then $\bot_N$ is the least element of Δ_N with respect to $\sqsubseteq$.*

Proof. We will show that for all X in Δ_N we have $\bot_N \sqsubseteq X$. Consider an arbitrary $X \in \Delta_N$. Expanding the definition of $\sqsubseteq$, we must show that

$$\forall p, \vec{v}.\ (p \in N) \Rightarrow \bot_N(p)(\vec{v}) \subseteq X(p)(\vec{v})$$

Suppose $p \in N$ and choose an arbitrary $\vec{v}$. Expanding the definition of $\bot_N$, we must show $\emptyset \subseteq X(p)(\vec{v})$. But this is immediate since $\emptyset$ is the least element with respect to $\subseteq$. □

Theorem 7. *The least fixed-point of $\omega_{\mathcal{D}}$ is $\bigsqcup\{\omega^i_{\mathcal{D}} \mid i \in \mathbb{N}\}$, where $\omega^i_{\mathcal{D}}$ is defined as follows.*

$$\begin{aligned} \omega^0_{\mathcal{D}} &= \bot_{dom(\mathcal{D})} \\ \omega^{i+1}_{\mathcal{D}} &= \omega_{\mathcal{D}}(\omega^i_{\mathcal{D}}) \end{aligned} \tag{2.9}$$

Proof. This follows from Theorem 6, Theorem 5, and Scott's fixed-point theorem. □

Least Fixed-point Semantics of Satisfaction The benefit of the theory of least fixed-points developed above is two-fold. First, it ensures that fixed-points exist and thus that Definition 6 does not vacuously hold. Furthermore, least fixed-points are often taken as the semantics of inductive definitions. Rather than Definition 6, we could have introduced the following.

Definition 9 (Alternate Satisfaction Relation). *Let $\mathcal{D}$ be a set of inductive predicate definitions and let $lfp(\omega_{\mathcal{D}})$ be the least fixed-point of $\omega_{\mathcal{D}}$ with respect to $\sqsubseteq$. Then we define least fixed-point satisfaction of Q with respect to inductive definitions $\mathcal{D}$ as follows.*

$$(s, h) \mid\models^{\mathcal{D}} Q \ \textit{ iff } \ (s, h) \models_{lfp(\omega_{\mathcal{D}})} Q$$

The development in this thesis does not depend on which fixed-point is taken as the meaning of a set of inductive predicates and could be carried out with either Definition 6 or Definition 9. We chose Definition 6 since it is more general, in the sense that $(s, h) \models^{\mathcal{D}} Q$ implies $(s, h) \mid\models^{\mathcal{D}} Q$. This ensures that all results given in terms of the satisfaction relation in Definition 6 also hold for the definition of satisfaction in terms of least fixed-points (Definition 9).

Example Let $\mathcal{D}$ be the definition list containing the single inductively-defined predicate below.

$$\begin{aligned} &\mathit{ls}(n, \mathit{start}, \mathit{end}) \equiv \\ &\qquad (\mathbf{emp} \wedge \mathit{start} = \mathit{end} \wedge n = 0) \\ &\quad \vee (n > 0 \wedge (\exists z.\, (\mathit{start} \mapsto [\mathsf{next} : z]) * \mathit{ls}(n-1, z, \mathit{end}))) \end{aligned}$$

Then $\mathit{lfp}(\omega_{\mathcal{D}})$ is the function that maps ls to the following function (where $\#(S)$ represents the cardinality of set S).

$$\begin{aligned} \lambda(n, s, e).\, \big\{h \;\big|\; &\#(\mathit{dom}(h)) = n \wedge \\ &\exists a_1, \ldots, a_n.\, s = a_1 \wedge e = a_n \wedge \\ &(\forall i.\, 1 \leq i < n \Rightarrow (a_i \in \mathit{dom}(h) \wedge h(a_i) = \{(\mathsf{next}, a_{i+1})\}))\big\} \end{aligned}$$

This maps the tuple (n, s, e) to the set of heaps containing only cells that are structured as a solitary singly-linked list segment of length n. Examples of such heaps are the empty heap $\{\}$, the singleton heap $\{(s, \{(\mathsf{next}, e)\})\}$ and the heap below, which contains a list segment of length 3 (in the set below, a_0 and a_1 must be chosen such that a_0, a_1 and s are all distinct).

$$\{(s, \{(\mathsf{next}, a_0)\}), (a_0, \{(\mathsf{next}, a_1)\}), (a_1, \{(\mathsf{next}, e)\})\}$$

Defining Inductive Predicates With Characteristic Formulae

An alternative to defining an inductive predicate symbol as above is to describe it in terms of the properties it satisfies. The key property of an inductive definition is that the interpretation of the definition should establish an equivalence between the predicate and the body of the definition. In fact, we will show in this section that requiring the predicate to satisfy this equivalence is just the same as defining it via fixed-points as we did before. We present this alternate approach because it more closely matches the reasoning performed by the tool we have developed (which is described in Chapter 5).

First we define the *characteristic formula* associated with a definition. This is the equivalence that we expect the interpretation of the predicate to satisfy.

Definition 10. *Let the **characteristic formula** of a set of inductive definitions $\mathcal{D}$, denoted $\lceil \mathcal{D} \rceil$, be defined as follows.*

$$\lceil p_1(\vec{x}_1) \equiv Q_1 :: \ldots :: p_n(\vec{x}_n) \equiv Q_n \rceil \overset{\text{def}}{=} \\ (\forall \vec{x}_1.\, p_1(\vec{x}_1) \Leftrightarrow Q_1) \wedge \ldots \wedge (\forall \vec{x}_n.\, p_n(\vec{x}_n) \Leftrightarrow Q_n)$$

Then we can show the following, which states that the set of fixed-points of $\mathcal{D}$ is exactly the set of interpretations satisfying the characteristic formula of $\mathcal{D}$. Recall that $\models Q$ holds if and only if $(s, h) \models Q$ holds for all s, h.

Theorem 8. *For all $s, h, \mathcal{D}, Q$, we have $(s, h) \models^{\mathcal{D}} Q$ if and only if $(s, h) \models_X Q$ holds for all $X \in \Delta_{dom(\mathcal{D})}$ such that $\models_X \lceil \mathcal{D} \rceil$.*

Proof. We first note that the definition of $(s, h) \models^{\mathcal{D}} Q$ states that $(s, h) \models_{X'} Q$ for all X' such that $\omega_{\mathcal{D}}(X') = X'$. We can complete the proof by showing that $\omega_{\mathcal{D}}(X) = X$ if and only if $\models_X \lceil \mathcal{D} \rceil$.

Let $\mathcal{D} = p_1(\vec{x}_1) \equiv Q_1 :: \ldots :: p_n(\vec{x}_n) \equiv Q_n$. Then $\lceil \mathcal{D} \rceil$ is the formula below.

$$(\forall \vec{x}_1.\, p_1(\vec{x}_1) \Leftrightarrow Q_1) \wedge \ldots \wedge (\forall \vec{x}_n.\, p_n(\vec{x}_n) \Leftrightarrow Q_n)$$

Since we have $\models_X \lceil \mathcal{D} \rceil$, this implies that for all s, h we have

$$(s, h) \models_X (\forall \vec{x}_1.\, p_1(\vec{x}_1) \Leftrightarrow Q_1) \wedge \ldots \wedge (\forall \vec{x}_n.\, p_n(\vec{x}_n) \Leftrightarrow Q_n)$$

Applying the semantics of satisfaction from Figure 2.7, we then have the following for each $s, h, i, \vec{v}$.

$$(s[\vec{x}_i \to \vec{v}], h) \models_X (p_i(\vec{x}_i) \Leftrightarrow Q_i) \tag{2.10}$$

We must show that $\omega_{\mathcal{D}}(X) = X$ implies the formula above for each $s, h, i, \vec{v}$, as well as the reverse implication. We have that $\omega_{\mathcal{D}}(X) = X$ if and only if $\big(\omega_{\mathcal{D}}(X)\big)(p_i)(\vec{v}) = X(p_i)(\vec{v})$ for all $p_i \in dom(\mathcal{D})$. Expanding $\omega_{\mathcal{D}}$ in the previous formula, we obtain the following for each i.

$$\{h \mid \exists s.\, (s[\vec{x}_i \to \vec{v}], h) \models_X Q_i\} = X(p_i)(\vec{v}) \tag{2.11}$$

We now show that (2.10) holds if and only (2.11) does, thus completing the proof. Suppose (2.10) holds. Then we have $(s[\vec{x}_i \to \vec{v}], h) \models_X p_i(\vec{x}_i)$ if and only if $(s[\vec{x}_i \to \vec{v}], h) \models_X Q_i$. Expanding the definition of satisfaction, we obtain $h \in X(p_i)(\llbracket \vec{x}_i \rrbracket\ s[\vec{x}_i \to \vec{v}])$ if and only if $(s[\vec{x}_i \to \vec{v}], h) \models_X Q_i$ or, simplifying further, the following.

$$h \in X(p_i)(\vec{v}) \text{ iff } (s[\vec{x}_i \to \vec{v}], h) \models_X Q_i$$

This holds if and only if

$$X(p_i)(\vec{v}) = \{h \mid (s[\vec{x}_i \to \vec{v}], h) \models_X Q_i\}$$

To show our goal (2.11) we must show that $(s[\vec{x}_i \to \vec{v}], h) \models_X Q_i$ if and only if $\exists s.\ (s[\vec{x}_i \to \vec{v}], h) \models_X Q_i$. The forward direction is immediate. The backward direction follows from Lemma 4 and the fact that, since Q_i is the body of an inductive definition with arguments $\vec{x}_i$, we have $\mathit{fv}(Q_i) \subseteq \vec{x}_i$. Since $s[\vec{x}_i \to \vec{v}] =_{\vec{x}_i} s'[\vec{x}_i \to \vec{v}]$ for any s, s', the Lemma allows us to assume the existence of some s' such that $(s'[\vec{x}_i \to \vec{v}], h) \models_X Q_i$ and conclude that $(s[\vec{x}_i \to \vec{v}], h) \models_X Q_i$. □

We will see the utility of this theorem when we discuss our implementation's treatment of inductive predicates in Section 5.2.

Induction Induction is commonly used to prove properties of inductively defined structures. Least fixed-points come with a built-in induction principle based on the construction given in Theorem 7. When working in the context of the satisfaction relation given as Definition 6, we do not have this principle available. However, we can still use mathematical induction over the naturals as a justification for inductive proofs. For example, given the list segment predicate ls from our example (page 44), we can show the following by induction on n_1.

$$\forall n_1, n_2, x, y, z.\ \mathit{ls}(n_1, x, y) * \mathit{ls}(n_2, y, z) \Rightarrow \mathit{ls}(n_1 + n_2, x, z)$$

Even when there is no parameter present that is suitable for induction, we can still use induction over the size of satisfying heaps to prove properties of our data structures.

2.3 Semantics of Programs

A program can be viewed as defining a *transition system*. In this section we first give the general definitions related to transition systems and then discuss the interpretation of a program as a transition system.

2.3.1 Transition Systems

Definition 11. *A* ***transition system*** S *is a tuple* $(A, I, F, \dashrightarrow)$ *where* A *is a set of states,* $I \subseteq A$ *is a set of initial states,* $F \subseteq A$ *is a set of final states, and* $\dashrightarrow \subseteq A \times A$ *is a transition relation.*

Each transition system defines a set of *traces*, which are sequences of states where adjacent states are related by the transition relation. We use the following standard notation for sequences.

ϵ is the empty sequence.

γ is a sequence consisting of one element—the execution state γ.

If T_1 and T_2 are sequences, then $T_1\,T_2$ is the sequence that results from concatenating T_1 and T_2. If T_1 is infinite, then $T_1\,T_2 = T_1$.

$\gamma \in T$ holds iff $\exists T_1, T_2.\ T = T_1\,\gamma\,T_2$.

$\mathit{len}(T)$ is the length of sequence T. If T is finite this is the number of elements in T. If T is infinite, then $\mathit{len}(T) = \omega$.

$T(i)$ is the i^{th} element of T, with the first element given by $T(0)$. This is only defined if $0 \leq i < \mathit{len}(T)$. The last element of a finite sequence T is given by $T(\mathit{len}(T)-1)$.

T_n is the trace obtained by discarding the first n elements of trace T. That is, if $T = \gamma_0\,\gamma_1\,\ldots\,\gamma_{n-1}\,T'$ then $T_n = T'$. If $\mathit{len}(T) \leq n$ then $T_n = \epsilon$.

We then define traces as follows.

Definition 12. *T is a **trace** of transition system $(A, I, F, \dashrightarrow)$ iff*

1. *$len(T) > 0$*
2. *$T(0) \in I$*
3. *$\forall i.$ if $0 \leq i < (len(T) - 1)$ then $T(i) \dashrightarrow T(i+1)$*
4. *T finite implies $T(len(T) - 1) \in F$.*

We write $traces(A, I, F, \dashrightarrow)$ to represent the set of traces of the transition system $(A, I, F, \dashrightarrow)$.

2.3.2 Programs As Transition Systems

We will now discuss how to form the transition system corresponding to a program P. We first define $\underset{P}{\longrightarrow}$, the transition relation associated with program P.

Definition 13. *Given program P, let $\underset{P}{\longrightarrow}$ be the least relation satisfying the following.*

1. *If $\gamma_1 \rightsquigarrow \gamma_2$ then $\gamma_1 \underset{P}{\longrightarrow} \gamma_2$*
2. *$\mathbf{goto}(l, (s, h)) \underset{P}{\longrightarrow} \langle P(l), (s, h) \rangle$*

This definition states that the program transitions as long as either the current continuation can transition via the $\rightsquigarrow$ relation or a $\mathbf{goto}(l, (s, h))$ state has been reached, in which case execution proceeds from the continuation at l.

We can now define the interpretation of a program as a transition system. Recall that G is the set of all execution states.

Definition 14. *We write $(\!(P \mid Q_0)\!)$ to represent the transition system corresponding to program P with initial precondition Q_0. Let I and F be sets of states defined as follows.*

$$I = \left\{\mathbf{goto}(l_0, (s, h)) \;\middle|\; (l_0 = initloc(P)) \wedge (s, h) \models Q_0\right\}$$
$$F = \left\{\mathbf{final}(s, h) \;\middle|\; s \in \mathit{Stores} \wedge h \in \mathit{Heaps}\right\} \cup \left\{\mathbf{error}\right\}$$

Then $(\!(P \mid Q_0)\!) = (G, I, F, \underset{P}{\longrightarrow})$.

The semantics of a program P is then taken to be the set of traces produced by the transition system corresponding to P.

Definition 15. *The meaning of program P in initial state Q_0 is the set of traces given by traces$(\!(P \mid Q_0)\!)$.*

Note that infinite traces arise not from execution at the continuation level, as continuations always terminate, but rather from the execution of an infinite sequence of continuations, each of which reaches a goto l statement for some label l.

2.3.3 Transitive Closure of Relations

In addition to the relations $\xrightarrow[P]{}$ and $\leadsto$, we will also use their non-reflexive transitive closures, defined as follows.

Definition 16. *If R is a relation of type $A \times A \rightarrow$ Bool for some set A, then the **transitive closure** of R, written as R^+ is the least relation satisfying*

$$\forall a, b \in A.\ aR^+b \Leftrightarrow ((aRb) \vee (\exists c \in A.\ aRc \wedge cR^+b))$$

Thus, $\xrightarrow[P]{}^+$ indicates the transitive closure of the $\xrightarrow[P]{}$ relation, $\leadsto^+$ is the transitive closure of $\leadsto$, etc.

2.3.4 Deadlock and Angelic Non-determinism

We now consider how our semantics of branch statements interacts with the program semantics just presented. In particular, we consider what occurs in an execution state of the form

$$\langle \mathsf{branch}\ e_1 \Rightarrow k_1, \ldots, e_n \Rightarrow k_n\ \mathsf{end}, (s, h)\rangle$$

where $[\![e_i]\!]\, s = \mathsf{false}$ for all i. Such a state cannot make any transitions, thus it could only appear at the end of a finite trace. But this is not permitted, since Definition 12 states that the last state in a finite trace must be in F, the set of final states. Definition 14 specifies F

for our programs and this set does not contain any execution states of the form $\langle k, (s, h)\rangle$. Such a state might be described as *stuck* or *deadlocked*. An important property of our trace semantics is that traces are not allowed to contain deadlocked states.

We will further illustrate this with a concrete example. Consider the continuation below.

$$k \overset{\text{def}}{=} \big(\mathsf{branch}\ \mathsf{true} \Rightarrow (\mathsf{branch}\ e_1 \Rightarrow k_1\ \mathsf{end}), \mathsf{true} \Rightarrow (\mathsf{branch}\ e_2 \Rightarrow k_2\ \mathsf{end})\ \mathsf{end}\big)$$

Suppose T is a trace of a program containing k and that $T(i) = \langle k, (s, h)\rangle$. Then it must be the case that $[\![e_1]\!]\, s = \mathsf{true}$ or $[\![e_2]\!]\, s = \mathsf{true}$. Otherwise, execution would get stuck as neither $(\mathsf{branch}\ e_1 \Rightarrow k_1\ \mathsf{end})$ nor $(\mathsf{branch}\ e_2 \Rightarrow k_2\ \mathsf{end})$ would be able to transition from memory state (s, h). And as we just saw, such deadlocked states are not allowed to appear in traces. Furthermore, if $[\![e_2]\!]\, s = \mathsf{false}$ then $T(i+1) = \langle \mathsf{branch}\ e_1 \Rightarrow k_1\ \mathsf{end}, (s, h)\rangle$. That is, non-determinism is resolved such that only cases which do not later cause execution to deadlock are chosen. Such a situation is often described as *angelic non-determinism*. But why is this the appropriate treatment of non-determinism here?

One answer is that, in some sense, it does not matter how we choose to deal with stuck branches. The source language we actually consider—the C programming language—contains only *total* branches, which are branches where the disjunction of the branch conditions is equivalent to true. This ensures that, in the source program, execution can never get stuck at a branch point. For any branch, there is always a well-defined next state.

Our soundness theorem will then tell us that every trace of the original program corresponds to a trace of the numeric program. Thus, the fact that the numeric program throws away deadlocked traces does not hurt us, since soundness tells us that those traces were not necessary in order to obtain an over-approximation[1]. Once we have an over-approximation, this can be used to prove a variety of properties of the original program, as we will see in Chapter 3.

If it does not matter for soundness, then why then do we bother with this interpretation of branches? The reason is that the numeric programs we generate constitute an inter-

[1] For the purposes of this discussion, a program P' is an *over-approximation* of a program P iff the set of traces of P' contains the set of traces of P. More details are given in Chapter 3.

mediate language for communicating with an external verification tool (an intermediate language that corresponds to the input language of the tool). As such, it makes sense to leverage the full power of this language and include the constructs that have proved to be useful when verifying programs (and which are thus supported by most external verification tools).

One such construct is the "assume" statement, which lets us represent—in the code—properties that we know to be true at a given program point. For example, suppose that, from a verification standpoint, the only important property of a library routine $\mathsf{foo}(x)$ is that it always returns a non-negative number. Then we can represent this in the code by replacing the statement "$y = \mathsf{foo}(x)$" with "$y := ?;\mathsf{assume}(y \geq 0)$". The statement "$\mathsf{assume}(y \geq 0)$" indicates that we should only consider traces for which $y \geq 0$ is true at this point, and discard all other traces. Our branch statements, with the given semantics, are similar in that the continuation "$\mathsf{branch}\ e_1 \Rightarrow k_1, e_2 \Rightarrow k_2\ \mathsf{end}$" states that only traces where e_1 or e_2 are true need to be considered. If we have only one condition, as in the continuation "$\mathsf{branch}\ e \Rightarrow k\ \mathsf{end}$," then the semantics correspond exactly to our informal description of $\mathsf{assume}(e)$ and we will adopt the notation $\mathsf{assume}(e);k$ as an abbreviation for $\mathsf{branch}\ e \Rightarrow k\ \mathsf{end}$.

In summary, since verification generally views a program as representing a set of traces and attempts to over- or under-approximate those traces, having a command in the language for filtering trace sets is very useful. Our semantics for the "$\mathsf{branch}\ \ldots\ \mathsf{end}$" construct provides this. The difficulties that may be encountered if one attempts to actually implement such a command are not a concern, since the source programs we consider do not make use of the trace filtering aspect of these commands.

2.4 Representing C Programs

The C language syntax contains a number of ambiguities and corner cases as described in [Necula et al., 2002]. In our implementation, we use the framework described in that paper (CIL) to reduce C to a more regular subset of the language. We will not go into a large amount of detail on how CIL constructs can be translated into our language (the CIL syntax is rather involved), but we will address some of the high-level issues that arise when working with code originally written in the C language.

2.4.1 Control Flow

Figure 2.9 shows how various control-flow constructs can be interpreted. The constructs considered in that figure are all well-structured, in that they do not contain jumps out of loops or case statements that fall through. Such irregular flow-of-control can be dealt with by asking CIL to convert `break` and `continue` statements into explicit gotos.

2.4.2 Memory Operations

Memory operations in C are considerably more complex than those permitted by the language in Section 2.1. However, they can be reduced to the simpler memory model that we use for our logic and analysis by a number of conversions. In the following, we will use the terminology *record* to refer to a collection of values structured using named fields. In C, these same constructs are called *structures* or *structs*. C requires that structure definitions and types always be proceeded by the `struct` keyword.[2]

Nested Records The C language allows nested records, as below, where `(*out)` indicates the dereference of the memory cell at the address stored in `out`.

```
struct inner {
  int x;
  int y;
};

struct outer {
```

[2] There are ways around this syntactic inconvenience, but for clarity and consistency, we do not use such tricks in these examples.

```
  int x;
  struct inner in;
};

int main() {
  struct outer *out;
  out = malloc(sizeof(struct outer));
  (*out).in.x = 5;
  ...
}
```

Such records can be flattened to contain only a single level of fields. If there are naming conflicts, as there are in this example, then fields must be renamed to avoid clashes. Code equivalent to the above that uses only a single level of record structure is given below.

```
struct outer {
  int x;
  int in_x;
  int in_y;
};

int main() {
  struct outer *out;
  out = malloc(sizeof(struct outer));
  (*out).in_x = 5;
  ...
}
```

The code for `main` in our syntax then becomes

$$
\begin{array}{l}
\mathsf{out} := \mathsf{alloc}(\mathsf{x}^{\mathsf{i}}, \mathsf{in\_x}^{\mathsf{i}}, \mathsf{in\_y}^{\mathsf{i}})\texttt{;} \\
\mathsf{out.in\_x} := 5\texttt{;} \\
\mathsf{halt}
\end{array}
$$

if(e) {
c_1
} else
c_2
}
l_1: c_3
	$\Longrightarrow$	branch $e \Rightarrow \text{ctrans}(c_1)$; goto l_1,
		$\neg e \Rightarrow \text{ctrans}(c_2)$; goto l_1 end
		; $l_1 : \text{ctrans}(c_3)$

l_1: while(e) {
c_1
}
c_2
	$\Longrightarrow$	l_1 : branch $e \Rightarrow \text{ctrans}(c_1)$; goto l_1,
		$\neg e \Rightarrow \text{ctrans}(c_2)$ end

switch(e) {
case e_1: c_1; break;
case e_2: c_2; break;
⋮
case e_n: c_n; break;
}
l_1: c
	$\Longrightarrow$	branch $(e = e_1) \Rightarrow \text{ctrans}(c_1)$; goto l_1,
		$(e = e_2) \Rightarrow \text{ctrans}(c_2)$; goto l_1,
		⋮
		$(e = e_n) \Rightarrow \text{ctrans}(c_n)$; goto l_1 end
		; $l_1 : \text{ctrans}(c)$

Figure 2.9: Translations of C programs with regular control-flow into the syntax presented in Section 2.1. The function “ctrans()” represents a recursive application of these rules. We assume that fresh labels (l_i) are generated and inserted in the C program wherever necessary to apply these rules. Translations for atomic commands are not given, but are discussed in Section 2.4.2.

If the record is not heap-allocated, but instead allocated on the stack, as in the `main` procedure given below, then we can convert the record fields to stack variables. For example, consider the code below.

```
int main() {
  struct outer out;
  out.in_x = 5;
  ...
}
```

This becomes the following.

```
int main() {
  int out_x;
  int out_in_x;
  int out_in_y;

  out_in_x = 5;
  ...
}
```

Translated into our language, this corresponds to

$$\mathsf{out\_in\_x} := 5;\ \mathsf{halt}$$

Addresses of substructures The above tricks for nested records fail in the presence of the “address-of” operator. For example, C permits the following, which specifies a record within a record and then uses “address-of” (the “&” operator) to obtain a pointer to the inner record.

```
int get_x(struct inner *in) {
  return (*in).x;
}

int main() {
  struct outer out;
  ...
  int x = get_x(&out.in);
  ...
}
```

In such cases, to perform a faithful translation, we have to keep the record nesting explicit, using pointers to connect the inner and outer records. In general, any time a component of a record may have its address taken, we have to ensure that this component is allocated as a separate heap cell. Below, we give the translation of the code above, including updated versions of the structure definitions. Note that the inner structure is now explicitly allocated on the heap.

```
struct inner {
  int x;
  int y;
};

struct outer {
  int x;
  struct inner *in;
};

int get_x(struct inner *in) {
  return (*in).x;
}

int main() {
  struct outer out;
  out.in = malloc(sizeof(struct inner));
  ...
  int x = get_x(out.in);
  ...
}
```

This can then be translated to the following code in our system (where the call to `get_x` has been inlined).

$$\text{out\_in} := \mathsf{alloc}(x^{\mathrm{i}}, y^{\mathrm{i}})\texttt{;}$$

$$x := \text{out\_in.x}$$

Pass by reference The "address-of" operator is also used to get around the call-by-value nature of C language functions. In the following example, the function `add_front` uses double-indirection to update the list pointer that is passed in by the `main` function.

```
struct list {
  struct list *next;
  int data;
};

void add_front(struct list **lst, int v) {
  struct list *temp = malloc(sizeof(struct list));
  temp->data = v;
  temp->next = (*lst);
  *lst = temp;
}

int main() {
  struct list *p;
  p = 0;
  add_front(&p, 1);
  add_front(&p, 2);
  add_front(&p, 3);
  ...
}
```

For such cases, as with nested records whose address is taken, we have to insert code that lays out the structure in memory and change commands that access the structure in a way this is consistent with the semantics of the original code. The basic rule is the same as before: any piece of memory that may have its address taken must be allocated as a separate cell in the heap. The code below is the translation of the code above. Only the code in `main` needs to be changed.

```
int main() {
  struct list **p;
  p = malloc(sizeof(struct list *));
  *p = 0;
  add_front(p, 1);
  add_front(p, 2);
  add_front(p, 3);
  ...
}
```

In general, if we have a stack variable x of type t whose address is taken, we must change the type of x to "pointer to t." At the start of the scope containing x, we allocate a new heap cell and set x to the address of this cell. Commands that previously accessed x are changed to instead access $*(x)$ (the dereference of x) and commands that had the form &x (address of x) are changed to instead refer to x directly.

The reason these rewrites are required is that, in our memory model, all fields associated with a record are always referred to through a common address. Other models are possible, in which record components are given different, often related, addresses. For example, if addresses are taken to be natural numbers, record components can be laid out sequentially in memory. Such models are sometimes referred to as *field splitting models* (Berdine [2006]) and, while they enable easier treatment of record components whose address is taken, they make it harder to write a rule for C-style de-allocation (where calling `free(x)` causes the entire contiguous block starting at `x` to be freed).

2.4.3 Unhandled Features

There are a number of C language features that cannot be translated into the program representation presented in Section 2.1. Pointer arithmetic cannot be translated, as we have adopted a type system specifically aimed at eliminating that feature. Our language's integer variables also do not match up exactly with C's integers. Our integers are unbounded whereas in C there are several types of integer variable, each of which can store different, finite subsets of the integers. For example, "`unsigned long x`" declares `x` to be a variable that can store an unsigned 32-bit value (that is, a value in the range 0 to $2^{32} - 1$). Such types could be easily added to our system. In addition to the types a and i that we have already, we would simply have additional base types representing bounded integers for which mathematical operations are performed modulo the range.

Such additional types do not cause problems, and in fact are included in our implementation. However, since our focus is on the type a of addresses and the analysis of data structures built through pointer manipulations, we omit these types from the theory presented here. Note that even if we add integer types corresponding to C's bounded integers, we still must retain the unbounded integer type i. This is needed because the size measures associated with data structures are unbounded.

This distinction between bounded and unbounded integers must be kept in mind when choosing tools to apply to the numeric programs that our algorithm generates. Since our numeric programs involve unbounded integers, the tools we use to analyze them must support these. Otherwise, we can end up with cases where, for example, we repeatedly cons onto a list, increasing the length by one each time, but due to modular arithmetic the tool concludes that the list is eventually empty (length equal to zero).

Finally, we do not support arrays or unions. Verification of arrays has been extensively studied [Halbwachs and Péron, 2008, Bozga et al., 2009, Gopan et al., 2005] and most of these approaches could likely be incorporated into our analysis to provide some level of support for arrays. A straightforward combination, such as a direct product of domains [Cousot and Cousot, 1979] would allow for tracking of heap properties and tracking of array properties, but would not permit interaction between the two. However, in C there are many ways in which arrays and the heap can interact—perhaps more so than in other languages since C considers arrays to be pointers and allows them to appear in most contexts where a pointer would be expected. Tracking such interactions is an interesting avenue of future work, but is outside the scope of this thesis.

2.5 Generating C Programs

The end goal of our analysis is to convert a program in the language given in Figure 2.1 into another program that only manipulates integer-valued variables and which can be passed to a separate program analysis tool for further checking. The program we generate will also be in the language given in Figure 2.1 and so we must consider how we will represent this program in a format that standard verification tools can accept. Most of our commands have standard analogues in C and other imperative languages. The exceptions are non-deterministic assignment ($x := ?$) and our branch construct.

The input format for program analysis tools is generally either some specific programming language, such as C or Java, or some form of transition system. The details vary and we will not go into the specific translations required for each tool. Instead, we note that we can generally perform such translations provided that the input language for the tool supports two basic features: non-deterministic values and *assume* statements.

Non-deterministic Values Non-determinism is often used by analysis tools to abstract portions of the code. For example, functions can sometimes be soundly abstracted by assuming that their result is non-deterministically chosen. Suppose we are checking the C code below for memory safety.

```
a = foo();
if(a > 0) {
  int x = malloc(sizeof(int));
  *x = 0;
}
else {
  a = a - 1;
}
```

Memory safety of this piece of code does not depend on the value of `a`, nor does it depend on which branch is taken (both branches are memory safe from any starting state). If we know that `foo` does not access the heap, then assuming that `foo` returns a non-deterministically chosen value still results in sound reasoning about memory safety and allows us to avoid analyzing the body of `foo` (which may be quite large).

Because this is a common abstraction technique, verification tools often expose the ability to generate non-deterministically chosen values. For example, BLAST recognizes the special identifier `__BLAST_NONDET`, which always represents a fresh, non-deterministically-chosen value. Systems without a special non-deterministic value often interpret undefined functions non-deterministically. For example, in ARMC, the code `x = foo();` is equivalent to $x := ?$ in our language if the function `foo` is undefined.

Assume Statements Another common feature is support for *assume* statements. The semantics of the sequence of statements assume(e); c is defined such that control only passes to c if the expression e is true. Otherwise, execution blocks or silently halts. The effect of this, and the source for this statement's name, is that it allows a program analysis tool to add the assumption e to the current symbolic state before analyzing c.

These statements can be used to model functions more precisely than non-deterministic values alone allow us to. For example, if `foo` is known to return a positive value and not modify the global state, then the command `x := foo()` can be abstracted by the code `x = nondet; assume(x > 0);` where `nondet` represents a non-deterministically chosen value. Our semantics results in the non-determinism being resolved *angelically*—that is, a non-deterministic value is chosen which satisfies the following assume statement.

Often, verification tools accept a version of C that is augmented with an assume statement that has the semantics above. Even if *assume* is not present in the input language explicitly, the command

```
assume(e); c
```

can be modeled as

```
if(e)
  { c }
else
  { exit(0); }
```

where `exit(0)` causes normal (non-error) termination of the program.

Representing Branches These two features combine to let us faithfully encode our branch construct. If we have the code below

$$\begin{aligned} \textsf{branch}\ \ & e_1 \Rightarrow k_1 \\ & e_2 \Rightarrow k_2 \\ & \qquad \vdots \\ & e_n \Rightarrow k_n\ \textsf{end} \end{aligned}$$

then this can be encoded by the following sequence of conditionals, non-deterministic assignment, and assume statements. We write `c1` for the translation of k_1, `c2` for the translation of k_2, etc.

```
a = nondet;
if(a == 1)
  { assume(e1); c1; }
else if (a == 2)
  { assume(e2); c2; }
...
else if (a == n)
  { assume(en); cn; }
else
  { assume(false); }
```

This encoding ensures that all valid paths through the code will be explored. The variable `a` can take on any value, and so any sound analysis tool must explore each branch. In each case, the analysis is allowed to assume the condition for that case (e_1, e_2, etc.). The branch where none of the conditions are true is modeled with `assume(false)`, which indicates that there are no valid executions along this branch (and this is exactly the semantics of our branch construct in the case where all branch conditions are false).

Chapter 3

Abstractions and Program Properties

In Chapter 2 we gave the semantics of programs in terms of the traces produced by a transition system. In this chapter, we present the logic we will use for describing properties of these traces. A common language for describing properties of traces is *linear temporal logic (LTL)* [Clarke et al., 1999], and the logic we describe in the next section is based on this.

In addition to presenting the logic we use for stating program properties, we formally define a notion of program abstraction in this section. Roughly, a program P' is an abstraction of program P with respect to some property ϕ if whenever ϕ holds of P', it also holds of P.

When setting up a framework for program abstraction, it is common for a program and its abstraction to require different numbers of executions steps to arrive at the same result. To take a simple example, the command $\mathsf{x} := 1$ and the commands $\mathsf{skip;x} := 1$ both transition to a state in which x has the value 1, but the second sequence requires two steps to reach this state.

This motivates the use of a logic for program properties that is not sensitive to the number of steps taken and the logic we describe in this chapter has this property. We also present equivalence relations between traces that are insensitive to the number of steps taken and use this notion of equivalence to formally define a notion of program abstraction.

Finally, we conclude by highlighting four specific program properties that we have focused on in our experiments.

The techniques used in this chapter are tailored toward our semantic domain but are based on standard notions of stuttering equivalence, simulation and stuttering simulation [Milner, 1971, Browne et al., 1988].

3.1 LTSL

In this section we describe a temporal logic based on LTL$\backslash$X [Clarke et al., 1999], or "linear temporal logic without X (the next-time operator)." This logic supports the stating of program properties involving constraints on ordering, necessity, and properties of sequences of events, but does not permit specifications of exactly how many steps are involved in satisfying the property. The variant of LTL$\backslash$X presented here differs from standard LTL$\backslash$X in that the atomic propositions consist of separation logic formulae and the traces over which temporal formulae are interpreted can be finite. The resulting logic will be referred to as LTSL (for "linear temporal separation logic"). The syntax of the logic is given in Figure 3.1.

An atomic formula is either a separation logic formula Q, the formula *err*, which represents an error state, the formula *final*, which represents a non-error final state, or the formula $atloc(l)$, which indicates that the current execution state is associated with label l. An LTSL formula is then composed of these atomic formulae plus the temporal operators $\mathbf{G}, \mathbf{F}$, and $\mathbf{U}$ and the Boolean operators $\curlywedge, \curlyvee$ and $\sim$, corresponding to conjunction, disjunction, and negation, respectively. We use these symbols in order to distinguish the connectives at the level of path formulae from the connectives $\wedge, \vee$, and $\neg$ that were already defined for separation logic formulae. We define implication as $a \supset b$ if and only if ${\sim}a \curlyvee b$.

The semantics of the LTSL constructs is defined in Figure 3.2. Recall that T_n is the trace obtained by discarding the first n elements of trace T (resulting in the empty trace ϵ if T does not contain at least n elements). A separation logic formula holds at a state

$$\begin{array}{lrcl} \textit{State Formulae} & \varsigma & ::= & Q \mid \textit{err} \mid \textit{final} \mid \textit{atloc}(l) \\ \textit{Path Formulae} & \phi & ::= & \varsigma \mid \phi \wedge \phi \mid \phi \vee \phi \mid {\sim}\phi \mid \mathbf{G}\phi \mid \mathbf{F}\phi \mid \phi\,\mathbf{U}\,\phi \end{array}$$

Figure 3.1: Syntax of the logic LTSL.

if the store and heap at that state satisfy the formula. The *err* and *final* formulas hold of error and final states respectively. The $atloc(l)$ formula holds if a state is of the form $\mathbf{goto}(l, (s, h))$. The semantics of the path formulas involves reasoning about a sequence of states. The formula $\mathbf{G}\phi$ holds if ϕ holds globally—that is, it holds of every suffix of the sequence. The formula $\mathbf{F}\phi$ holds if ϕ holds of some suffix of the sequence. If we interpret the sequence as a series of points in time, then $\mathbf{G}\phi$ says that ϕ holds at *all* future points, whereas $\mathbf{F}\phi$ says that ϕ holds at *some* future point. Note that "future" here includes what might, in common usage, be referred to as the "present" (that is, it includes the first state in the trace). The formula $\phi_1\,\mathbf{U}\,\phi_2$ holds when ϕ_2 holds at some future point and ϕ_1 holds at every point up to (but not necessarily including) the point at which ϕ_2 holds.

An LTSL formula holds of a transition system S if and only if it holds of all traces of S. The relation $T \models_X \phi$ below is the one given in Figure 3.2.

Definition 17. *Let S be a transition system. Then $S \models_X \phi$ iff $\forall T \in traces(S).\ T \models_X \phi$.*

We say that an LTSL formula ϕ holds of a program P with initial states satisfying Q_0 iff $(\!(P \mid Q_0)\!) \models_X \phi$.

LTL$\backslash$X is generally interpreted over infinite paths. However, our execution traces can be finite and the semantics presented in Figure 3.2 provides for interpretation of LTSL formulae over finite paths. This interpretation of the LTSL operators over finite paths given here is consistent with the other common method of accommodating finite paths, which is to extend them to infinite paths by replicating the final state.

Note that, as in the semantics for separation logic formulae given in Figure 2.7, the satisfaction relation given here is parametric in the set of inductive predicates X. All the properties we discuss in this section will hold for any set X satisfying the conditions given

State Formulae

$$
\begin{array}{lll}
\gamma \models_X \mathit{err} & \text{iff} & \gamma = \mathbf{error} \\
\gamma \models_X \mathit{final} & \text{iff} & \gamma = \mathbf{final}(s,h) \text{ for some } s,\ h \\
\gamma \models_X \mathit{atloc}(l) & \text{iff} & \gamma = \mathbf{goto}(l,(s,h)) \text{ for some } s,\ h \\
\gamma \models_X Q & \text{iff} & \text{there exists } s,h \text{ such that } (s,h) \models_X Q \text{ and} \\
 & & \quad (\gamma = \langle k,(s,h)\rangle \text{ for some } k, \text{ or } \gamma = \mathbf{final}(s,h), \text{ or} \\
 & & \qquad \gamma = \mathbf{goto}(l,(s,h)) \text{ for some } l)
\end{array}
$$

Path Formulae

$$
\begin{array}{lll}
T \models_X \varsigma & \text{iff} & \mathit{len}(T) > 0 \text{ and } T(0) \models_X \varsigma \\
T \models_X {\sim}\phi & \text{iff} & T \not\models_X \phi \\
T \models_X \phi_1 \mathbin{\dot\vee} \phi_2 & \text{iff} & T \models_X \phi_1 \ \text{ or } \ T \models_X \phi_2 \\
T \models_X \phi_1 \mathbin{\dot\wedge} \phi_2 & \text{iff} & T \models_X \phi_1 \ \text{ and } \ T \models_X \phi_2 \\
T \models_X \mathbf{G}\phi & \text{iff} & \forall i.\ 0 \leq i < \mathit{len}(T) \text{ implies } T_i \models_X \phi \\
T \models_X \mathbf{F}\phi & \text{iff} & \exists i.\ 0 \leq i < \mathit{len}(T) \text{ and } T_i \models_X \phi \\
T \models_X \phi_1 \,\mathbf{U}\, \phi_2 & \text{iff} & \exists i.\ 0 \leq i < \mathit{len}(T) \text{ and } T_i \models_X \phi_2 \\
 & & \ \text{and } (\forall j.\ 0 \leq j < i \text{ implies } T_j \models_X \phi_1)
\end{array}
$$

Figure 3.2: Semantics of LTSL formulae. The notation T_i denotes the suffix of T starting at position i (where the first element has position 0). The satisfaction relation for Q is in Figure 2.7. We write $T \not\models_X \phi$ to indicate that the relation $T \models_X \phi$ does not hold.

in Section 2.2.2. Thus, all theorems given in this section should be considered universally quantified over X, unless otherwise specified.

3.1.1 Notation

To facilitate the compact representation of execution states, we will sometimes label control points in continuations with numbers enclosed in circles. We then use each number to refer to the continuation starting at that point in the term. For example, the continuation below contains four numbered control points.

$$\begin{aligned}
&\textcircled{1}\,\mathsf{branch\ x} = 0 \Rightarrow \textcircled{2}\,\mathsf{x} := \mathsf{x} + 1\texttt{;}\ \mathsf{halt},\\
&\qquad\qquad\mathsf{x} > 0 \Rightarrow \textcircled{3}\,\mathsf{x} := \mathsf{x} - 1\texttt{;}\ \textcircled{4}\,\mathsf{halt\ end}
\end{aligned}$$

The numbers then represent the following continuations:

$$\begin{aligned}
\textcircled{1} \equiv\ & \mathsf{branch\ x} = 0 \Rightarrow \mathsf{x} := \mathsf{x} + 1\texttt{;}\ \mathsf{halt},\\
&\qquad\qquad \mathsf{x} > 0 \Rightarrow \mathsf{x} := \mathsf{x} - 1\texttt{;}\ \mathsf{halt\ end}
\end{aligned}\tag{3.1}$$

$$\textcircled{2} \equiv \mathsf{x} := \mathsf{x} + 1\texttt{;}\ \mathsf{halt}\tag{3.2}$$

$$\textcircled{3} \equiv \mathsf{x} := \mathsf{x} - 1\texttt{;}\ \mathsf{halt}\tag{3.3}$$

$$\textcircled{4} \equiv \mathsf{halt}$$

3.1.2 Examples

Consider the following program.

$$\begin{aligned}
&P_1 \overset{\mathrm{def}}{=}\\
&\qquad \mathsf{L}_0 : \textcircled{1}\,\mathsf{x} := 0\texttt{;}\ \textcircled{2}\,\mathsf{goto}\ \mathsf{L}_1;\\
&\qquad \mathsf{L}_1 : \textcircled{3}\,\mathsf{branch\ x} < 2 \Rightarrow \textcircled{4}\,\mathsf{x} := \mathsf{x} + 1\texttt{;}\ \textcircled{5}\,\mathsf{goto}\ \mathsf{L}_1,\\
&\qquad\qquad\qquad\quad \mathsf{x} \geq 2 \Rightarrow \textcircled{6}\,\mathsf{x} := 0\texttt{;}\ \textcircled{7}\,\mathsf{goto}\ \mathsf{L}_1\\
&\qquad\qquad\qquad\qquad \mathsf{end}
\end{aligned}$$

Below is an example trace through this system. We only show the value of variable x since this is the only variable that appears in the program. We start this example trace in a state where x has the value 12. Similar traces would exist for all initial values of x.

$$\begin{gathered}
\mathbf{goto}(\mathsf{L}_0, (\{(\mathsf{x}, 12)\},\ \{\})) \\
\langle \textcircled{1}, (\{(\mathsf{x}, 12)\},\ \{\})\rangle \\
\langle \textcircled{2}, (\{(\mathsf{x}, 0)\},\ \{\})\rangle \\
\mathbf{goto}(\mathsf{L}_1, (\{(\mathsf{x}, 0)\},\ \{\})) \\
\langle \textcircled{3}, (\{(\mathsf{x}, 0)\},\ \{\})\rangle \\
\langle \textcircled{4}, (\{(\mathsf{x}, 0)\},\ \{\})\rangle \\
\langle \textcircled{5}, (\{(\mathsf{x}, 1)\},\ \{\})\rangle \\
\mathbf{goto}(\mathsf{L}_1, (\{(\mathsf{x}, 1)\},\ \{\})) \\
\langle \textcircled{3}, (\{(\mathsf{x}, 1)\},\ \{\})\rangle \\
\langle \textcircled{4}, (\{(\mathsf{x}, 1)\},\ \{\})\rangle \\
\langle \textcircled{5}, (\{(\mathsf{x}, 2)\},\ \{\})\rangle \\
\mathbf{goto}(\mathsf{L}_1, (\{(\mathsf{x}, 2)\},\ \{\})) \\
\langle \textcircled{3}, (\{(\mathsf{x}, 2)\},\ \{\})\rangle \\
\langle \textcircled{6}, (\{(\mathsf{x}, 2)\},\ \{\})\rangle \\
\langle \textcircled{7}, (\{(\mathsf{x}, 0)\},\ \{\})\rangle \\
\mathbf{goto}(\mathsf{L}_1, (\{(\mathsf{x}, 0)\},\ \{\})) \\
\vdots
\end{gathered}$$

We will now state some properties satisfied by this trace. First, it does not terminate. This corresponds to the LTSL formula $\sim\!(\mathbf{F}(\mathit{final} \vee \mathit{err}))$. It also visits location L_1 infinitely often. This corresponds to the formula $\mathbf{G}(\mathbf{F}(\mathit{atloc}(\mathsf{L}_1)))$. Note that the formula $\mathbf{G}(\mathbf{F}(\varsigma))$ does not, in general, guarantee that ς holds infinitely often. It can also be satisfied by finite traces ending in a state satisfying ς. This means that our example formula $\mathbf{G}(\mathbf{F}(\mathit{atloc}(\mathsf{L}_1)))$ would also be satisfied by any finite trace ending in a state of the form $\mathbf{goto}(\mathsf{L}_1, (s, h))$. However, such traces are ruled out by the semantics of programs given in Definition 14. Since the state $\mathbf{goto}(l, (s, h))$ can always make a transition, it is not allowed to be the final state in a trace.

Finally, at label L_1 in the example program, x is always less than or equal to 2, which corresponds to the formula $\mathbf{G}(\mathit{atloc}(\mathsf{L}_1) \mathbin{\dot{\supset}} \mathsf{x} \leq 2)$. All of these properties are satisfied by all traces of the program and thus hold of the transition system $(\!(P_1 \mid \mathsf{true})\!)$.

As a second example, consider the program below.

```
P2 ≝
        L0 : x := nil; a := 0; goto L1;
        L1 : branch true ⇒ t := alloc(next); t.next := x;
                              x := t; a := a + 1; goto L1,
                    true ⇒ halt
             end
```

This program satisfies the property $\mathbf{G}(\mathit{atloc}(\mathsf{L}_1) \mathbin{\dot{\supset}} \mathit{ls}(\mathsf{a}, \mathsf{x}, \mathsf{nil}))$, where $\mathit{ls}(\mathsf{a}, \mathsf{x}, \mathsf{nil})$ is the predicate defined below, which states that there is a list of length a starting at memory address x.

$$\begin{aligned}
&\mathit{ls}(n, \mathit{start}, \mathit{end}) \equiv \\
&\qquad (\mathbf{emp} \wedge \mathit{start} = \mathit{end} \wedge n = 0) \\
&\quad \vee (n > 0 \wedge (\exists z.\ (\mathit{start} \mapsto [\mathsf{next} : z]) * \mathit{ls}(n-1, z, \mathit{end})))
\end{aligned}$$

It is also the case that every trace either visits location L_1 infinitely often, or the trace terminates in a state $\mathbf{final}(s, h)$. This corresponds to the property $\mathbf{F}(\mathit{final}) \mathbin{\dot{\vee}} \mathbf{G}(\mathbf{F}(\mathit{atloc}(\mathsf{L}_1)))$.

3.1.3 Core Connectives

Not all the connectives defined in Figure 3.2 need to be considered primitive. Many can be defined in terms of other connectives. The following list of connectives is sufficient to define the others.

$$\dot{\wedge} \quad \sim \quad \mathbf{U}$$

The following theorem shows how to define the other connectives in terms of these. In the following, we write $\phi \Leftrightarrow \phi'$ as shorthand for $\forall T.\ (T \models_X \phi)$ iff $(T \models_X \phi')$.

Theorem 9.

$$\phi_1 \mathbin{\dot\vee} \phi_2 \Leftrightarrow {\sim}({\sim}\phi_1 \mathbin{\dot\wedge} {\sim}\phi_2) \tag{3.4}$$

$$\mathbf{F}\phi \Leftrightarrow \mathsf{true}\,\mathbf{U}\;\phi \tag{3.5}$$

$$\mathbf{G}\phi \Leftrightarrow {\sim}(\mathbf{F}({\sim}\phi)) \tag{3.6}$$

Proof. **Equivalence 1:** $\phi_1 \mathbin{\dot\vee} \phi_2 \Leftrightarrow {\sim}({\sim}\phi_1 \mathbin{\dot\wedge} {\sim}\phi_2)$

Suppose we have a trace T and $T \models_X \phi_1 \mathbin{\dot\vee} \phi_2$. Then either $T \models_X \phi_1$ or $T \models_X \phi_2$. Without loss of generality, suppose it is $T \models_X \phi_1$ that holds. Then $T \models_X {\sim}\phi_1$ does not hold and thus $T \models_X ({\sim}\phi_1) \mathbin{\dot\wedge} ({\sim}\phi_2)$ does not hold. But this means that $T \models_X {\sim}\big(({\sim}\phi_1) \mathbin{\dot\wedge} ({\sim}\phi_2)\big)$ does hold, thus establishing the forward direction of the equivalence.

For the backward direction, assume that ${\sim}({\sim}\phi_1 \mathbin{\dot\wedge} {\sim}\phi_2)$ holds of T. Then $({\sim}\phi_1 \mathbin{\dot\wedge} {\sim}\phi_2)$ does not hold of T. This implies that either ${\sim}\phi_1$ or ${\sim}\phi_2$ does not hold. Without loss of generality, assume it is ${\sim}\phi_1$ that does not hold. Then ϕ_1 does hold, which implies that $\phi_1 \mathbin{\dot\vee} \phi_2$ does hold of T.

Equivalence 2: $\mathbf{F}\phi \Leftrightarrow \mathsf{true}\,\mathbf{U}\;\phi$

Suppose $T \models_X \mathbf{F}\phi$ for an arbitrary T. Then by the semantics in Figure 3.2 we have that there is an i satisfying

$$0 \leq i < \mathit{len}(T) \text{ and } T_i \models_X \phi$$

We must show the following

$$\exists i'.\, 0 \leq i' < \mathit{len}(T) \text{ and } T_i' \models_X \phi \text{ and } \forall j.\, 0 \leq j < i' \text{ implies } T_j \models_X \mathsf{true}$$

We let i' be i. Our assumption on i tells us that the formula $0 \leq i' < \mathit{len}(T)$ is satisfied, as is $T_i \models_X \phi$. All that remains is to show

$$\forall j.\, 0 \leq j < i \text{ implies } T_j \models_X \mathsf{true}$$

Since $j < i'$ and $i' < \mathit{len}(T)$ we have that $j \leq \mathit{len}(T) - 2$ and thus the trace T_j contains at least two states. This implies that $T_j(0)$ cannot be the final state in the trace T_j. This fact ensures that $T_j(0)$ has either the form $\langle k, (s, h)\rangle$ or $\mathbf{goto}(l, (s, h))$. In either case, we have $(T_j(0) \models_X \texttt{true})$ and thus $(T_j \models_X \texttt{true})$. Since j was arbitrary, we have this for all j.

For the reverse direction, suppose that $(T \models_X \texttt{true}\ \mathbf{U}\ \phi)$ holds. Then we have

$$\exists i.\ 0 \leq i < \mathit{len}(T) \text{ and } T_i \models_X \phi \text{ and } \forall j.\ 0 \leq j < i \text{ implies } T_j \models_X \texttt{true}$$

But this implies

$$\exists i.\ 0 \leq i < \mathit{len}(T) \text{ and } T_i \models_X \phi$$

(we have simply dropped the last conjunct). This is the semantics of $\mathbf{F}\phi$.

Equivalence 3: $\mathbf{G}\phi \Leftrightarrow {\sim}(\mathbf{F}({\sim}\phi))$

Suppose we have $\mathbf{G}\phi$. Then by the semantics of LTSL (Figure 3.2) we have

$$\forall i.\ 0 \leq i < \mathit{len}(T) \text{ implies } T_i \models_X \phi \tag{3.7}$$

We must show that $\mathbf{F}({\sim}\phi)$ does not hold. The proof is by contradiction. Suppose $\mathbf{F}({\sim}\phi)$ did hold. Then there would exist a j with $0 \leq j < \mathit{len}(T)$ such that $T_j \models_X {\sim}\phi$. This implies that $T_j \models_X \phi$ does not hold. But by (3.7) we have that $T_j \models_X \phi$ does hold, leading to a contradiction.

For the backward direction, suppose that ${\sim}(\mathbf{F}({\sim}\phi))$ holds. Then we have that the following does not hold

$$\exists i.\ 0 \leq i < \mathit{len}(T) \text{ and } T_i \models_X {\sim}\phi$$

This is equivalent to saying that the following formula *does* hold

$$\forall i.\ \neg(0 \leq i < \mathit{len}(T)) \text{ or } T_i \not\models_X {\sim}\phi$$

Expanding the semantics of ${\sim}$, this is equivalent to

$$\forall i.\ \neg(0 \leq i < \mathit{len}(T)) \text{ or } T_i \models_X \phi$$

If we now pick an arbitrary j and suppose that $0 \leq j < \mathit{len}(T)$, then the assumption above tells us that $T_j \models_X \phi$ must hold. Thus we have

$$\forall j.\ 0 \leq j < \mathit{len}(T) \text{ implies } T_j \models_X \phi$$

which is the definition of $T \models_X \mathbf{G}\phi$. □

3.2 Stuttering Equivalence

We consider traces equivalent up to repeated states or *stuttering*. We use a definition of stuttering based on that in [Manolios, 2001] and [Martí-Oliet et al., 2008]. To formally define stuttering, we first define what it means for traces to *match* according to an equivalence relation E.

Definition 18. *If T and T' are traces, we write $matches(T, T', \alpha, \beta, E)$ iff E is an equivalence relation on states and α and β are strictly increasing functions $\alpha, \beta : \mathbb{N} \to \mathbb{N}$ with $\alpha(0) = \beta(0) = 0$ such that, for all $i, j, k \in \mathbb{N}$,*

$$\alpha(i) \leq j < \alpha(i+1) \textit{ and } \beta(i) \leq k < \beta(i+1)$$
$$\textit{implies}$$
$$\big(j < \mathit{len}(T) \Leftrightarrow k < \mathit{len}(T')\big) \textit{ and } \big(j < \mathit{len}(T) \Rightarrow (T(j))\ E\ (T'(k))\big)$$

The functions α and β partition the traces into matching segments. The condition that $(j < \mathit{len}(T)) \Leftrightarrow (k < \mathit{len}(T'))$ ensures that, if the traces are both finite, then the final segment of T matches the final segment of T'. It also ensures that if the final segment of T ends at $\alpha(i)$ then $\alpha(i) = \mathit{len}(T)$ and $\beta(i) = \mathit{len}(T')$. In essence, this states that there is no segment that "straddles" the end of either trace.

We can now define stuttering equivalence of traces with respect to an equivalence relation E.

Definition 19. *Two traces T and T' are E-**stuttering equivalent**, written $T \sim_E T'$, iff $\exists \alpha, \beta.\ matches(T, T', \alpha, \beta, E)$.*

If two traces match, there is always a canonical α, β that witness this. The canonical matching function for trace T, written B_T, is defined below.

Definition 20. *Given a trace T, let B_T be the strictly increasing function of type $\mathbb{N} \to \mathbb{N}$ defined as follows.*

$$B_T(0) = 0$$

$$B_T(i+1) = \begin{cases} \textit{the least j such that } j > B_T(i) \wedge \neg\big((T(j))\ E\ (T(B_T(i)))\big) \\ \qquad\qquad\qquad\qquad \textit{if such a } j \textit{ exists} \\ len(T) \qquad\qquad\quad \textit{if no such } j \textit{ exists and } B_T(i) < len(T) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{and } T \textit{ is finite} \\ B_T(i) + 1 \qquad\quad\ \textit{otherwise} \end{cases}$$

The function B_T divides T into blocks such that all elements within the same block are related by E and these blocks have maximum size. If T is finite, the last of these blocks ends at $len(T)$. If T is infinite, either the first case of the definition will apply infinitely often, or we will eventually reach some tail consisting of elements that are all E-related. If this happens, then the third case of the definition applies and B_T begins counting up by one at each step. Note that B_T is clearly strictly increasing. For each case of the inductive definition, we have that $B_T(i+1) > B_T(i)$.

The following theorem then states that if a match exists, the matching functions can be replaced with the canonical matching functions for the two traces.

Theorem 10. *If $matches(T, T', \alpha, \beta, E)$ then $matches(T, T', B_T, B_{T'}, E)$.*

Proof. We have $B_T(0) = 0$ and $B_{T'}(0) = 0$ from the definition of B. This is one condition for $matches(T, T', B_T, B_{T'}, E)$. To complete the proof, we must show that the following holds for an arbitrary i, j, k.

$$B_T(i) \leq j < B_T(i+1) \text{ and } B_{T'}(i) \leq k < B_{T'}(i+1)$$
$$\text{implies}$$
$$\big(j < len(T) \Leftrightarrow k < len(T')\big) \text{ and } \big(j < len(T) \Rightarrow (T(j))\ E\ (T'(k))\big)$$

Let i, j, k be as above. We then case split on the case of Definition 20 that was used to define $B_T(i+1)$.

CASE 1 [First or second case of Definition 20 was used for $B_T(i+1)$] In this case, we can establish the following, which states that if a block of B_T ends at some index, then there is also a block of α that ends at that index, and similarly for $B_{T'}$ and β. Furthermore, if it is the r^{th} block of α that coincides with B_T, then it is also the r^{th} block of β that coincides with $B_{T'}$.

$$\forall q \in \mathbb{N}.\ q \leq i+1 \Rightarrow \exists r \in \mathbb{N}.\ B_T(q) = \alpha(r) \wedge B_{T'}(q) = \beta(r) \tag{3.8}$$

Proof. We show this by induction on q. The 0 case is straightforward. We let $r = 0$. Since $B_T(0), B_{T'}(0), \alpha(0)$, and $\beta(0)$ are all equal to 0, we have the equalities in the conclusion immediately.

For the inductive case, we assume that there exists some r such that $B_T(q) = \alpha(r)$ and $B_{T'}(q) = \beta(r)$ and we show there exists some s such that $B_T(q+1) = \alpha(s)$ and $B_{T'}(q+1) = \beta(s)$ provided $q+1 \leq i+1$.

Showing $B_T(q+1) = \alpha(s)$ We have $q+1 \leq i+1$, which implies $q \leq i$. Since $B_T(q+1)$ was defined by either the first or second case of Definition 20, we also have that either there is some next block of elements not related by E to those at $B_T(q)$ or $B_T(q)$ marks the start of the last block of E-related elements in a finite trace. Since α is strictly increasing, there is some s such that $\alpha(s) \leq B_T(q+1) < \alpha(s+1)$. If $\alpha(s) = B_T(q+1)$ then we have shown the first conjunct of our goal. We will show that in the other case we obtain a contradiction. Suppose $\alpha(s) < B_T(q+1)$. Then we have

$$\alpha(s) \leq B_T(q+1) - 1 < B_T(q+1) < \alpha(s+1) \tag{3.9}$$

and thus, because we have $matches(T, T', \alpha, \beta, E)$, we know that the following holds.

$$T(B_T(q+1) - 1)\ E\ (T(B_T(q+1)))$$

This contradicts the maximality of block q of B_T if $B_T(q+1)$ is the index of the next block that is not E-related to $T(B_T(q))$ (that is, if $B_T(q+1)$ is defined via the first case in Definition 20). If $B_T(q+1) = len(T)$ (that is, if $B_T(q+1)$ was defined via the second case in Definition 20), then we case split on whether $\alpha(s) = \mathit{len}(T)$. If it does, then we are done, as $B_T(q+1) = \mathit{len}(T)$ and thus $B_T(q+1) = \alpha(s)$. If it does not, then we again have (3.9). Because $\mathit{matches}(T, T', \alpha, \beta, E)$ holds, this implies $B_T(q+1) - 1 < \mathit{len}(T)$ if and only if $B_T(q+1) < \mathit{len}(T)$. But this cannot be since $B_T(q+1) = \mathit{len}(T)$.

Showing $B_{T'}(q+1) = \beta(s)$ To show that $\beta(s) = B_{T'}(q+1)$, we note that we have $\alpha(r) = B_T(q)$ and $\alpha(s) = B_T(q+1)$. This implies that there are $s - r$ blocks of α which correspond to the single block of B_T from q to $q+1$. Because we have $\mathit{matches}(T, T', \alpha, \beta, E)$, each of these blocks of α must match the corresponding block of β. This implies $\forall x.\ \beta(r) \leq x < \beta(s) \Rightarrow T'(\beta(x))\ E\ T'(\beta(r))$. To show that $B_{T'}(q+1) = \beta(s)$, we must show that this segment from $\beta(r)$ to $\beta(s)$ constitutes a maximal block of E-related elements in T'. We already have that the elements are E-related. To see that it is maximal, first note that one of the first two cases of Definition 20 were used to define B_T. From this, we have that either $\alpha(s) = \mathit{len}(T)$ or $\neg(T(\alpha(r))\ E\ T(\alpha(s)))$. Due to $\mathit{matches}(T, T', \alpha, \beta, E)$, this implies that either $\beta(s) = \mathit{len}(T')$ or $\neg(T'(\beta(r))\ E\ T'(\beta(s)))$. In either case, we have a maximal block of E-related elements in T' and so the definition of $B_{T'}$ ensures $B_{T'}(q+1) = \beta(s)$. □

We now return to the proof of the following.

$$B_T(i) \leq j < B_T(i+1) \text{ and } B_{T'}(i) \leq k < B_{T'}(i+1)$$
$$\text{implies}$$
$$\big(j < \mathit{len}(T) \Leftrightarrow k < \mathit{len}(T')\big) \text{ and } \big(j < \mathit{len}(T) \Rightarrow (T(j))\ E\ (T'(k))\big)$$

We first show the requirement that elements in the same block be E-related (the second conjunct in the consequent). Suppose $B_T(i) \leq j < B_T(i{+}1)$ and $B_{T'}(i) \leq k < B_{T'}(i{+}1)$. We have from (3.8) that there exists some r such that $B_T(i) = \alpha(r)$ and $B_{T'}(i) = \beta(r)$. From $\mathit{matches}(T, T', \alpha, \beta, E)$ we then have $T(\alpha(r))\ E\ T'(\beta(r))$ and thus we have $T(B_T(i))\ E\ T'(B_{T'}(i))$. Since $B_T(i+1)$ is the first index s such that $s > B_T(i)$ and either $\neg(T(B_T(i))\ E\ T(s))$ or $j = \mathit{len}(T)$, we have that $T(B_T(i))\ E\ T(j)$ for all j such that $B_T(i) \leq j < B_T(i{+}1)$. Similarly, since $B_{T'}(i{+}1)$ is either $\mathit{len}(T')$ or the index of the first element after $B_{T'}(i)$ in T' that is not E-related to $B_{T'}(i)$, we have $T'(B_{T'}(i))\ E\ T'(k)$ for all k satisfying $B_{T'}(i) \leq k < B_{T'}(i+1)$. Since E is an equivalence relation and $T(B_T(i))\ E\ T'(B_{T'}(i))$, this gives us $T(j)\ E\ T(k)$ as desired.

For the length requirement, we have that either the first or second case of the definition of $B_T(i+1)$ applies, implying that either $B_T(i+1) < \mathit{len}(T)$ or $B_T(i+1) = \mathit{len}(T)$. In either case, for any j with $B_T(i) \leq j < B_T(i+1)$ we have $j < \mathit{len}(T)$. It remains to show that for k satisfying $B_{T'}(i) \leq k < B_{T'}(i+1)$ we have $k < \mathit{len}(T')$. From (3.8) we have that there is some r such that $B_T(i+1) = \alpha(r)$ and $B_{T'}(i+1) = \beta(r)$. This, together with $\mathit{matches}(T, T', \alpha, \beta, E)$ and $\alpha(r) \leq \mathit{len}(T)$ implies that $\beta(r) \leq \mathit{len}(T)$ and thus $B_{T'}(i+1) \leq \mathit{len}(T')$, which implies $k < \mathit{len}(T')$ as required.

CASE 2 [Third case of Definition 20 was used for $B_T(i+1)$] In this case, we have that $B_T(i)$ is some point along an infinite tail of T where all elements are E-related. Let i' be the first element in this tail, which is necessarily less than or equal to $B_T(i)$. Either $i' = 0$ or there is some block of E-related elements prior to this infinite tail. We consider each case separately.

CASE $i' = 0$: In this case, T consists entirely of an infinite sequence of elements that are E-related. Since we have $\mathit{matches}(T, T', \alpha, \beta, E)$, this implies that T' is an infinite sequence of elements such that for all x, x' we have $T(x)\ E\ T'(x')$. Given such a situation, it trivially follows that for our j and k we have $T(j)\ E\ T'(k)$.

CASE $i' > 0$: In this case, there is some block of T prior to the infinite tail of E-related elements. Let $B_T(x)$ mark the start of this block. Since $i' > B_T(x)$ and $\neg(T(B_T(x))\ E\ T(i'))$, we have that the first case of Definition 20 must have been used when defining $B_T(x)$. Thus, **CASE 1** applies to $B_T(x)$, as does (3.8). That is, we have the following.

$$\forall q \in \mathbb{N}.\ q \leq x+1 \Rightarrow \exists r \in \mathbb{N}.\ B_T(q) = \alpha(r) \wedge B_{T'}(q) = \beta(r)$$

This implies that there is some r such that $B_T(x+1) = \alpha(r)$ and $B_{T'}(x+1) = \beta(r)$. This plus $\mathit{matches}(T, T', \alpha, \beta, E)$ implies that $T(B_T(x+1))\ E\ T'(B_{T'}(x+1))$. Since $B_T(x)$ marks the start of the block just before the infinite tail, $B_T(x+1)$ marks the start of the infinite tail (and so we have $i' = B_T(x+1)$). Since $B_T(x+1) = \alpha(r)$ and

$matches(T, T', \alpha, \beta, E)$, it must be the case that $\beta(r)$, which is equal to $B_{T'}(x+1)$, marks the start of an infinite tail of E-related elements in T'. From $T(B_T(x{+}1))\ E\ T'(B_{T'}(x{+}1))$, it follows that for all $y \geq B_T(x+1)$ and for all $z \geq B_{T'}(x+1)$, we have $T(y)\ E\ T'(z)$. Thus, we will have our result (that $T(j)\ E\ T(k)$) if we can show that $j \geq B_T(x+1)$ and $k \geq B_{T'}(x+1)$.

Since $i' = B_T(x+1)$, and we have $i' \leq i$, we have $B_T(x+1) \leq B_T(i)$. Since B_T is strictly increasing, this implies $x+1 \leq i$. Since $B_{T'}$ is strictly increasing we then have $B_{T'}(x+1) \leq B_{T'}(i)$. Since $j \geq B_T(i)$ and $k \geq B_{T'}(i)$ we then have our result.

For the length requirement, we have in both cases that T is infinite and thus, because of $matches(T, T', \alpha, \beta, E)$, T' is also infinite. So the $j \leq len(T) \Leftrightarrow k \leq len(T)$ conjunct of our goal holds trivially since $len(T) = len(T') = \omega$. □

The relation $\sim_E$ is symmetric, reflexive, and transitive. These properties result from the following properties of $matches$.

Lemma 7. *The following three statements hold of the* $matches$ *relation.*

$$matches(T, T', \alpha, \beta, E) \Rightarrow matches(T', T, \beta, \alpha, E)$$

$$matches(T, T, \lambda x.\ x, \lambda x.\ x, E)$$

$$matches(T, T', \alpha, \alpha', E) \wedge matches(T', T'', \alpha', \alpha'', E) \Rightarrow matches(T, T'', \alpha, \alpha'', E)$$

Proof. Recall that E is an equivalence relation. The first property, symmetry, follows from the fact that the definition of $matches$ is symmetric in T, α and T', β. The second property, reflexivity, is proved as follows. Both α and β are the identity relation, so T is partitioned by α (resp. β) into blocks consisting of a single element. Thus, we must establish that for any $i \in \mathbb{N}$ we have $i < len(T) \Leftrightarrow i < len(T)$ and $i < len(T) \Rightarrow (T(i))\ E\ (T(i))$. The first property is a tautology and the second follows from the fact that E is an equivalence relation and thus is reflexive.

For the third property, transitivity, we have $\alpha(0) = \alpha'(0) = 0$ and $\alpha'(0) = \alpha''(0) = 0$, thus $\alpha(0) = \alpha''(0) = 0$. This is the first part of the definition of $matches$. For the second part, we have the following

$$\forall i, j, k.\ \Big(\alpha(i) \leq j < \alpha(i+1)\Big) \wedge \Big(\alpha'(i) \leq k < \alpha'(i+1)\Big) \Rightarrow$$
$$\Big(j < \mathit{len}(T) \Leftrightarrow k < \mathit{len}(T')\Big) \wedge \Big(j < \mathit{len}(T) \Rightarrow \big(T(j)\big)\ E\ \big(T'(k)\big)\Big)$$
$$\forall i, j, k.\ \Big(\alpha'(i) \leq j < \alpha'(i+1)\Big) \wedge \Big(\alpha''(i) \leq k < \alpha''(i+1)\Big) \Rightarrow$$
$$\Big(j < \mathit{len}(T') \Leftrightarrow k < \mathit{len}(T'')\Big) \wedge \Big(j < \mathit{len}(T) \Rightarrow \big(T'(j)\big)\ E\ \big(T''(k)\big)\Big)$$

and we must show the following

$$\forall i, j, k.\ \Big(\alpha(i) \leq j < \alpha(i+1)\Big) \wedge \Big(\alpha''(i) \leq k < \alpha''(i+1)\Big) \Rightarrow$$
$$\Big(j < \mathit{len}(T) \Leftrightarrow k < \mathit{len}(T'')\Big) \wedge \Big(j < \mathit{len}(T) \Rightarrow \big(T(j)\big)\ E\ \big(T''(k)\big)\Big)$$

The following derivation establishes this.

1 $\forall i, j, k.\ \alpha(i) \leq j < \alpha(i+1) \wedge \alpha'(i) \leq k < \alpha'(i+1) \Rightarrow$
$((j < \mathit{len}(T)) \Leftrightarrow (k < \mathit{len}(T'))) \wedge (j < \mathit{len}(T) \Rightarrow T(j)\ E\ T'(k))$ (Given)

2 $\forall i, j, k.\ \alpha'(i) \leq j < \alpha'(i+1) \wedge \alpha''(i) \leq k < \alpha''(i+1) \Rightarrow$
$((j < \mathit{len}(T')) \Leftrightarrow (k < \mathit{len}(T''))) \wedge (j < \mathit{len}(T) \Rightarrow T'(j)\ E\ T''(k))$ (Given)

3 $\alpha(i) \leq j < \alpha(i+1)$ (Assumption)

4 $\alpha''(i) \leq k < \alpha''(i+1)$ (Assumption)

5 $\exists k'.\ \alpha'(i) \leq k' < \alpha'(i+1)$ (α' is strictly increasing)

6 $\alpha'(i) \leq k' < \alpha'(i+1)$ ($\exists$-elim)

7 $((j < \mathit{len}(T)) \Leftrightarrow (k' < \mathit{len}(T'))) \wedge (j < \mathit{len}(T) \Rightarrow T(j)\ E\ T'(k'))$
(line 1 with lines 3 and 6)

8 $((k' < \mathit{len}(T')) \Leftrightarrow (k < \mathit{len}(T''))) \wedge (k' < \mathit{len}(T) \Rightarrow T'(k')\ E\ T''(k))$
(line 2 with lines 6 and 4)

9 $((j < \mathit{len}(T)) \Leftrightarrow (k < \mathit{len}(T'')))$
(First conjuncts of lines 7 and 8 and transitivity of $\Leftrightarrow$)

10 $j < \mathit{len}(T)$ (Assumption)

11 $T(j)\ E\ T'(k')$ (Line 7 second conjunct and line 10)

12 $k' < len(T')$ (Line 7 first conjunct and line 10)

13 $T'(k')\ E\ T''(k)$ (Line 8 second conjunct and above)

14 $T(j)\ E\ T''(k)$ (Transitivity of E and lines 11 and 13)

15 $j < len(T) \Rightarrow T(j)\ E\ T''(k)$ ($\Rightarrow$-introduction lines 10 and 14)

16 $((j < len(T)) \Leftrightarrow (k < len(T''))) \wedge (j < len(T) \Rightarrow T(j)\ E\ T''(k))$

($\wedge$-intro lines 9 and above)

17 $\alpha(i) \leq j < \alpha(i+1) \wedge \alpha''(i) \leq k < \alpha''(i+1) \Rightarrow$

$((j < len(T)) \Leftrightarrow (k < len(T''))) \wedge (j < len(T) \Rightarrow T(j)\ E\ T''(k))$

($\Rightarrow$-intro: 3 and 4)

□

Given Lemma 7, we can now establish that $\sim_E$ is an equivalence relation.

Theorem 11. *$\sim_E$ is an equivalence relation.*

Proof. That $\sim_E$ is reflexive and symmetric follows immediately from Lemma 7 and the definition of $\sim_E$. Transitivity also requires Theorem 10. We have $T \sim_E T'$ and $T' \sim_E T''$ and must show $T \sim_E T''$. From the definition of $\sim_E$ applied to our two assumptions, we have $matches(T, T', \alpha, \beta, E)$ and $matches(T', T'', \alpha', \beta', E)$. By Theorem 10 we can convert these assumptions to $matches(T, T', B_T, B_{T'}, E)$ and $matches(T', T'', B_{T'}, B_{T''}, E)$. By Lemma 7 we then have $matches(T, T'', B_T, B_{T''}, E)$ which implies $T \sim_E T''$. □

Furthermore, given an appropriate equivalence relation, we can even compose $\sim_E$ statements involving different Es.

Theorem 12. *Let E'' be an equivalence relation satisfying the following.*

$$\forall a, b, c.\ (a\ E\ b \wedge b\ E'\ c \Rightarrow a\ E''\ c)$$

Then $T \sim_E T'$ and $T' \sim_{E'} T''$ implies $T \sim_{E''} T''$.

Proof. We first apply the definition of $\sim$ (Definition 19) to obtain $matches(T, T', \alpha, \beta, E)$ and $matches(T', T'', \alpha', \beta', E')$ for some $\alpha, \beta, \alpha', \beta'$. We then apply Theorem 10 to obtain $matches(T, T', B_T, B_{T'}, E)$ and $matches(T', T'', B_{T'}, B_{T''}, E)$. We now show that $matches(T, T'', B_T, B_{T''}, E'')$ holds and thus $T \sim_{E''} T''$.

Let i, j, k be such that $B_T(i) \leq j < B_T(i+1)$ and $B_{T''}(i) \leq k < B_{T''}(i+1)$. We must show $j < len(T) \Leftrightarrow k < len(T'')$ and $j < len(T)$ implies $T(j)\ E''\ T''(k)$. From $matches(T, T', B_T, B_{T'}, E)$ we have that $T(j)\ E\ T'(B_{T'}(i))$. From our assumption $matches(T', T'', B_{T'}, B_{T''}, E'')$ we have $T'(B_{T'}(i))\ E\ T''(k)$. Combining these, we have $T(j)\ E''\ T(k)$, which is one of our goals.

For $j < len(T) \Leftrightarrow k < len(T'')$, we note that $matches(T, T', B_T, B_{T'}, E)$ implies $j < len(T) \Leftrightarrow B_{T'}(i) < len(T')$ and $matches(T', T'', B_{T'}, B_{T''}, E')$ implies $B_{T'}(i) < len(T') \Leftrightarrow k < len(T'')$. Combining these, we have our goal of $j < len(T) \Leftrightarrow k < len(T'')$. □

3.2.1 Mapping Between Stuttering Equivalent Traces

The following Lemma will be very useful in several upcoming proofs. It establishes the existence of functions that map between related positions in stuttering equivalent traces.

Lemma 8. *If $T \sim_E T'$ then there exist functions $f : \mathbb{N} \to \mathbb{N}$ and $f^{-1} : \mathbb{N} \to \mathbb{N}$ such that $\forall i.\ T_i \sim_E T'_{f(i)}$ and $\forall i.\ T_{f^{-1}(i)} \sim_E T'_i$ and f and f^{-1} are monotonic and $\forall i.\ f^{-1}(f(i)) \leq i$.*

Proof. Since $T \sim_E T'$ we have that there are strictly increasing functions α, β with the properties listed in Definition 18 and reproduced below.

$$\alpha, \beta \text{ strictly increasing} \tag{3.10}$$

$$\alpha(0) = \beta(0) = 0 \tag{3.11}$$

$$\begin{aligned} &\forall i, j, k.\ \alpha(i) \leq j < \alpha(i+1) \wedge \beta(i) \leq k < \beta(i+1) \Rightarrow \\ &\qquad \big(j < len(T) \Leftrightarrow k < len(T')\big) \wedge \big(j < len(T) \Rightarrow (T(j))\ E\ (T'(k))\big) \end{aligned} \tag{3.12}$$

We first define $f(i)$. Since $i \in \mathbb{N}$ we have $i \geq 0$. Because α is strictly increasing and $\alpha(0) = 0$ and $i \geq 0$, we have that there exists a c such that $\alpha(c) \leq i < \alpha(c+1)$. Given

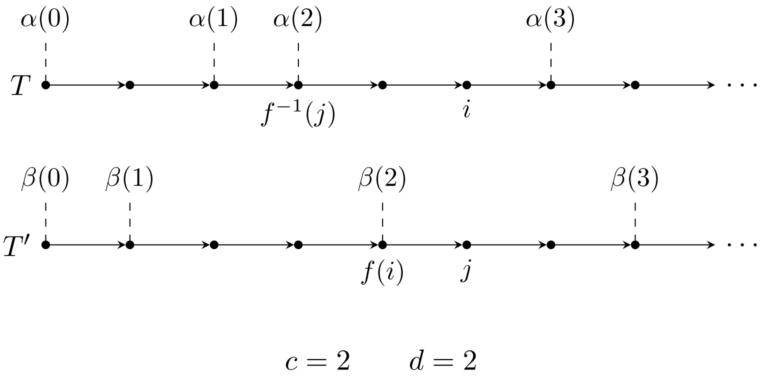


Figure 3.3: Example depicting the sequences, functions, and variables involved in the proof of Lemma 8.

this c, we then define $f(i)$ as follows.

$$f(i) = \begin{cases} \mathit{len}(T') & \text{if } i \geq \mathit{len}(T) \\ \beta(c) & \text{if } i < \mathit{len}(T) \end{cases}$$

Essentially, by discarding the first i elements of T, we have changed the starting point of our trace and thus also the starting point for the matching functions α and β. The constant c is the index for α that brackets i. That is, $\alpha(c) \leq i < \alpha(c+1)$. We use this value to appropriately adjust the starting point of T'. Figure 3.3 gives an overview.

We first present the proof for the $T_i \sim_E T'_{f(i)}$ conjunct and the properties of f, then we give the proof of $T_{f^{-1}(i)} \sim_E T'_i$ and the properties of f^{-1}.

$T_i \sim_E T'_{f(i)}$ and Properties of f

We first handle the case where $i \geq \mathit{len}(T)$. In this case, $T_i = \epsilon$ and $T'_{f(i)} = \epsilon$ and $\epsilon \sim_E \epsilon$. We now consider the case where $i < \mathit{len}(T)$.

We need to produce functions α' and β' satisfying the conditions in Definition 18. In constructing these, we are allowed to use the α and β that we know exist due to the assumption $T \sim_E T'$ (formulas (3.10), (3.11), and (3.12)). The functions are as follows.

$$\alpha'(n) = max(\alpha(n+c) - i, 0) \tag{3.13}$$

$$\beta'(n) = max(\beta(n+c) - f(i), 0) \tag{3.14}$$

$\boldsymbol{\alpha'(0) = \beta'(0) = 0}$ We first show that $\alpha'(0) = \beta'(0) = 0$. We have $\alpha'(0) = max(\alpha(c) - i, 0)$. From the definition of c, above, we have $\alpha(c) \leq i$. This implies $\alpha(c) - i \leq 0$ which implies $max(\alpha(c) - i, 0) = 0$. For $\beta'(0)$ we have $\beta'(0) = max(\beta(c) - f(i), 0)$ and $f(i) = \beta(c)$, which gives us $\beta'(0) = max(\beta(c) - \beta(c), 0) = 0$.

Strictly Increasing We must also check that α' and β' are strictly increasing. We will first consider α'. To show α' is strictly increasing, it suffices to show that $\alpha'(1) > 0$. This is due to the *max* operation in the definition of α' and the fact that α is strictly increasing. Given the definition of α' (3.13), we have that if $\alpha'(n) > 0$ for some n, then $\alpha'(n) = \alpha(n+c) - i$. Since α is strictly increasing, we have $\alpha(n+c+1) > \alpha(n+c)$ and thus $\alpha(n+c+1) - i > \alpha(n+c+1) - 1$ and finally $\alpha'(n+1) > \alpha'(n)$. Thus, $\alpha'(n) > 0$ implies α' is strictly increasing on the interval $[n, \infty)$. As we have already shown $\alpha'(0) = 0$, showing $\alpha'(1) > 0$ will give us that α' is strictly increasing on the interval $[0, \infty)$, as desired.

To show that $\alpha'(1) > 0$, note that $\alpha'(1) = max(\alpha(1+c) - i, 0)$. We have from our choice of c that $i < \alpha(c+1)$. This implies $\alpha(1+c) - i > 0$ which implies $max(\alpha(1+c) - i, 0) > 0$.

The case for β' is similar. Since β is also strictly increasing and β' is defined using *max* with 0, the same reasoning applies and to show β' is strictly increasing it suffices to show that $\beta'(1) > 0$. We have $\beta'(1) = max(\beta(1+c) - f(i), 0)$. The definition of $f(i)$ is $\beta(c)$, so we have $\beta'(1) = max(\beta(1+c) - \beta(c), 0)$. Since β is strictly increasing we have $\beta(1+c) > \beta(c)$ implying that $\beta'(1) > 0$.

End of Last Blocks Coincide Let j' and k' satisfy $\alpha'(i') \leq j' < \alpha'(i'+1)$ and $\beta'(i') \leq k' < \beta(i'+1)$. We must show that $(j' < len(T_i)) \Leftrightarrow (k' < len(T'_{f(i)}))$.

Expanding the definition of α' and β' we have

$$\alpha(i'+c) - i \leq j' < \alpha(i'+1+c) - i$$
$$\beta(i'+c) - f(i) \leq k' < \beta(i'+1+c) - f(i)$$

Rewriting by moving i and $f(i)$ to the inside of the inequalities, we obtain

$$\alpha(i'+c) \leq j'+i < \alpha(i'+1+c) \tag{3.15}$$
$$\beta(i'+c) \leq k'+f(i) < \beta(i'+1+c) \tag{3.16}$$

Note that now we have $j'+i$ is a quantity bounded between $\alpha(i'+c)$ and $\alpha(i'+c+1)$ (consecutive values of α) and similarly for β in the second formula. By (3.12) we then have $(j'+i < \mathit{len}(T)) \Leftrightarrow (k'+f(i) < \mathit{len}(T'))$. This implies

$$(j' < \mathit{len}(T) - i) \Leftrightarrow (k' < \mathit{len}(T') - f(i))$$

Since $\mathit{len}(T_i) = \mathit{len}(T) - i$ and $\mathit{len}(T'_{f(i)}) = \mathit{len}(T') - f(i)$ this gives us

$$(j' < \mathit{len}(T_i)) \Leftrightarrow (k' < \mathit{len}(T'_{f(i)}))$$

which is our goal.

E-related To show that $j' < \mathit{len}(T_i) \Rightarrow (T_i(j'))\ E\ (T'_{f(i)}(k'))$ we first assume $j' < \mathit{len}(T_i)$ and apply the conclusion above (that $j' < \mathit{len}(T_i) \Leftrightarrow k' < \mathit{len}(T'_{f(i)})$) to conclude $k' < \mathit{len}(T'_{f(i)})$. This ensures that both $T_i(j')$ and $T'_{f(i)}(k')$ are defined. Next, we note that $T_i(j') = T(i+j')$ and $T'_{f(i)}(k') = T'(f(i)+k')$. Thus, it suffices to show that $\big(T(i+j')\big)\ E\ \big(T'(f(i)+k')\big)$. From (3.15), (3.16), and (3.12) we have $\big(T(j'+i)\big)\ E\ \big(T'(k'+f(i))\big)$ which, together with commutativity of $+$, proves our goal.

Monotonicity of f Recall that for $i \geq \mathit{len}(T)$ we have $f(i) = \mathit{len}(T')$ and for $i < \mathit{len}(T)$ we have $f(i) = \beta(c)$ for the c such that $\alpha(c) \leq i < \alpha(c+1)$. We now prove that such an f is monotonic. Suppose $a \leq b$. We will show that $f(a) \leq f(b)$. There are three cases. If $a \geq \mathit{len}(T)$ then $b \geq \mathit{len}(T)$ and $f(a) = f(b) = \mathit{len}(T')$. If

$a < \mathit{len}(T)$ and $b \geq \mathit{len}(T)$ then $f(b) = \mathit{len}(T')$. For $f(a)$, we first choose c such that $\alpha(c) \leq a < \alpha(c+1)$. By (3.12) and $a < \mathit{len}(T)$ we then have $\beta(c) < \mathit{len}(T')$. Since $f(a) = \beta(c)$ we have $f(a) < \mathit{len}(T')$. Thus $f(a) < f(b)$.

Finally, we consider $a < \mathit{len}(T)$ and $b < \mathit{len}(T)$. We first choose c such that $\alpha(c) \leq a < \alpha(c+1)$ and d such that $\alpha(d) \leq b < \alpha(d+1)$. Since α is strictly increasing, this can always be done. Since $a \leq b$ and α is strictly increasing, we have $c \leq d$. Now, since $c \leq d$ and β is strictly increasing, we have $\beta(c) \leq \beta(d)$. Since $f(a) = \beta(c)$ and $f(b) = \beta(d)$ we then have $f(a) \leq f(b)$.

Definition of f^{-1}: We are given some $i \geq 0$. We first let d be the number such that $\beta(d) \leq i < \beta(d+1)$. Since β is strictly increasing, such a d always exists. We then define $f^{-1}(i)$ as follows.

$$f^{-1}(i) = \begin{cases} \mathit{len}(T) & \text{if } i \geq \mathit{len}(T') \\ \alpha(d) & \text{if } i < \mathit{len}(T') \end{cases}$$

$T_{f^{-1}(i)} \sim_E T'_i$ and Properties of f^{-1}

We now show that $\forall i.\ T_{f^{-1}(i)} \sim_E T'_i$. Similar to before, the α' and β' that show this are

$$\begin{aligned} \alpha'(n) &= \mathit{max}(\alpha(n+d) - f^{-1}(i), 0) \\ \beta'(n) &= \mathit{max}(\beta(n+d) - i, 0) \end{aligned} \tag{3.17}$$

For $i \geq \mathit{len}(T')$, we have $T'_i = \epsilon$ and $T_{f^{-1}(i)} = T_{\mathit{len}(T)} = \epsilon$. Since $\epsilon \sim_E \epsilon$, we have $T_{f^{-1}(i)} \sim_E T'_i$. We next consider the case where $i < \mathit{len}(T')$, considering in turn each property that must hold of α' and β'.

$\alpha'(0) = \beta'(0) = 0$ We have $\alpha'(0) = \mathit{max}(\alpha(d) - f^{-1}(i), 0)$. We have $f^{-1}(i) = \alpha(d)$. Thus, $\alpha'(0) = \mathit{max}(\alpha(d) - \alpha(d), 0) = 0$. For $\beta'(0)$, we have $\beta'(0) = \mathit{max}(\beta(0+d) - i, 0)$. We have from our choice of d that $\beta(d) \leq i$. Thus, $\beta(d) - i \leq 0$ and $\mathit{max}(\beta(d) - i, 0) = 0$.

Strictly Increasing As before, $\alpha'(1) > 0$ will be sufficient to prove α' is strictly increasing (given the assumption that α is strictly increasing) and similarly for β'. We have

$\alpha'(1) = max(\alpha(1 + d) - f^{-1}(i), 0) = max(\alpha(1 + d) - \alpha(d), 0)$. Since α is strictly increasing, we have $\alpha(1 + d) - \alpha(d) > 0$ which implies $\alpha'(1) > 0$.

For $\beta'(1)$, we have $\beta'(1) = max(\beta(1 + d) - i, 0)$. We have from our choice of d that $i < \beta(d + 1)$ which implies $\beta(1 + d) - i > 0$ and thus $\beta'(1) > 0$.

End of Last Blocks Coincide Suppose $\alpha'(i') \leq j' < \alpha'(i'{+}1)$ and $\beta'(i') \leq k' < \beta(i'{+}1)$. We must show that $(j' < len(T_{f^{-1}(i)})) \Leftrightarrow (k' < len(T'_i))$.

Expanding the definition of α' and β' we have

$$\alpha(i' + d) - f^{-1}(i) \leq j' < \alpha(i' + 1 + d) - f^{-1}(i)$$
$$\beta(i' + d) - i \leq k' < \beta(i' + d + 1) - i$$

Rewriting by moving i and $f^{-1}(i)$ to the inside of the inequalities, we obtain

$$\alpha(i' + d) \leq j' + f^{-1}(i) < \alpha(i' + 1 + d) \tag{3.18}$$
$$\beta(i' + d) \leq k' + i < \beta(i' + d + 1) \tag{3.19}$$

Note that now we have $j' + f^{-1}(i)$ is a quantity bounded between $\alpha(i' + d)$ and $\alpha(i' + d + 1)$ (consecutive values of α) and similarly for β in the second formula. By (3.12) we then have $(j' + f^{-1}(i) < len(T)) \Leftrightarrow (k' + i < len(T'))$. This implies $(j' < len(T) - f^{-1}(i)) \Leftrightarrow (k' < len(T') - i)$. Since $len(T_{f^{-1}(i)}) = len(T) - f^{-1}(i)$ and $len(T'_i) = len(T') - i$ this gives us $(j' < len(T_{f^{-1}(i)})) \Leftrightarrow (k' < len(T'_i))$ which is our goal.

E-related To show that $j' < len(T_{f^{-1}(i)}) \Rightarrow (T_{f^{-1}(i)}(j'))\ E\ (T'_i(k'))$ we first assume that $j' < len(T_{f^{-1}(i)})$ and apply our result above to conclude $k' < len(T'_i)$. This ensures that both $T_{f^{-1}(i)}(j')$ and $T'_i(k')$ are defined. We next note that $T_{f^{-1}(i)}(j') = T(f^{-1}(i) + j')$ and $T'_i(k') = T'(i + k')$. Thus, it suffices to show that $\big(T(f^{-1}(i) + j')\big)\ E\ \big(T'(i + k')\big)$. From (3.18), (3.19), and (3.12) we have $\big(T(j' + f^{-1}(i))\big)\ E\ \big(T'(k' + i)\big)$ which, together with commutativity of $+$, proves our goal.

Monotonicity of f^{-1} Recall that for $i \geq \mathit{len}(T')$ we have $f^{-1}(i) = \mathit{len}(T)$ and for $i < \mathit{len}(T')$ we have $f^{-1}(i) = \alpha(d)$ for the d such that $\beta(d) \leq i < \beta(d+1)$. We now prove that such an f^{-1} is monotonic. Suppose $a \leq b$. We will show that $f^{-1}(a) \leq f^{-1}(b)$. There are three cases. If $a \geq \mathit{len}(T')$ and $b \geq \mathit{len}(T')$ then $f^{-1}(a) = f^{-1}(b) = \mathit{len}(T)$. If $a < \mathit{len}(T')$ and $b \geq \mathit{len}(T')$ then $f^{-1}(b) = \mathit{len}(T)$. For $f^{-1}(a)$, we first choose the d such that $\beta(d) \leq a < \beta(d+1)$. By (3.12) and $a < \mathit{len}(T')$ we then have $\alpha(d) < \mathit{len}(T)$. Since $f^{-1}(a) = \alpha(d)$ we have $f^{-1}(a) < \mathit{len}(T)$. Thus $f^{-1}(a) < f^{-1}(b)$.

Finally, we consider $a < \mathit{len}(T')$ and $b < \mathit{len}(T')$. To compute $f^{-1}(a)$ and $f^{-1}(b)$, we first choose d_1 such that $\beta(d_1) \leq a < \beta(d_1+1)$ and d_2 such that $\beta(d_2) \leq b < \beta(d_2+1)$. Since β is strictly increasing and $a \leq b$ we have $d_1 \leq d_2$. Since α is strictly increasing, we then have $\alpha(d_1) \leq \alpha(d_2)$. Since $f^{-1}(a) = \alpha(d_1)$ and $f^{-1}(b) = \alpha(d_2)$ we then have $f^{-1}(d_1) \leq f^{-1}(d_2)$.

Inverse Relationship We now show that $f^{-1}(f(i)) \leq i$. Let i be an arbitrary natural number. If $i \geq \mathit{len}(T)$ then $f(i) = \mathit{len}(T')$ and $f^{-1}(\mathit{len}(T')) = \mathit{len}(T)$. Since $i \geq \mathit{len}(T)$ we have $f^{-1}(f(i)) = \mathit{len}(T) \leq i$. We now consider the case where $i < \mathit{len}(T)$.

In this case, we have $f(i) = \beta(c)$ for some c such that $\alpha(c) \leq i < \alpha(c+1)$ and $f^{-1}(f(i)) = f^{-1}(\beta(c)) = \alpha(d)$ for some d such that $\beta(d) \leq \beta(c) < \beta(d+1)$. Since β is strictly increasing, $\beta(d) \leq \beta(c) < \beta(d+1)$ implies that $c = d$. We can then use this equality to derive from $f^{-1}(f(i)) = \alpha(d)$ the fact that $f^{-1}(f(i)) = \alpha(c)$. Since we have $\alpha(c) \leq i$ we then have $f^{-1}(f(i)) \leq i$ which was our goal.

□

3.2.2 Stuttering Containment

We now use this notion of stuttering equivalence to define stuttering containment for sets and define stuttering equivalence of trace sets as mutual containment.

Definition 21. *Let $\mathbf{T}$ and $\mathbf{T}'$ be sets of traces. Then $\mathbf{T}'$ **E-stuttering contains** $\mathbf{T}$, written $\mathbf{T} \lesssim_E \mathbf{T}'$, iff $\forall T \in \mathbf{T}.\ \exists T' \in \mathbf{T}'.\ T \sim_E T'$. We say $\mathbf{T}$ is **E-stuttering equivalent** to $\mathbf{T}'$, written $\mathbf{T} \approx_E \mathbf{T}'$, iff $\mathbf{T} \lesssim_E \mathbf{T}'$ and $\mathbf{T}' \lesssim_E \mathbf{T}$.*

When $\mathbf{T} \approx_E \mathbf{T}'$ and the relation E is clear from context we will simply say that $\mathbf{T}$ and $\mathbf{T}'$ are *stuttering equivalent*.

We can now obtain a version of Theorem 12 for stuttering containment.

Theorem 13. *Let E'' be an equivalence relation satisfying the following.*

$$\forall a, b, c.\ (a\ E\ b \wedge b\ E'\ c \Rightarrow a\ E''\ c)$$

Then $\mathbf{T} \lesssim_E \mathbf{T}'$ and $\mathbf{T}' \lesssim_{E'} \mathbf{T}''$ implies $\mathbf{T} \lesssim_{E''} \mathbf{T}''$.

Proof. We must show the following.

$$\forall T \in \mathbf{T}.\ \exists T'' \in \mathbf{T}''.\ T \sim_{E''} T''$$

From our assumption $\mathbf{T} \lesssim_E \mathbf{T}'$ we have

$$\forall T \in \mathbf{T}.\ \exists T' \in \mathbf{T}'.\ T \sim_E T'$$

From our assumption $\mathbf{T}' \lesssim_{E'} \mathbf{T}''$ we have

$$\forall T' \in \mathbf{T}'.\ \exists T'' \in \mathbf{T}''.\ T' \sim_{E'} T''$$

Combining these we have

$$\forall T \in \mathbf{T}.\ \exists T' \in \mathbf{T}', T'' \in \mathbf{T}''.\ T \sim_E T' \wedge T' \sim_{E'} T''$$

We can then apply Theorem 12 to obtain

$$\forall T \in \mathbf{T}.\ \exists T' \in \mathbf{T}', T'' \in \mathbf{T}''.\ T \sim_{E''} T''$$

Eliminating the quantification on T' then gives us our goal. □

3.2.3 Programs and Stuttering Equivalence

We now tie these general notions of stuttering equivalence and containment to programs and give some examples of stuttering equivalent programs.

The trace sets of interest for programs are those obtained when executing the program from a state satisfying some precondition. Thus, for some programs P and P' and preconditions Q and Q', we will be interested in questions such as whether the relation *traces*$(\!(P \mid Q)\!) \lesssim_E$ *traces*$(\!(P' \mid Q')\!)$ holds for some equivalence relation E. Since the semantics of a program can be viewed as the set of traces produced by that program, this provides a connection between the semantics of P and the semantics of P' (provided each is started in a satisfactory initial state). This will form the basis of our notion of *abstraction*.

Definition 22. *A program P' with precondition Q' is an **abstraction** of a program P with precondition Q, with respect to an equivalence relation E iff Q and Q' are separation logic formulae and*

$$\textit{traces}(\!(P \mid Q)\!) \lesssim_E \textit{traces}(\!(P' \mid Q')\!)$$

When Q, Q' and E are clear from context, we will just say that P' is an abstraction of P.

This property can be more or less useful depending on the particular preconditions involved (and also depending on the equivalence relation utilized). For example, if Q is false, then we can establish this for any P, P', Q'. The conciseness of the term *abstraction* is useful in informal discussions, and we will restrict ourselves to using it in such settings. For the presentation of the formal development, we will use the more precise notation developed previously (i.e. $\lesssim_E, \approx_E$, etc.).

The strongest correspondence between programs P and P' is given by the statement *traces*$(\!(P \mid \mathsf{true})\!) \approx_\equiv$ *traces*$(\!(P' \mid \mathsf{true})\!)$, where $\equiv$ is the identity relation on execution states. Since our execution states include the current continuation, this will only hold when $P = P'$, where the equality is up to reordering of labeled continuations (with the initial continuation not subject to reordering). In order to get a more interesting (and weaker) correspondence, we move to the following notion of equality. Let $\doteq$ be the least relation satisfying the following.

$$\begin{array}{rcl}
\mathbf{goto}(l,(s,h)) & \doteq & \mathbf{goto}(l,(s,h)) \\
\langle k,(s,h)\rangle & \doteq & \langle k',(s,h)\rangle \\
\mathbf{final}(s,h) & \doteq & \mathbf{final}(s,h) \\
\mathbf{error} & \doteq & \mathbf{error}
\end{array}$$

Note that $\doteq$ identifies exactly those states that are the same modulo the current continuation k. Now we can describe programs that involve different continuations, but which produce stuttering equivalent sequences of store, heap pairs (and location, store, heap triples in the case of goto states). Figure 3.4 lists four programs that are stuttering equivalent in the sense that for any P and P' in the figure, we have *traces*$(\!(P \mid \mathsf{true})\!) \approx_{\doteq}$ *traces*$(\!(P' \mid \mathsf{true})\!)$. In each case, the traces of P_i consist of one occurrence of the state $\mathbf{goto}(\mathsf{L}_0,(s,h))$ followed by either one (as in P_1, P_2) or two (as in P_3, P_4) occurrences of the state $\langle k,(s,h)\rangle$ for some k, followed by one (as in P_1, P_3, P_4) or two (as in P_2) occurrences of the state $\langle k,(s[\mathsf{a} \to 0],h)\rangle$, followed by the traces starting from $\mathbf{goto}(\mathsf{L}_1,(s[\mathsf{a} \to 0],h))$. Examining one of the example programs in detail, we see that traces produced by P_3 have the following form.

$$\begin{array}{c}
\mathbf{goto}(\mathsf{L}_0,(s,h)) \\
\langle \mathsf{branch}\ \ldots \mathsf{end},(s,h)\rangle \\
\langle \mathsf{a} := 0\texttt{;}\,\mathsf{goto}\ \mathsf{L}_1,(s,h)\rangle \\
\langle \mathsf{goto}\ \mathsf{L}_1,(s[\mathsf{a} \to 0],h)\rangle \\
\mathbf{goto}(\mathsf{L}_1,(s[\mathsf{a} \to 0],h)) \\
\langle \mathsf{halt},(s[\mathsf{a} \to 0],h)\rangle \\
\mathbf{final}(s[\mathsf{a} \to 0],h)
\end{array}$$

It is also instructive to consider which changes violate stuttering equivalence. The program below, while quite similar to P_4, is not stuttering equivalent from precondition

```
$P_1 \overset{\text{def}}{=}$
        $L_0$ : a := $0$; goto $L_1$;
        $L_1$ : halt
            end
```

```
$P_2 \overset{\text{def}}{=}$
        $L_0$ : a := $0$; a := $0$; goto $L_1$;
        $L_1$ : halt
            end
```

```
$P_3 \overset{\text{def}}{=}$
    $L_0$ : branch  true $\Rightarrow$ a := $0$; goto $L_1$,
                    true $\Rightarrow$ a := $0$; goto $L_1$;
            end
    $L_1$ : halt
            end
```

```
$P_4 \overset{\text{def}}{=}$
    $L_0$ : branch  x $> 0 \Rightarrow$ a := $0$; goto $L_1$,
                    x $\leq 0 \Rightarrow$ a := $0$; goto $L_1$
            end;
    $L_1$ : halt
            end
```

Figure 3.4: Four examples of stuttering equivalent programs. Each example involves a different continuation at L_0.

true.

```
$P_4' \overset{\text{def}}{=}$
    $L_0$ : branch  x $> 0 \Rightarrow$ a := $0$; goto $L_1$,
                    x $< 0 \Rightarrow$ a := $0$; goto $L_1$
            end;
    $L_1$ : halt
            end
```

The reason this program is not stuttering equivalent to the programs in Figure 3.4 is that, due to the lack of a branch for $x = 0$ in the continuation at L_0, P_4' does not contain traces in which $s(x) = 0$ (where s is the store associated with some state in the trace). However, P_4' *is* stuttering equivalent to the other programs when evaluated from the precondition $x \neq 0$. This is an example of the importance of the initial conditions (as represented by the precondition). By removing certain sets of traces from consideration, the precondition can cause programs that do not correspond in general to be stuttering equivalent.

There are, however, programs which cannot be made stuttering equivalent according to $\doteq$ regardless of the precondition. Consider the program below.

```
P'_1 ≝
    L_0 : a := 0; b := 0; b := 1; goto L_1;
    L_1 : halt
          end
```

This program is similar to P_1 except that it mentions an additional variable b. The traces of P_1' contain states where $s(\mathsf{b}) = 0$ and states where $s(\mathsf{b}) = 1$. The value of b in any trace of P_1 will always be constant, preventing these two programs to from being related by $\lesssim_{\doteq}$ for any precondition other than false.

However, these programs are stuttering equivalent if we change the equivalence relation on execution states to one that does not take into account the value of b. Consider the equivalence relation given below, which is the $=_V$ relation on stores (Definition 1) lifted to execution states.

Definition 23. $=_V$ *is the least relation satisfying the following.*

$$\begin{array}{rcll}
\mathbf{goto}(l,(s,h)) & =_V & \mathbf{goto}(l,(s',h)) & \textit{iff } s =_V s' \\
\langle k,(s,h)\rangle & =_V & \langle k',(s',h)\rangle & \textit{iff } s =_V s' \\
\mathbf{final}(s,h) & =_V & \mathbf{final}(s',h) & \textit{iff } s =_V s' \\
\mathbf{error} & =_V & \mathbf{error} &
\end{array}$$

With this relation, we can now specify the correspondence between P_1 and P_1'. We have *traces*$(\!(P_1 \mid \mathsf{true})\!) \approx_{(=_{\{\mathsf{a}\}})}$ *traces*$(\!(P_1' \mid \mathsf{true})\!)$.

Heap-Manipulating Examples New commands can also be added to heap-manipulating programs while preserving this version of stuttering equivalence. Figure 3.5 gives some examples of relationships between programs that involve the heap. P_5 gives a program that frees a linked list at x with length a. As it frees elements, it keeps track of the length of the remaining portion of the list by updating a.

$P_5 \overset{\text{def}}{=}$

```
L0 : goto L1;
L1 : branch $x \neq$ nil ⇒
                 t := x;
                 x := x.next;
                 free t;
                 a := a − 1;
                 goto L1,
             x = nil ⇒ halt
     end
```

$P_6 \overset{\text{def}}{=}$

```
L0 : goto L1;
L1 : branch a > 0 ⇒
                 t := x;
                 x := x.next;
                 free t;
                 a := a − 1;
                 goto L1,
             a = 0 ⇒ halt
     end
```

$P_7 \overset{\text{def}}{=}$

```
L0 : goto L1;
L1 : branch a > 0 ⇒
                 a := a − 1;
                 goto L1,
             a = 0 ⇒ halt
     end
```

$P_8 \overset{\text{def}}{=}$

```
L0 : a := ?; goto L1;
L1 : branch a > 0 ⇒
                 a := a − 1;
                 goto L1,
             a = 0 ⇒ halt
     end
```

$$\begin{aligned}
\mathit{traces}(\!(P_5 \mid \mathit{ls}(\mathsf{a},\mathsf{x},\mathsf{nil}))\!) \quad &\approx_{=_{\{\mathsf{x},\mathsf{t},\mathsf{a}\}}} \quad \mathit{traces}(\!(P_6 \mid \mathit{ls}(\mathsf{a},\mathsf{x},\mathsf{nil}))\!)\\
\mathit{traces}(\!(P_6 \mid \mathit{ls}(\mathsf{a},\mathsf{x},\mathsf{nil}))\!) \quad &\approx_{\overset{\mathsf{s}}{=}_{\{\mathsf{a}\}}} \quad \mathit{traces}(\!(P_7 \mid \mathit{ls}(\mathsf{a},\mathsf{x},\mathsf{nil}))\!)\\
\mathit{traces}(\!(P_7 \mid \mathit{ls}(\mathsf{a},\mathsf{x},\mathsf{nil}))\!) \quad &\lesssim_{=_{\{\mathsf{a}\}}} \quad \mathit{traces}(\!(P_8 \mid \exists a.\ \mathit{ls}(a,\mathsf{x},\mathsf{nil}))\!)
\end{aligned}$$

Figure 3.5: Increasingly weaker abstractions of P_5.

When started from the precondition $\mathit{ls}(\mathsf{a},\mathsf{x},\mathsf{nil})$ this program is *safe*, in the sense that no traces from this precondition end with $\mathbf{error}$. This corresponds to the LTSL property $\sim\!(\mathbf{F}(\mathit{err}))$.

The program also has the property that for every state of the form $\mathbf{goto}(\mathsf{L}_1,(s,h))$, we have $(s,h) \models_X \mathit{ls}(\mathsf{a},\mathsf{x},\mathsf{nil})$. Put another way, $\mathit{ls}(\mathsf{a},\mathsf{x},\mathsf{nil})$ is an *invariant* of location L_1. This corresponds to the LTSL property $\mathbf{G}(\mathit{atloc}(\mathsf{L}_1) \Rightarrow \mathit{ls}(\mathsf{a},\mathsf{x},\mathsf{nil}))$.

Finally, the program always *terminates*, meaning that its trace set contains no infinite traces. The LTSL formula corresponding to termination is $\mathbf{F}(\mathit{final})$.

Program P_6 is stuttering equivalent to P_5 in the sense that they satisfy

$$\mathit{traces}(\!(P_5 \mid \mathit{ls}(\mathsf{a},\mathsf{x},\mathsf{nil}))\!) \approx_{\doteq} \mathit{traces}(\!(P_6 \mid \mathit{ls}(\mathsf{a},\mathsf{x},\mathsf{nil}))\!)$$

That is, when started in a state satisfying $\mathit{ls}(\mathsf{a},\mathsf{x},\mathsf{nil})$, their traces consist of the same sequence of memory states with the only difference being possible repetition of some states. In this case, there is not even any repetition. The only difference between the two programs is that P_5 branches on $\mathsf{x} \neq \mathsf{nil}$, whereas P_6 branches on $\mathsf{a} > 0$. Since a is always equal to the length of the list at x, these conditions are equivalent and result in the same set of traces.

Program P_7 consists solely of the commands involving a. Such a program is not stuttering equivalent to P_5 or P_6 given any of the equality relations on execution states that have been discussed so far. However, it is stuttering equivalent given the relation below.

Definition 24. *$\overset{\mathrm{s}}{=}_V$ is the least relation on execution states that satisfies the following.*

$$\begin{array}{rcll}
\mathbf{goto}(l,(s,h)) & \overset{\mathrm{s}}{=}_V & \mathbf{goto}(l,(s',h')) & \textit{iff } s =_V s' \\
\langle k,(s,h)\rangle & \overset{\mathrm{s}}{=}_V & \langle k',(s',h')\rangle & \textit{iff } s =_V s' \\
\mathbf{final}(s,h) & \overset{\mathrm{s}}{=}_V & \mathbf{final}(s',h') & \textit{iff } s =_V s' \\
\mathbf{error} & \overset{\mathrm{s}}{=}_V & \mathbf{error} &
\end{array}$$

The $\overset{\mathrm{s}}{=}_V$ relation is the same as $=_V$ except that the heaps are not required to be the same. We can now state the relationship between P_6 and P_7. It is

$$\mathit{traces}(\!(P_6 \mid \mathit{ls}(\mathsf{a},\mathsf{x},\mathsf{nil}))\!) \approx_{\overset{\mathrm{s}}{=}_{\{\mathsf{a}\}}} \mathit{traces}(\!(P_7 \mid \mathit{ls}(\mathsf{a},\mathsf{x},\mathsf{nil}))\!)$$

and the same relation holds between P_5 and P_7.

The program P_8 is an example of a program that is not stuttering equivalent to any of the previous programs, but does stuttering contain the traces of some of them. We have the following.

$$\mathit{traces}(\!(P_7 \mid \mathit{ls}(\mathsf{a},\mathsf{x},\mathsf{nil}))\!) \lesssim_{=_{\{\mathsf{a}\}}} \mathit{traces}(\!(P_8 \mid \mathit{ls}(\mathsf{a},\mathsf{x},\mathsf{nil}))\!)$$

The program P_8 contains traces stuttering equivalent to the traces in P_7, but also contains traces where the non-deterministic assignment causes a to have a value other than the length of the list.

The non-deterministic assignment can also be used to ensure that we consider executions where a is the length of the list even when such a situation is not guaranteed by the precondition. For example, the following relationship holds.

$$\mathit{traces}(\!(P_7 \mid \mathit{ls}(\mathsf{a}, \mathsf{x}, \mathsf{nil}))\!) \lesssim_{=_{\{\mathsf{a}\}}} \mathit{traces}(\!(P_8 \mid \exists \mathsf{a}.\ \mathit{ls}(\mathsf{a}, \mathsf{x}, \mathsf{nil}))\!)$$

Note that we are abstracting a program that assumes a is the length of the list by a program that only assumes there exists some length—the requirement that some program variable is storing the length is dropped in the precondition of P_8.

This use of non-determinism is an important component of the numeric abstraction technique that is the subject of Chapters 4 and 5.

3.3 Stuttering Equivalence and LTSL Properties

We now present some theorems relating stuttering equivalence and containment and satisfaction of LTSL properties.

Definition 25. *A state formula ς is E-**invariant** for an equivalence relation E iff*

$$\forall \gamma, \gamma'.\ \gamma \mathrel{E} \gamma' \Rightarrow \big((\gamma \models_X \varsigma) \Leftrightarrow (\gamma' \models_X \varsigma)\big)$$

*An LTSL formula ϕ is E-**invariant** iff all state formulae in ϕ are E-invariant. The set of E-invariant LTSL formulae is denoted LTSL^E.*

In the case of the a path formula containing the state formula Q, this definition above does not require that sub-formulas of Q be E-invariant. However, all examples of E-invariant state formulae that we will present in this thesis are composed of E-invariant sub-formulas.

Formulae that are E-invariant are preserved by E-stuttering equivalence.

Theorem 14. *If $\phi \in LTSL^E$ and $T \sim_E T'$ then $T \models_X \phi$ if and only if $T' \models_X \phi$.*

We first state an easy lemma, which follows directly from the definition of LTSL^E.

Lemma 9. *If $\phi \in LTSL^E$, then for all path formulae ϕ' such that ϕ' is a sub-formula of ϕ, we have $\phi' \in LTSL^E$.*

Proof. By the definition of LTSL^E (Definition 25) we have that all state formulae in ϕ are E-invariant. Since ϕ' is a sub-formula of ϕ, the set of state formulae appearing in ϕ' is a subset of those appearing in ϕ. Thus, all the state formulae in ϕ' are E-invariant and so $\phi' \in$ LTSL^E. □

We now turn to the proof of the theorem above (Theorem 14).

Proof. The proof is by induction on the structure of ϕ. We only consider the core connectives $\curlywedge$, $\sim$, and $\mathbf{U}$ as the other connectives are definable in terms of these (Theorem 9). We start with the base case, in which $\phi = \varsigma$ for some state formula ς.

CASE $\phi = \varsigma$: We first consider the forward direction of the "if and only if." Suppose $T \models_X \varsigma$. From the semantics in Figure 3.2 we have that $len(T) > 0$ and $T(0) \models_X \varsigma$. From our assumption that $T \sim_E T'$ we have $matches(T, T', \alpha, \beta, E)$ and, by the definition of matches, this gives us $0 < len(T) \Leftrightarrow 0 < len(T')$ and $T(0)\ E\ T'(0)$. Since we have $len(T) > 0$ this gives us $len(T') > 0$. From our assumption that $\phi \in$ LTSL^E and Definition 25 we then have

$$\gamma\, E\, \gamma' \Rightarrow \big((\gamma \models_X \varsigma) \Leftrightarrow (\gamma' \models_X \varsigma)\big)$$

Applying this to $T(0)\ E\ T'(0)$ we obtain $T(0) \models_X \varsigma \Leftrightarrow T'(0) \models_X \varsigma$. As $T(0) \models_X \varsigma$ is one of our assumptions, we then have $T'(0) \models_X \varsigma$, which, combined with $len(T') > 0$ gives us $T' \models_X \varsigma$. The backward direction is the same, except that T and T' are exchanged.

CASE $\phi = \phi_1 \curlywedge \phi_2$: We first consider the forward direction of the "if and only if." We assume $T \models_X \phi_1 \curlywedge \phi_2$. By the semantics of $\curlywedge$ we then have $T \models_X \phi_1$ and $T \models_X \phi_2$. By Lemma 9 and $\phi \in$ LTSL^E we have $\phi_1 \in$ LTSL^E and $\phi_2 \in$ LTSL^E. This allows us to apply the inductive hypothesis to each of these formulae yielding $T' \models_X \phi_1$ and $T' \models_X \phi_2$.

Again applying the semantics of $\dot{\wedge}$ we obtain $T' \models_X \phi_1 \wedge \phi_2$ which is our goal. The reverse implication is identical, but with T and T' exchanged.

CASE $\phi = {\sim}\phi_1$: We first consider the forward implication and assume $T \models_X {\sim}\phi_1$. The semantics of $\sim$ then give us that $T \not\models_X \phi_1$. The inductive hypothesis then gives us $T' \not\models_X \phi_1$ (since the conclusion of the theorem is an "if and only if"). From this, we apply the semantics of $\sim$ to obtain our goal: $T' \models_X {\sim}\phi_1$. The reverse implication is the same, but with T and T' exchanged.

CASE $\phi = \phi_1 \mathbf{U} \phi_2$: As before, Lemma 9 tells us that $\phi_1 \in \text{LTSL}^E$ and $\phi_2 \in \text{LTSL}^E$, which is one condition needed to apply the inductive hypothesis.

The following derivation establishes the forward direction of the implication. We start from the assumption that $T \models_X \phi_1 \mathbf{U} \phi_2$, which tells that there is some i satisfying the two initial assumptions below.

1	$0 \leq i < len(T) \wedge \big(T_i \models_X \phi_2\big)$	(Given)
2	$\forall j.\, 0 \leq j < i \Rightarrow \big(T_j \models_X \phi_1\big)$	(Given)
3	$T \sim_E T'$	(Given)
4	$T_i \sim_E T'_{f(i)}$	(Lemma 8 (for the f defined in that lemma))
5	$T'_{f(i)} \models_X \phi_2$	(Inductive Hypothesis: line 1 conjunct 2 and line 4)
6	$0 \leq j' < f(i)$	(Assumption)
7	$(T_{f^{-1}(j')}) \sim_E (T'_{j'})$	(Lemma 8 (f^{-1} defined in the Lemma))
8	$j' < f(i)$	(6)
9	$f^{-1}(j') < f^{-1}(f(i))$	(Lemma 8, monotonicity of f^{-1})
10	$f^{-1}(f(i)) \leq i$	(Lemma 8)
11	$f^{-1}(j') < i$	(9 and 10)
12	$T_{f^{-1}(j')} \models_X \phi_1$	(2 and 11)
13	$T'_{j'} \models_X \phi_1$	(Inductive Hyp: 7 and 12)
14	$\forall j'.\, 0 \leq j' < f(i) \Rightarrow \big(T'_{j'} \models_X \phi_1\big)$	($\forall$-intro, $\Rightarrow$-intro: 6 and 13)
15	$\exists i.\, T'_i \models_X \phi_2 \wedge \forall j'.\, 0 \leq j' < i \Rightarrow \big(T'_{j'} \models_X \phi_1\big)$	($\exists$-intro ($f(i) \rightarrow i$): 5 and 14)

16 $\quad T' \models_X \phi_1 \, \mathbf{U} \, \phi_2$ (Semantics of **U**)

As before, since $\sim_E$ is symmetric, the proof of the backward implication is the same as for the forward direction, but with T and T' exchanged. □

A corollary of Theorem 14 is that stuttering containment preserves satisfaction of LTSL^E properties in one direction.

Corollary 1. *If $\phi \in$ LTSL^E and S, S' are transition systems and traces$(S) \lesssim_E$ traces(S') then $S' \models_X \phi$ implies $S \models_X \phi$.*

Proof. This follows from the fact that LTSL formulae are interpreted universally over trace sets. Suppose $S' \models_X \phi$. By Definition 17 this implies

$$\forall T' \in \textit{traces}(S').\, T' \models_X \phi \tag{3.20}$$

That *traces*$(S) \lesssim_E$ *traces*(S') implies the following.

$$\forall T \in \textit{traces}(S).\, \exists T' \in \textit{traces}(S').\, T \sim_E T' \tag{3.21}$$

We now show $\forall T \in$ *traces*(S). $T \models_X \phi$, which implies $S \models_X \phi$ by Definition 17. Suppose $T \in$ *traces*(S). By (3.21) we have $\exists T' \in$ *traces*(S'). $T \sim_E T'$. Then by Theorem 14 and (3.20) we have $T \models_X \phi$, which is our goal. □

These results are not new. Analogous theorems are presented in [Clarke et al., 1999] and [Clarke and Schlingloff, 2001]. Here we have adapted these results to our particular formal setup, with separation logic formulae as the state formulae for the temporal logic and transitions systems arising from programs in our source language.

3.3.1 Syntactic Descriptions of E-invariance

The theorems above are stated in terms of E-invariant LTSL formulae, and the definition of E-invariance (Definition 25) is given in terms of the satisfaction relation $\models_X$ for LTSL formulae. However, we can also give syntactic restrictions that enforce E-invariance for the equality relations $=_V$ and $\overset{s}{=}_V$. These syntactic restrictions are much easier to check than the semantic properties used in Definition 25.

Syntactic Description of $=_V$-invariance

Definition 26. *Let LTSL(V) be the set of LTSL formulae with free variables contained in the set V.*

Theorem 15. *If $\phi \in$ LTSL(V) then ϕ is $=_V$-invariant.*

Proof. We must show that if the free variables of ϕ are contained in V, then all state formulae ς which are subterms of ϕ have the following property.

$$\forall \gamma, \gamma'. \left(\gamma =_V \gamma'\right) \Rightarrow \left((\gamma \models_X \varsigma) \Leftrightarrow (\gamma' \models_X \varsigma)\right)$$

We first note that if the free variables of ϕ are contained in V and ς is a subterm of ϕ, then the free variables of ς are contained in V. We now consider an arbitrary γ, γ' such that $\gamma =_V \gamma'$ and show that $(\gamma \models_X \varsigma) \Leftrightarrow (\gamma' \models_X \varsigma)$. The proof is by case analysis on the state formula ς.

CASE $\varsigma = \mathit{err}$: That $\gamma \models_X \mathit{err}$ holds implies $\gamma = \mathbf{error}$. The relation $\gamma =_V \gamma'$ then implies $\gamma' = \mathbf{error}$ which implies $\gamma' \models_X \mathit{err}$. The reverse direction is identical with γ and γ' exchanged.

CASE $\varsigma = \mathit{final}$: That $\gamma \models_X \mathit{final}$ holds implies $\gamma = \mathbf{final}(s, h)$ for some s, h. The relation $\gamma =_V \gamma'$ then implies $\gamma' = \mathbf{final}(s', h)$ where $s =_V s'$. This implies $\gamma' \models_X \mathit{final}$. The reverse direction is the same with γ and γ' exchanged.

CASE $\varsigma = \mathit{atloc}(l)$: That $\gamma \models_X \mathit{atloc}(l)$ holds implies $\gamma = \mathbf{goto}(l, (s, h))$. The relation $\gamma =_V \gamma'$ then implies $\gamma' = \mathbf{goto}(l, (s', h))$ with $s =_V s'$. This implies $\gamma' \models_X \mathit{atloc}(l)$. The reverse direction is the same with γ and γ' exchanged.

CASE $\varsigma = Q$: That $\gamma \models_X Q$ holds implies $\gamma = \langle k, (s, h)\rangle$ or $\gamma = \mathbf{goto}(l, (s, h))$ or $\gamma = \mathbf{final}(s, h)$ and in each case $(s, h) \models_X Q$. We will consider the $\gamma = \langle k, (s, h)\rangle$ case. The others are similar. We have $\gamma =_V \gamma'$ which implies that $\gamma' = \langle k', (s', h)\rangle$ where

$s =_V s'$. By Lemma 4 and the fact that $fv(Q) \subseteq V$ we then have $(s', h) \models_X Q$. This implies $\langle k', (s', h)\rangle \models_X Q$ according to the semantics given in Figure 3.2. □

Next, we have a similar result for $\overset{\mathrm{s}}{=}_V$.

Syntactic Description of $\overset{\mathrm{s}}{=}_V$-invariance

Definition 27. *Let LTSLP(V) be the set of pure LTSL formulae with free variables in V. These are LTSL(V) formulae that do not contain subterms that are in the grammar for spatial predicates given in Figure 2.6. That is, they do not contain subterms of the form* $\mathbf{emp}$, $e^{\mathrm{a}} \mapsto [\rho]$, *or* $p^{\vec{\tau}}(\vec{e}^{\,\vec{\tau}})$.

Theorem 16. *If $\phi \in$ LTSLP(V) then ϕ is $\overset{\mathrm{s}}{=}_V$-invariant.*

Proof. The proof is similar to the proof for $=_V$ above. We must show that if $\phi \in$ LTSLP(V) then for all state formulae ς which are sub-formulae of ϕ, we have

$$\forall \gamma, \gamma'.\ (\gamma \overset{\mathrm{s}}{=}_V \gamma') \Rightarrow \big(\gamma \models_X \varsigma\big) \Leftrightarrow \big(\gamma' \models_X \varsigma\big) \tag{3.22}$$

The formula ς must have the form *final*, *err*, $atloc(l)$, or Q. The first three cases are identical to the corresponding cases in the proof of Theorem 15 above. For $\varsigma = Q$, we have that Q is pure since Q is a sub-formula of ϕ and $\phi \in$ LTSLP(V). Given the semantics of $\gamma \models_X \varsigma$ in the case where $\varsigma = Q$, showing condition (3.22) reduces to showing the following.

$$\text{if } Q \text{ is pure then } (s \overset{\mathrm{s}}{=}_V s') \Rightarrow \forall h, h'.\ \big((s, h) \models_X Q\big) \Leftrightarrow \big((s', h') \models_X Q\big)$$

We show this by induction on Q, recalling that since Q is pure, the base cases $Q = \mathbf{emp}$, $Q = e^{\mathrm{a}} \mapsto [\rho]$ and $Q = p^{\vec{\tau}}(\vec{e}^{\,\vec{\tau}})$ need not be considered.

CASE $Q = e^{\mathrm{b}}$: In this case, the semantics of Q is independent of the heap. The definition of $\models_X$ from Figure 2.7 tells us that $(s, h) \models_X Q$ iff $[\![e^{\mathrm{b}}]\!]\, s =$ ***true***. By Lemma 1 we have that $[\![e^{\mathrm{b}}]\!]\, s = [\![e^{\mathrm{b}}]\!]\, s$, which implies $(s, h) \models_X Q$ iff $(s', h') \models_X Q$.

CASE $Q = Q_1 * Q_2$: We have $(s, h) \models_X Q_1 * Q_2$ iff there exist h_1, h_2 such that $dom(h_1) \cap dom(h_2) = \emptyset$ and $h = h_1 \cap h_2$ and $(s, h_1) \models_X Q_1$ and $(s, h_2) \models_X Q_2$.

That $\mathit{fv}(Q) \subseteq V$ implies $\mathit{fv}(Q_1) \subseteq V$ and $\mathit{fv}(Q_2) \subseteq V$. This allows us to apply the induction hypothesis.

But we must first determine how to split the heap. We wish to show $(s', h') \models_X Q_1 {*} Q_2$ for an arbitrary h'. To do this, we must show that there exists h'_1, h'_2 such that $\mathit{dom}(h'_1) \cap \mathit{dom}(h'_2) = \emptyset$ and $h' = h'_1 \cup h'_2$ and $(s', h'_1) \models_X Q_1$ and $(s', h'_2) \models_X Q_2$. We let $h'_1 = h'$ and let $h'_2 = \{\}$. Clearly $\mathit{dom}(h'_1) \cap \mathit{dom}(h'_2) = \emptyset$ and $h' = h'_1 \cup h'_2$. Our inductive hypothesis tells us that since $(s, h) \models_X Q_1$, we can conclude $(s', h'_1) \models_X Q_1$ and similarly for Q_2. This completes the proof.

CASE $Q = Q_1 \wedge Q_2$: We have $(s, h) \models_X Q_1 \wedge Q_2$ iff $(s, h) \models_X Q_1$ and $(s, h) \models_X Q_2$. Again, $\mathit{fv}(Q) \subseteq V$ implies $\mathit{fv}(Q_1) \subseteq V$ and $\mathit{fv}(Q_2) \subseteq V$, allowing us to apply the inductive hypothesis to $(s, h) \models_X Q_1$, obtaining $(s', h') \models_X Q_1$ for an arbitrary h' (and similarly for $(s', h') \models_X Q_2$). This implies our result.

CASE $Q = Q_1 \vee Q_2$: This case is very similar to the $*$ and $\wedge$ cases. We have $(s, h) \models_X Q_1 \vee Q_2$ iff $(s, h) \models_X Q_1$ or $(s, h) \models_X Q_2$. In either case, we have $\mathit{fv}(Q_i) \subseteq V$ and apply our inductive hypothesis to obtain $(s, h) \models_X Q_i$ iff $(s', h') \models_X Q_i$ for an arbitrary h', which lets us conclude that $(s, h) \models_X Q$ iff $(s', h) \models_X Q$.

CASE $Q = (Q_1 \Rightarrow Q_2)$: We will consider the forward direction first and show that for all h' we have $(s, h) \models_X (Q_1 \Rightarrow Q_2)$ implies $(s', h') \models_X (Q_1 \Rightarrow Q_2)$. Suppose $(s, h) \models_X (Q_1 \Rightarrow Q_2)$. Then by the definition of $\models_X$ given in Figure 2.7 we have $(s, h) \models_X Q_1$ implies $(s, h) \models_X Q_2$. Now, suppose $(s', h') \models_X Q_1$. Since $\mathit{fv}(Q) = \mathit{fv}(Q_1) \cup \mathit{fv}(Q_2)$ and $\mathit{fv}(Q) \subseteq V$, we have $\mathit{fv}(Q_1) \subseteq V$ and $\mathit{fv}(Q_2) \subseteq V$. This lets us apply our inductive hypothesis, obtaining $(s, h) \models_X Q_1$. This implies $(s, h) \models_X Q_2$ by our assumption, which, applying the inductive hypothesis again, gives us $(s', h') \models_X Q_2$. Thus, we have shown that $(s', h') \models_X Q_1$ implies $(s', h') \models_X Q_2$, which lets us conclude $(s', h') \models_X (Q_1 \Rightarrow Q_2)$. The proof of the backwards direction is symmetric, with s and s' interchanged.

CASE $Q = \exists x.\, Q'$: We consider the forward direction first. The relation $(s, h) \models_X \exists x.\, Q$ implies there exists a v such that $(s[x \to v], h) \models_X Q'$. Consider the store $s'[x \to v]$. Since $s =_V s'$, we have $s[x \to v] =_{V \cup \{x\}} s'[x \to v]$. We have that $\mathit{fv}(Q) = \mathit{fv}(Q') - \{x\}$

$P \stackrel{\text{def}}{=}$

```
L$_0$ : goto L$_1$;
L$_1$ : branch $x \neq$ nil ⇒
                t := x;
                x := x.next;
                free t;
                goto L$_1$,
            x = nil ⇒ halt
    end
```

$P' \stackrel{\text{def}}{=}$

```
L$_0$ : goto L$_1$;
L$_1$ : branch a $> 0$ ⇒
                t := x;
                x := x.next;
                free t;
                a := a $- 1$;
                goto L$_1$,
            a $= 0$ ⇒ halt
    end
```

Figure 3.6: Two programs with traces related by $\approx_{=_{\{\mathsf{x},\mathsf{t}\}}}$

and $fv(Q) \subseteq V$ which implies $fv(Q') \subseteq V \cup \{x\}$. We can then apply our inductive hypothesis to $(s[x \to v], h) \models_X Q'$, obtaining $(s'[x \to v], h') \models_X Q'$ for an arbitrary h'. This implies $(s', h') \models_X \exists x.\ Q'$. The backward direction is symmetric, with s and s' interchanged.

CASE $Q = \forall x.\ Q$: We consider the forward direction first. Let h' be an arbitrary heap. The relation $(s, h) \models_X \forall x.\ Q$ implies that for all v we have $(s[x \to v], h) \models_X Q'$. Consider an arbitrary v'. Instantiating v above with v' we have $(s[x \to v'], h) \models_X Q'$. Since $s =_V s'$, we have $s[x \to v] =_{V \cup \{x\}} s'[x \to v]$. We have that $fv(Q) = fv(Q') - \{x\}$ and $fv(Q) \subseteq V$ which implies $fv(Q') \subseteq V \cup \{x\}$. We can then apply our inductive hypothesis to $(s[x \to v'], h) \models_X Q'$, obtaining $(s'[x \to v'], h') \models_X Q'$. Since v' was arbitrary, we conclude that for all v' we have $(s'[x \to v'], h') \models_X Q'$, which implies $(s', h') \models_X \forall x.\ Q'$. The backward direction is symmetric, with s and s' interchanged. □

3.3.2 Translating Results Obtained By Analyzing Abstractions

Corollary 1 stated the connection between E-stuttering trace containment and E-invariant LTL\X properties. Given programs P and P' and preconditions Q and Q' such that *traces*$(\!(P \mid Q)\!) \lesssim_E$ *traces*$(\!(P' \mid Q')\!)$, this allows us to take a property ϕ, which we would like to check for $(\!(P \mid Q)\!)$ and instead check that it holds of $(\!(P' \mid Q')\!)$. For example, in

Figure 3.6 we give two programs satisfying the following.

$$\mathit{traces}(\!(P \mid \mathit{ls}(\mathsf{a},\mathsf{x},\mathsf{nil}))\!) \approx_{=\{\mathsf{x},\mathsf{t}\}} \mathit{traces}(\!(P' \mid \mathit{ls}(\mathsf{a},\mathsf{x},\mathsf{nil}))\!)$$

Suppose we want to show that P terminates. Termination corresponds to the LTSL property $\mathbf{F}(\mathit{final})$. We can check that this property holds of P', which it does since variable a decreases during each iteration and is bounded below by 0. This then implies that P satisfies $\mathbf{F}(\mathit{final})$ and thus P also terminates.

This approach, of stating a property of the original program and then proving it holds of the abstraction, naturally leads one to consider properties stated over the free variables of the original program. However, it can also be useful to consider properties involving the variables that occur in the abstraction, but not in the original program (a is an example of such a variable in P'). We could ask a static analysis to analyze P' and return an invariant that holds at L_1. Such an invariant may involve variables in P' that are not in P and thus the property may not hold of P. For example, the property $\mathbf{G}(\mathit{atloc}(\mathsf{L}_1) \supset \mathit{ls}(\mathsf{a},\mathsf{x},\mathsf{nil}))$ holds of $(\!(P' \mid \mathit{ls}(\mathsf{a},\mathsf{x},\mathsf{nil}))\!)$. However, since the variable a is not updated by P, this property does not hold of P, even when started from the same set of initial states.

We can, however, translate the property that holds of P' to a property that holds of P by accounting for the fact that the variable a is not updated by P. By existentially quantifying a, we capture the fact that there is a value of a that makes the property true, without requiring a to actually be updated with the appropriate value. The property that holds of P then becomes $\mathbf{G}(\mathit{atloc}(\mathsf{L}_1) \supset \exists \mathsf{a}.\ \mathit{ls}(\mathsf{a},\mathsf{x},\mathsf{nil}))$.

This mode of reasoning is captured by the following theorem, which allows us to relate properties of P' to properties of P even when P' includes variables not present in P. First we define a function $\boxed{\exists}(V,\phi)$ which existentially quantifies the variables in V in all state formulae. We write $\exists V.\ Q$ where V is a finite set of variables to represent the existential quantification of all variables in V (that is, $(\exists V.\ Q) = (\exists v_1, v_2, \ldots, v_n.\ Q)$ if $V = \{v_1, v_2, \ldots, v_n\}$).

Definition 28. *Let V be a finite set of variables. Then $\boxed{\exists}(V,\phi)$ and $\boxed{\forall}(V,\phi)$ are defined via mutual induction as given in Figure 3.7.*

$$
\begin{array}{rcl|rcl}
\boxed{\exists}(V, \varsigma) & = & \begin{cases} \exists V.\, Q & \text{if } \varsigma = Q \\ & \quad \text{for some } Q \\ \varsigma & \text{otherwise} \end{cases} & \boxed{\forall}(V, \varsigma) & = & \begin{cases} \forall V.\, Q & \text{if } \varsigma = Q \\ & \quad \text{for some } Q \\ \varsigma & \text{otherwise} \end{cases} \\
\boxed{\exists}(V, \phi_1 \curlywedge \phi_2) & = & \big(\boxed{\exists}(V, \phi_1)\big) \curlywedge \big(\boxed{\exists}(V, \phi_2)\big) & \boxed{\forall}(V, \phi_1 \curlywedge \phi_2) & = & \big(\boxed{\forall}(V, \phi_1)\big) \curlywedge \big(\boxed{\forall}(V, \phi_2)\big) \\
\boxed{\exists}(V, \phi_1 \curlyvee \phi_2) & = & \big(\boxed{\exists}(V, \phi_1)\big) \curlyvee \big(\boxed{\exists}(V, \phi_2)\big) & \boxed{\forall}(V, \phi_1 \curlyvee \phi_2) & = & \big(\boxed{\forall}(V, \phi_1)\big) \curlyvee \big(\boxed{\forall}(V, \phi_2)\big) \\
\boxed{\exists}(V, {\sim}\phi) & = & {\sim}\big(\boxed{\forall}(V, \phi)\big) & \boxed{\forall}(V, {\sim}\phi) & = & {\sim}\big(\boxed{\exists}(V, \phi)\big) \\
\boxed{\exists}(V, \mathbf{G}\phi) & = & \mathbf{G}\big(\boxed{\exists}(V, \phi)\big) & \boxed{\forall}(V, \mathbf{G}\phi) & = & \mathbf{G}\big(\boxed{\forall}(V, \phi)\big) \\
\boxed{\exists}(V, \mathbf{F}\phi) & = & \mathbf{F}\big(\boxed{\exists}(V, \phi)\big) & \boxed{\forall}(V, \mathbf{F}\phi) & = & \mathbf{F}\big(\boxed{\forall}(V, \phi)\big) \\
\boxed{\exists}(V, \phi_1 \,\mathbf{U}\, \phi_2) & = & \big(\boxed{\exists}(V, \phi_1)\big) \,\mathbf{U}\, \big(\boxed{\exists}(V, \phi)\big) & \boxed{\forall}(V, \phi_1 \,\mathbf{U}\, \phi_2) & = & \big(\boxed{\forall}(V, \phi_1)\big) \,\mathbf{U}\, \big(\boxed{\forall}(V, \phi)\big)
\end{array}
$$

Figure 3.7: Definition of $\boxed{\exists}$ and $\boxed{\forall}$.

Theorem 17. *Suppose $T \sim_{=_V} T'$ and let $V' = fv(\phi) - V$. Then $T' \models_X \phi$ implies $T \models_X \boxed{\exists}(V', \phi)$ and $T' \not\models_X \phi$ implies $T \not\models_X \boxed{\forall}(V', \phi)$.*

Corollary 2. *Let $V' = fv(\phi) - V$. If $traces(\!(P \mid Q)\!) \lesssim_{=_V} traces(\!(P' \mid Q')\!)$ and $(\!(P' \mid Q')\!) \models_X \phi$ then $(\!(P \mid Q)\!) \models_X \boxed{\exists}(V', \phi)$.*

To the best of our knowledge, this theorem has not been stated before, perhaps because most of the work on LTL\X makes minimal assumptions about the language of state formulae; in particular, existential and universal quantification are not assumed to be present.

Before we proceed with the proof, we first establish the following lemma.

Lemma 10. *If $len(T') > 0$ and $T \sim_{=_V} T'$ then $len(T) > 0$ and $T(0) =_V T'(0)$.*

Proof. The conditions $len(T) > 0$ and $len(T') > 0$ are required for $T(0)$ and $T'(0)$ to be defined. The proof proceeds as follows.

1 $T \sim_{=_V} T'$ (Given)

2 $len(T') > 0$ (Given)

3 $\exists \alpha, \beta.\ matches(T, T', \alpha, \beta, =_V)$ (Def. of $\sim_E$ (Def. 19))

4 $matches(T, T', \alpha, \beta, =_V)$ ($\exists$-elim)

$$\begin{gathered}
\boxed{\exists}(V, \phi_1 \curlyvee \phi_2) = \\
\boxed{\exists}(V, \sim(\sim\phi_1 \curlywedge \sim\phi_2)) = \\
\sim(\boxed{\forall}(V, \sim\phi_1 \curlywedge \sim\phi_2)) = \\
\sim\big(\boxed{\forall}(V, \sim\phi_1) \curlywedge \boxed{\forall}(V, \sim\phi_2)\big) = \\
\sim\big(\sim(\boxed{\exists}(V, \phi_1)) \curlywedge \sim(\boxed{\exists}(V, \phi_2))\big) = \\
\big(\boxed{\exists}(V, \phi_1)\big) \curlyvee \big(\boxed{\exists}(V, \phi_2)\big)
\end{gathered}$$

$$\begin{gathered}
\boxed{\exists}(V, \mathbf{F}\phi) = \\
\boxed{\exists}(V, \mathsf{true}\ \mathbf{U}\ \phi) = \\
\big(\boxed{\exists}(V, \mathsf{true})\big)\ \mathbf{U}\ \big(\boxed{\exists}(V, \phi)\big) = \\
\mathsf{true}\ \mathbf{U}\ \big(\boxed{\exists}(V, \phi)\big) = \\
\mathbf{F}(\boxed{\exists}(V, \phi))
\end{gathered}$$

$$\begin{gathered}
\boxed{\exists}(V, \mathbf{G}\phi) = \\
\boxed{\exists}(V, \sim(\mathbf{F}(\sim\phi))) = \\
\sim(\boxed{\forall}(V, \mathbf{F}(\sim\phi))) = \\
\sim(\mathbf{F}(\boxed{\forall}(V, \sim\phi))) = \\
\sim\big(\mathbf{F}(\sim(\boxed{\exists}(V, \phi)))\big) = \\
\mathbf{G}\big(\boxed{\exists}(V, \phi)\big)
\end{gathered}$$

Figure 3.8: Derivations showing that our definition of $\boxed{\exists}$ is consistent with the rewritings given in Theorem 9. The corresponding derivations for $\boxed{\forall}$ are identical, with the symbols $\boxed{\exists}$ and $\boxed{\forall}$ interchanged.

5 $\alpha(0) = \beta(0) = 0$ (Def. of $matches$ (Def. 18))

6 $\alpha(0) \leq 0 < \alpha(1) \wedge \beta(0) \leq 0 < \beta(1)$ (Above and α, β strictly increasing)

7 $\forall i, j, k.\ \big(\alpha(i) \leq j < \alpha(i+1)\big) \wedge \big(\beta(i) \leq k < \beta(i+1)\big) \Rightarrow$
$\big(\mathit{len}(T) > 0 \Leftrightarrow \mathit{len}(T') > 0\big) \wedge \big(T(j) =_V T'(k)\big)$ (Def. of $matches$)

8 $\big(\mathit{len}(T) > 0 \Leftrightarrow \mathit{len}(T') > 0\big) \wedge \big(T(0) =_V T'(0)\big)$ ($\Rightarrow$-elim: above two lines)

□

We now present the proof of Theorem 17. We will only consider the core connectives $\sim$, $\curlywedge$, and $\mathbf{U}$. To justify this simplification, we must show that Definition 28 is consistent with the encoding of $\curlyvee, \mathbf{F}$, and $\mathbf{G}$ in terms of these core connectives. This is demonstrated by the derivations in Figure 3.8, where we first translate a formula into its core representation as given by Theorem 9, then apply the definition of $\boxed{\exists}$, then rewrite the result according to Theorem 9. The formula we obtain in the end should be the same as that given by Definition 28. The corresponding derivations for $\boxed{\forall}$ are identical, with the symbols $\boxed{\exists}$ and $\boxed{\forall}$ interchanged.

Proof. (of Theorem 17) The proof is by induction on the formula ϕ. We have the following assumptions.

$$T \sim_{=_V} T' \tag{3.23}$$

$$V' = \mathit{fv}(\phi) - V \tag{3.24}$$

And we wish to show

$$T' \models_X \phi \text{ implies } T \models_X \boxed{\exists}(V', \phi)$$

and

$$T' \not\models_X \phi \text{ implies } T \not\models_X \boxed{\forall}(V', \phi)$$

Base Cases

We now consider the $\boxed{\exists}$ conjunct for the first three base cases, which are as follows.

$$\begin{aligned} \phi &= \mathit{atloc}(l) \\ \phi &= \mathit{err} \\ \phi &= \mathit{final} \end{aligned}$$

These are all proved in the same way. We present derivations for each base case, but they all have the same structure. The final base case, $\phi = Q$, is presented last and the structure of the proof is different in that case.

CASE $\phi = \mathit{atloc}(l)$:

1 $T' \models_X \mathit{atloc}(l)$ (Given)

2 $\mathit{len}(T') > 0 \wedge \big(T'(0) \models_X \mathit{atloc}(l)\big)$

(Def. of $\models_X$ relation for path formulae (Figure 3.2))

3 $\exists s, h.\, T'(0) = \mathbf{goto}(l, (s, h))$

(Def. of $\models_X$ relation for state formulae (Figure 3.2))

4 $T'(0) = \mathbf{goto}(l, (s, h))$ ($\exists$-elim)

5 $\mathit{len}(T) > 0 \wedge \big(T(0) =_V T'(0)\big)$

(Lemma 10: assumption (3.23) and line 2 conjunct 1)

6 $T(0) =_V \mathbf{goto}(l, (s, h))$ (Above and line 4)

7 $\exists s'.\, T(0) = \mathbf{goto}(l, (s', h)) \wedge s =_V s'$ (Def. of $=_V$ (Def. 23))

8 $T(0) \models_X \mathit{atloc}(l)$ (Def. of $\models_X$ (for state formulae))

9 $T \models_X \mathit{atloc}(l)$ (Def. of $\models_X$ (for path formulae): above and line 5 conjunct 1)

10 $T \models_X \boxed{\exists}(V', \mathit{atloc}(l))$ (Def. of $\boxed{\exists}$ (Def. 28))

CASE $\phi = \mathit{err}$:

1 $T' \models_X \mathit{err}$ (Given)

2 $\mathit{len}(T') > 0 \wedge \big(T'(0) \models_X \mathit{err}\big)$ (Def. of $\models_X$ relation (Figure 3.2))

3 $T'(0) = \mathbf{error}$ (Def. of $\models_X$ relation (Figure 3.2))

4 $\mathit{len}(T) > 0 \wedge \big(T(0) =_V T'(0)\big)$

(Lemma 10: assumption (3.23) and line 2 conjunct 1)

5 $T(0) =_V \mathbf{error}$ (Above and line 3)

6 $T(0) = \mathbf{error}$ (Def. of $=_V$ (Def. 23))

7 $T(0) \models_X \mathit{err}$ (Def. of $\models_X$ (for state formulae))

8 $T \models_X \mathit{err}$ (Def. of $\models_X$ (for path formulae): above and line 4 conjunct 1)

9 $T \models_X \boxed{\exists}(V', \mathit{err})$ (Def. of $\boxed{\exists}$ (Def. 28))

CASE $\phi = \mathit{final}$:

1 $T' \models_X \mathit{final}$ (Given)

2 $\mathit{len}(T') > 0 \wedge T'(0) \models_X \mathit{final}$ (Def. of $\models_X$ relation (Figure 3.2))

3 $\exists s, h.\, T'(0) = \mathbf{final}(s, h)$ (Def. of $\models_X$ relation (Figure 3.2))

4 $T'(0) = \mathbf{final}(s, h)$ ($\exists$-elim)

5 $\mathit{len}(T) > 0 \wedge \big(T(0) =_V T'(0)\big)$

(Lemma 10: assumption (3.23) and line 4 conjunct 1)

6 $T(0) =_V \mathbf{final}(s, h)$ (Above and line 5)

7 $\exists s'.\, T(0) = \mathbf{final}(s', h) \wedge s =_V s'$ (Def. of $=_V$ (Def. 23))

8 $T(0) \models_X \mathit{final}$ (Def. of $\models_X$ (for state formulae))

9 $T \models_X \mathit{final}$ (Def. of $\models_X$ (for path formulae): above and line 7 conjunct 1)

10 $T \models_X \boxed{\exists}(V', \mathit{final})$ (Def. of $\boxed{\exists}$ (Def. 28))

CASE $\phi = Q$: We have that $T' \models_X Q$ and want to show that $T \models_X \boxed{\exists}(V', Q)$. The definition of $\models_X$ states that our assumption $T' \models_X Q$ implies $\mathit{len}(T') > 0 \wedge T'(0) \models_X Q$. We also have the assumption $T \sim_{=_V} T'$ which, by Lemma 10, implies $\mathit{len}(T) > 0$ and $T(0) =_V T'(0)$. We have by the definition of $\boxed{\exists}$ (Definition 28) that $\boxed{\exists}(V', Q) = \exists V'.\, Q$ and from the definition of $\models_X$ we have that $T \models_X \exists V'.\, Q$ iff $\mathit{len}(T) > 0$ and $T(0) \models_X \exists V'.\, Q$. Thus, our goal reduces to showing that $T(0) \models_X \exists V'.\, Q$ based on the assumptions $T'(0) \models_X Q$ and $T(0) =_V T'(0)$.

We now case split on the form of $T'(0)$. Based on the semantics of LTSL in Figure 3.2 and $T'(0) \models_X Q$ we have that $T'(0)$ either has the form $\langle k, (s, h)\rangle$, or $\mathbf{goto}(l, (s, h))$, or $\mathbf{final}(s, h)$ and that whichever case holds, we have $(s, h) \models_X Q$. All the cases are proved in the same way, so we will only show $\langle k, (s, h)\rangle$ here.

We have from $T(0) =_V T'(0)$ and $T'(0) = \langle k, (s, h)\rangle$ that $T(0) = \langle k', (s', h)\rangle$ for some s' such that $s' =_V s$. We want to show $(s', h) \models_X \exists V'.\, Q$, which will hold if we can give some s'' that differs from s' only on the values of variables in V' and for which $(s'', h) \models_X Q$ holds. The needed s'' is defined as follows.

$$s''(x) = \begin{cases} s'(x) & \text{if } x \notin V' \\ s(x) & \text{if } x \in V' \end{cases}$$

Clearly this s'' differs from s' only in the values of variables in V'. We will show that $(s'', h) \models_X Q$ by applying Lemma 4 to our assumption that $(s, h) \models_X Q$. In order to apply this lemma, we must show that $s =_{\mathit{fv}(Q)} s''$. To do this, we consider an arbitrary variable x and show that if $x \in \mathit{fv}(Q)$ then $s(x) = s''(x)$. From $V' = \mathit{fv}(Q) - V$ and

$x \in \mathit{fv}(Q)$, we have that either $x \in V'$ or $x \in V$. If $x \in V'$ then we have $s''(x) = s(x)$ (our goal) from the definition of s''. If $x \in V$ then we have from $s =_V s'$ that $s(x) = s'(x)$. Then, from the definition of s'' we have that either $s''(x) = s(x)$ (in which case we have attained our goal) or $s''(x) = s'(x)$, in which case transitivity of equality with $s(x) = s'(x)$ gives us $s(x) = s''(x)$. Thus, we have $s =_{\mathit{fv}(Q)} s''$ and can apply Lemma 4 obtaining our goal of $(s'', h) \models_X Q$ and completing the proof of this case.

We now show the base cases for the $\boxed{\forall}$ conjunct. They are similar to the $\boxed{\exists}$ cases except that since our assumption involves the $\models_X$ relation *not* holding, there is some disjunction involved. In particular, a trace can fail to satisfy a state formula either by being empty or by being non-empty with a first state that is not of the appropriate form. This is demonstrated by the following derivation.

1 $\quad T' \not\models_X \varsigma$ (Given)

2 $\quad \neg(\mathit{len}(T') > 0 \wedge \big(T'(0) \models_X \varsigma\big))$ (Def. of $\models_X$)

3 $\quad \neg(\mathit{len}(T') > 0) \vee \big(T'(0) \not\models_X \varsigma\big)$ (Boolean Reasoning)

The empty cases are all handled uniformly. We show the derivation for these below.

1 $\quad \neg(\mathit{len}(T') > 0)$ (Given)

2 $\quad \exists \alpha, \beta.\ \mathit{matches}(T, T', \alpha, \beta, =_V)$ (Assumption (3.23) and Def. of $\sim_E$ (Def. 19))

3 $\quad \mathit{matches}(T, T', \alpha, \beta, =_V)$ ($\exists$-elim)

4 $\quad \alpha(0) = \beta(0) = 0$ (Def. of $\mathit{matches}$ (Def. 18))

5 $\quad \alpha(0) \leq 0 < \alpha(1) \wedge \beta(0) \leq 0 < \beta(1)$ (Above and α, β strictly increasing)

6 $\quad \forall i, j, k.\ \big(\alpha(i) \leq j < \alpha(i+1)\big) \wedge \big(\beta(i) \leq k < \beta(i+1)\big) \Rightarrow$
$\qquad j < \mathit{len}(T) \Leftrightarrow k < \mathit{len}(T')$ (Def. of $\mathit{matches}$)

7 $\quad \big(\alpha(0) \leq 0 < \alpha(1)\big) \wedge \big(\beta(0) \leq 0 < \beta(1)\big) \Rightarrow$
$\qquad 0 < \mathit{len}(T) \Leftrightarrow 0 < \mathit{len}(T')$ ($\forall$-elim, $i, j, k = 0$)

8 $\quad 0 < \mathit{len}(T) \Leftrightarrow 0 < \mathit{len}(T')$ ($\Rightarrow$-elim: above and line 5)

9 $\neg(len(T) > 0)$ (Above and line 1)

10 $T \not\models_X \boxed{\forall}(V', \varsigma)$ (Def of $\models_X$ for path formulae)

This leaves us with the task of showing that $T'(0) \not\models_X \varsigma$ implies $T(0) \not\models_X \boxed{\forall}(V', \varsigma)$ under the assumption that $len(T') > 0$ and $len(T) > 0$. As before, Lemma 10 gives us that $T(0) =_V T'(0)$. We consider each base case, starting with $\varsigma = err$.

CASE $\varsigma =$ *err*:

1 $len(T) > 0$ (Given)

2 $len(T') > 0$ (Given)

3 $T(0) =_V T'(0)$ (Given)

4 $T'(0) \not\models_X$ *err* (Given)

5 $T'(0) \neq \mathbf{error}$ (Def. of $\models_X$)

6 $(T'(0) = \mathbf{final}(s, h)) \vee (T'(0) = \mathbf{goto}(l, (s, h))) \vee (T'(0) = \langle k, (s, h)\rangle)$
(Case analysis)

At this point, the reasoning is the same for each disjunct. We show $T'(0) = \mathbf{final}(s, h)$ as an example.

7 $T'(0) = \mathbf{final}(s, h)$ (Given)

8 $T(0) =_V \mathbf{final}(s, h)$ (Above and line 3)

9 $T(0) = \mathbf{final}(s', h) \wedge s' =_V s$ (Def. of $=_V$ (Def. 23))

10 $T(0) \neq \mathbf{error}$ (Def. of $=$ (syntactic equality))

11 $T(0) \not\models_X$ *err* (Def. of $\models_X$ for state formulae)

12 $T(0) \not\models_X \boxed{\forall}(V', \mathit{err})$ (Def. of $\boxed{\forall}$)

CASE $\varsigma =$ *final*:

1 $len(T) > 0$ (Given)

2 $len(T') > 0$ (Given)

3 $T(0) =_V T'(0)$ (Given)

4 $T'(0) \not\models_X \mathit{final}$ (Given)

5 $\forall s, h.\, T'(0) \neq \mathbf{final}(s, h)$ (Def. of $\models_X$)

We now begin a proof by contradiction aimed at showing that $T(0) \neq \mathbf{final}(s, h)$ for all s, h.

6 $T(0) = \mathbf{final}(s', h')$ (Assumption)

7 $\mathbf{final}(s', h') =_V T'(0)$ (Above and line 3)

8 $T'(0) = \mathbf{final}(s'', h') \wedge s'' =_V s'$ (Def. of $=_V$)

9 $T'(0) \neq \mathbf{final}(s'', h')$ ($\forall$-elim, line 5)

10 false (Previous two lines)

11 $(T(0) = \mathbf{final}(s', h')) \Rightarrow \mathsf{false}$ ($\Rightarrow$-intro line 5 and above)

12 $T(0) \neq \mathbf{final}(s', h')$ (Boolean reasoning)

13 $\forall s', h'.\, T(0) \neq \mathbf{final}(s', h')$ ($\forall$-intro)

14 $T(0) \not\models_X \mathit{final}$ (Def. of $\models_X$ for state formulae)

15 $T(0) \not\models_X \boxed{\forall}(V', \mathit{final})$ (Def. of $\boxed{\forall}$)

CASE $\varsigma = \mathit{atloc}(l)$:

1 $\mathit{len}(T) > 0$ (Given)

2 $\mathit{len}(T') > 0$ (Given)

3 $T(0) =_V T'(0)$ (Given)

4 $T'(0) \not\models_X \mathit{atloc}(l)$ (Given)

5 $\forall s, h.\, T'(0) \neq \mathbf{goto}(l, (s, h))$ (Def. of $\models_X$)

We now begin a proof by contradiction aimed at showing that $T(0) \neq \mathbf{final}(s, h)$ for all s, h

6 $T(0) = \mathbf{goto}(l, (s', h'))$ (Assumption)

7 $\mathbf{final}(s', h') =_V T'(0)$ (Above and line 3)

8 $T'(0) = \mathbf{goto}(l, (s'', h')) \wedge s'' =_V s'$ (Def. of $=_V$)

9 $T'(0) \neq \mathbf{goto}(l, (s'', h'))$ ($\forall$-elim, line 5)

10 false (Previous two lines)

11 $(T(0) = \mathbf{goto}(l, (s', h'))) \Rightarrow \mathsf{false}$ ($\Rightarrow$-intro line 5 and above)

12 $T(0) \neq \mathbf{goto}(l, (s', h'))$ (Boolean reasoning)

13 $\forall s', h'.\, T(0) \neq \mathbf{goto}(l, (s', h'))$ ($\forall$-intro)

14 $T(0) \not\models_X \mathit{atloc}(l)$ (Def. of $\models_X$ for state formulae)

15 $T(0) \not\models_X \boxed{\forall}(V', \mathit{atloc}(l))$ (Def. of $\boxed{\forall}$)

CASE $\varsigma = Q$: This case is structured as a proof by contradiction. We have $T(0) =_V T'(0)$ and $T'(0) \not\models_X Q$. We will show that from $T(0) \models_X \boxed{\forall}(V', Q)$ we can derive a contradiction, leading us to conclude that our goal formula $T(0) \not\models_X \boxed{\forall}(V', Q)$ must hold.

Since $\boxed{\forall}(V', Q) = \forall V'.\, Q$, the assumption $T(0) \models_X \boxed{\forall}(V', Q)$ implies that $T(0) \models_X \forall V'.\, Q$. We now case split on the form of $T(0)$, which must be either $\mathbf{final}(s, h)$, $\mathbf{goto}(l, (s, h))$, or $\langle k, (s, h)\rangle$. As these are all handled the same way (only the s, h portion is important), we will only consider $\langle k, (s, h)\rangle$ here.

From $T(0) =_V T'(0)$ and $T(0) = \langle k, (s, h)\rangle$ we have $T'(0) = \langle k', (s', h)\rangle$ such that $s' =_V s$. The assumption $T(0) \models_X \forall V'.\, Q$ implies that $(s, h) \models_X \forall V'.\, Q$ which implies that for all s'' such that s'' and s differ only in the values assigned to variables in V', we have $(s'', h) \models_X Q$. In particular, we will consider the s'' given below.

$$s''(x) = \begin{cases} s(x) & \text{if } x \notin V' \\ s'(x) & \text{if } x \in V' \end{cases}$$

We will now derive a contradiction from $(s'', h) \models_X Q$ and $T'(0) \not\models_X Q$ and $s' =_V s$. We start by proving $s'' =_{\mathit{fv}(Q)} s'$. Suppose $x \in \mathit{fv}(Q)$. Then since $V' = \mathit{fv}(Q) - V$ we have either $x \in V'$ or $x \in V$. If $x \in V'$ then by the definition of s'' we have $s''(x) = s'(x)$ which is our goal. If $x \in V$ then we can establish $s''(x) = s'(x)$ regardless of which case of the s'' definition we are in. If $s''(x) = s(x)$, then by $s' =_V s$ we have $s'(x) = s(x)$

and thus $s''(x) = s'(x)$. If $s''(x) = s'(x)$ then this is already our goal formula and we are done.

Now that we have shown $s'' =_{fv(Q)} s'$ we can apply Lemma 4 to our assumption of $(s'', h) \models_X Q$ to obtain $(s', h) \models_X Q$. Recall that $T'(0) = \langle k', (s', h)\rangle$. The definition of $\models_X$ then gives us that $T'(0) \models_X Q$. But this contradicts the assumption $T'(0) \not\models_X Q$.

Inductive Cases

We now consider the connectives that operate on path formulae. These constitute the inductive cases. We consider only the core connectives, as justified by the derivations in Figure 3.8 and Theorem 9.

CASE 3 [${\sim}\phi$]

CASE 3.1 [$\boxed{\exists}$ conjunct]

1 $T' \models_X {\sim}\phi$ (Assumption)

2 $T' \not\models_X \phi$ (Semantics of $\sim$ (Figure 3.2))

3 $T \not\models_X \boxed{\forall}(V', \phi)$ (Inductive Hypothesis)

4 $T \models_X {\sim}\big(\boxed{\forall}(V', \phi)\big)$ (Semantics of $\sim$ (Figure 3.2))

5 $T \models_X \boxed{\exists}(V', {\sim}\phi)$ (Def. of $\boxed{\exists}$ (Def. 28))

CASE 3.2 [$\boxed{\forall}$ conjunct] This case is the dual of the above case.

1 $T' \not\models_X {\sim}\phi$ (Assumption)

2 $T' \models_X \phi$ (Semantics of $\sim$ (Figure 3.2))

3 $T \models_X \boxed{\exists}(V', \phi)$ (Inductive Hypothesis)

4 $T \not\models_X {\sim}\big(\boxed{\exists}(V', \phi)\big)$ (Semantics of $\sim$ (Figure 3.2))

5 $T \models_X \boxed{\forall}(V', {\sim}\phi)$ (Def. of $\boxed{\forall}$ (Def. 28))

CASE 4 [$\phi_1 \mathbin{\dot{\wedge}} \phi_2$]

CASE 4.1 [$\boxed{\exists}$ conjunct]

1 $T' \models_X \phi_1 \curlywedge \phi_2$ (Assumption)

2 $T' \models_X \phi_1$ and $T' \models_X \phi_2$ (Semantics of $\curlywedge$ (Figure 3.2))

3 $T \models_X \boxed{\exists}(V', \phi_1)$ and $T \models_X \boxed{\exists}(V', \phi_2)$ (Inductive Hypothesis)

4 $T \models_X \boxed{\exists}(V', \phi_1) \curlywedge \boxed{\exists}(V', \phi_2)$ (Semantics of $\curlywedge$ (Figure 3.2))

5 $T \models_X \boxed{\exists}(V', \phi_1 \curlywedge \phi_2)$ (Def. of $\boxed{\exists}$ (Def. 28))

CASE 4.2 [$\boxed{\forall}$ conjunct]

1 $T' \not\models_X \phi_1 \curlywedge \phi_2$ (Assumption)

2 $T' \not\models_X \phi_1$ or $T' \not\models_X \phi_2$ (Semantics of $\curlyvee$ (Figure 3.2))

Without loss of generality, we assume that the $T' \not\models_X \phi_1$ case holds. The other case is identical.

3 $T' \not\models_X \phi_1$ (Given)

4 $T \not\models_X \boxed{\forall}(V', \phi_1)$ (Inductive Hypothesis)

5 $T \not\models_X \boxed{\forall}(V', \phi_1) \curlywedge \boxed{\forall}(V', \phi_2)$ (Semantics of $\curlywedge$ (Figure 3.2))

6 $T \not\models_X \boxed{\forall}(V', \phi_1 \curlywedge \phi_2)$ (Def. of $\boxed{\forall}$ (Def. 28))

CASE 5 [ϕ_1 **U** ϕ_2]

CASE 5.1 [$\boxed{\exists}$ conjunct]

1 $T' \models_X \phi_1 \,\mathbf{U}\, \phi_2$ (Assumption)

2 $\exists i.\, 0 \leq i < \mathit{len}(T') \wedge (T'_i \models_X \phi_2) \wedge \big(\forall j.\, 0 \leq j < i \Rightarrow T'_j \models_X \phi_1\big)$
(Semantics of **U** (Figure 3.2))

3 $(0 \leq i < \mathit{len}(T')) \wedge (T'_i \models_X \phi_2) \wedge \big(\forall j.\, 0 \leq j < i \Rightarrow T'_j \models_X \phi_1\big)$
($\exists$-elim)

We first establish that there is a T_k such that $T_k \models_X \boxed{\exists}(V', \phi_2)$

4 $T' \sim_{=_V} T$ (Assumption (3.23) and Theorem 11)

5 $T'_i \sim_{=_V} T_{f(i)}$ (Lemma 8)

6 $(f(i) < \mathit{len}(T)) \Leftrightarrow (i < \mathit{len}(T'))$

(Def. of $\sim_{=_V}$ (Def. 19) and Def. of *matches* (Def. 18))

7 $f(i) < \mathit{len}(T)$ (Above and line 3 first conjunct)

8 $0 \leq f(i)$ (f has type $\mathbb{N} \to \mathbb{N}$)

9 $0 \leq f(i) < \mathit{len}(T)$ (Above two lines)

10 $T'_i \models_X \phi_2$ (3 second conjunct)

11 $T_{f(i)} \models_X \boxed{\exists}(V', \phi_2)$ (Induction Hypothesis: 5 and 10)

12 $(0 \leq f(i) < \mathit{len}(T)) \wedge (T_{f(i)} \models_X \boxed{\exists}(V', \phi_2))$ ($\wedge$-intro, above and line 9)

We next show that for all j such that $0 \leq j < f(i)$ we have $T_j \models_X \boxed{\exists}(V', \phi_1)$

13 $0 \leq j < f(i)$ (Assumption)

14 $f^{-1}(j) < f^{-1}(f(i))$ (Lemma 8, monotonicity of f^{-1})

15 $f^{-1}(f(i)) \leq i$ (Lemma 8, composition of f and f^{-1})

16 $0 \leq f^{-1}(j)$ (f^{-1} has type $\mathbb{N} \to \mathbb{N}$)

17 $0 \leq f^{-1}(j) < i$ (Previous three lines)

18 $T'_{f^{-1}(j)} \models_X \phi_1$ (line 3 last conjunct and 17)

19 $T_j \sim_{=_V} T'_{f^{-1}(j)}$ (Lemma 8)

20 $T_j \models_X \boxed{\exists}(V', \phi_1)$ (Induction Hypothesis: 18, 19)

21 $0 \leq j < f(i) \Rightarrow T_j \models_X \boxed{\exists}(V', \phi_1)$ (Imp. Intro.: lines 13 and 20)

22 $\forall j.\, 0 \leq j < f(i) \Rightarrow T_j \models_X \boxed{\exists}(V', \phi_1)$ ($\forall$-introduction)

23 $\big(\exists x.\, 0 \leq x < \mathit{len}(T) \wedge T_x \models_X \boxed{\exists}(V', \phi_2) \wedge \big(\forall j.\, 0 \leq j < x \Rightarrow T_j \models_X \boxed{\exists}(V', \phi_1)\big)$

($\exists$-intro with $x = f(i)$: lines 12 and 22)

24 $T \models_X \big(\boxed{\exists}(V', \phi_1)\big)\ \mathbf{U}\ \big(\boxed{\exists}(V', \phi_2)\big)$ (Semantics of $\mathbf{U}$ (Figure 3.2))

25 $T \models_X \boxed{\exists}(V', \phi_1\ \mathbf{U}\ \phi_2)$ (Def. of $\boxed{\exists}$ (Def. 28))

CASE 5.2 [$\boxed{\forall}$ Case]

1 $T' \not\models_X \phi_1 \, \mathbf{U} \, \phi_2$ (Assumption)

2 $\forall k.\ k \geq \mathit{len}(T') \lor T'_k \not\models_X \phi_2 \lor (\exists j.\ 0 \leq j < k \land T'_j \not\models_X \phi_1)$

(Semantics of **U** (Figure 3.2))

Let p be an arbitrary natural number.

3 $T'_{f(p)} \sim_{=_V} T_p$ (Lemma 8 and assumption (3.23))

4 $f(p) \geq \mathit{len}(T') \lor T'_{f(p)} \not\models_X \phi_2 \lor (\exists j.\ 0 \leq j < f(p) \land T'_j \not\models_X \phi_1)$

(line 2 with $k = f(p)$)

Case 1: $f(p) \geq \mathit{len}(T')$

5 $(f(p) < \mathit{len}(T')) \Leftrightarrow (p < \mathit{len}(T))$

(Def. of $\sim_{=_V}$ (Def. 19) and Def. of *matches* (Def. 18) and line 3)

6 $p \geq \mathit{len}(T)$ (Line 5 and this case assumption)

Case 2: $T'_{f(p)} \not\models_X \phi_2$

7 $T_p \not\models_X \boxed{\forall}(V', \phi_2)$ (Inductive Hypothesis: line 3 and this case assumption)

Case 3: $\exists j.\ 0 \leq j < f(p) \land T'_j \not\models_X \phi_1$

8 $0 \leq j < f(p) \land T'_j \not\models_X \phi_1$ ($\exists$-elim)

9 $f^{-1}(j) < f^{-1}(f(p))$ (Lemma 8, monotonicity of f^{-1})

10 $T_{f^{-1}(j)} \sim_{=_V} T'_j$ (Lemma 8 and assumption 3.23)

11 $f^{-1}(f(p)) \leq p$ (Lemma 8, composition of f and f^{-1})

12 $0 \leq f^{-1}(j) < p$ (lines 11 and 9 and f^{-1} has type $\mathbb{N} \to \mathbb{N}$)

13 $T_{f^{-1}(j)} \not\models_X \boxed{\forall}(V', \phi_1)$ (Inductive hypothesis: line 8 conjunct 2 and line 10)

14 $\exists m.\ 0 \leq m < p \land T_m \not\models_X \boxed{\forall}(V', \phi_1)$

($\exists$-intro with $m = f^{-1}(j)$: lines 12 and 13)

We now combine the results from Cases 1, 2, and 3 to obtain the following disjunction.

15 $p \geq \mathit{len}(T) \lor T_p \not\models_X \boxed{\forall}(V', \phi_2) \lor \exists m.\ 0 \leq m < p \land T_m \not\models_X \boxed{\forall}(V', \phi_1)$

($\lor$-intro: lines 7 and 14)

16 $\forall p.\, p \geq \mathit{len}(T) \lor T_p \not\models_X \boxed{\forall}(V', \phi_2) \lor \exists m.\, 0 \leq m < p \land T_m \not\models_X \boxed{\forall}(V', \phi_1)$ (∀-intro)

17 $T \not\models_X \boxed{\forall}(V', \phi_1)\ \mathbf{U}\ \boxed{\forall}(V', \phi_2)$ (Semantics of **U** (Figure 3.2))

18 $T \not\models_X \boxed{\forall}(V', \phi_1\ \mathbf{U}\ \phi_2)$ (Def. of $\boxed{\forall}$ (Def. 28))

□

We also have that the set of quantified variables can always be extended.

Lemma 11.

1. *If* $T \models_X \boxed{\exists}(V, \phi)$ *and* $V' \supseteq V$ *then* $T \models_X \boxed{\exists}(V', \phi)$.

2. *If* $T \not\models_X \boxed{\forall}(V, \phi)$ *and* $V' \supseteq V$ *then* $T \not\models_X \boxed{\forall}(V', \phi)$.

Proof. The proof is by induction on the structure of the formula ϕ. The inductive cases all follow directly from the inductive hypothesis, the definitions of $\boxed{\forall}$ and $\boxed{\exists}$, and the semantics of LTSL operators. We give the example of $\phi = {\sim}\phi'$. Suppose $T \models_X \boxed{\exists}(V, {\sim}\phi')$. Then by the definition of $\boxed{\exists}$ we have $T \models_X {\sim}(\boxed{\forall}(V, \phi'))$. This implies $T \not\models_X \boxed{\forall}(V, \phi')$. Applying the inductive hypothesis, we have $T \not\models_X \boxed{\forall}(V', \phi')$. Applying the semantics of $\models_X$ and the definition of $\boxed{\exists}$ to this formula gives us $T \models_X {\sim}(\boxed{\forall}(V', \phi'))$ and then $T \models_X \boxed{\exists}(V', {\sim}\phi')$. This completes the proof of this case.

The proof for $\boxed{\forall}$ is dual ($\boxed{\exists}$ and $\boxed{\forall}$ are interchanged, as are $\models_X$ and $\not\models_X$). We start from $T \not\models_X \boxed{\forall}(V, {\sim}\phi')$ and derive $T \not\models_X {\sim}(\boxed{\exists}(V, \phi'))$ and then $T \models_X \boxed{\exists}(V, \phi')$. The inductive hypothesis gives us $T \models_X \boxed{\exists}(V', \phi')$. Applying the semantics of ${\sim}$ gives us $T \not\models_X {\sim}(\boxed{\exists}(V', \phi'))$. Applying the definition of $\boxed{\forall}$ gives $T \not\models_X \boxed{\forall}(V', {\sim}\phi')$.

The base cases *err*, *final*, and $\mathit{atloc}(l)$ are all straightforward since if ϕ is one of these formulae, we have $\boxed{\exists}(V, \phi) = \boxed{\forall}(V, \phi) = \phi$ for all sets of variables V.

The only interesting case is $\phi = Q$. In this case, the $\boxed{\exists}$ conjunct follows from the fact that, $\exists V.\ Q \Rightarrow \exists V'.\ Q$ if $V' \supseteq V$. Formally, we have $T \models_X \boxed{\exists}(V, Q)$. Applying the definition of $\models_X$ gives us that $T(0) = \langle k, (s, h)\rangle$ or $T(0) = \mathbf{goto}(l, (s, h))$ or

$T(0) = \mathbf{final}(s, h)$. In all these cases we have $(s, h) \models_X \boxed{\exists}(V, Q)$, which is equivalent to $(s, h) \models_X \exists V.\ Q$. At this point, we reason that for any V' such that $V' \supset V$, we have $(s, h) \models_X \exists V.\ Q$ implies $(s, h) \models_X \exists V'.\ Q$. Re-applying the definitions of $\boxed{\exists}$ and $\models_X$ we then derive $T(0) \models_X \boxed{\exists}(V', Q)$ and finally $T \models_X \boxed{\exists}(V', Q)$.

The $\boxed{\forall}$ case is similar except that we make use of the fact that if $V' \supseteq V$ then $(s, h) \not\models_X \forall V.\ Q$ implies $(s, h) \not\models_X \forall V'.\ Q$. □

3.3.3 Example

Consider the example below, which iterates through a linked list.

```
$P \overset{\text{def}}{=}$
    $\mathsf{L_0}$ : goto $\mathsf{L_1}$
    $\mathsf{L_1}$ : branch $\mathsf{x} \neq \mathsf{nil} \Rightarrow$
                     $\mathsf{x} := \mathsf{x.next}$;
                     goto $\mathsf{L_1}$,
                 $\mathsf{x} = \mathsf{nil} \Rightarrow$ halt
         end
```

A shape analysis such as those in [Berdine et al., 2007, Gotsman et al., 2007, Distefano and Parkinson, 2008] might discover an invariant at $\mathsf{L_1}$ similar to the one below, where $ls(\mathsf{a}, \mathsf{x}, \mathsf{y})$ is the list segment predicate defined on page 69.

$$\exists \mathsf{a}, \mathsf{b}, \mathsf{x}'.\ ls(\mathsf{a}, \mathsf{x}', \mathsf{x}) * ls(\mathsf{b}, \mathsf{x}, \mathsf{nil})$$

This describes the shape of the heap (there are two linked list segments with x pointing to the head of the second segment) but includes no information about data structure sizes (the size information is existentially quantified). We will call analyses producing invariants such as this *shape-focused analyses* in recognition of the fact that they focus on shape invariants and support little, if any, reasoning about size (some analyses do keep limited size information by tracking whether a data structure is empty).

We can use the addition of extra variables and Corollary 2 to generate invariants that are more precise than those generated by a shape-focused analysis. In the following program we have included statements modifying variables a and b (we will show how to generate such a program in Chapter 4 and how to automate this process in Chapter 5).

```
P′ ≝
    L_0 : a := 0; b := n; goto L_1
    L_1 : branch x ≠ nil ⇒
                      x := x.next;
                      a := a + 1;
                      b := b − 1;
                      goto L_1,
                  x = nil ⇒ halt
          end
```

We have the following relationship between P and P'.

$$\mathit{traces}(\!(P \mid \mathit{ls}(\mathsf{n}, \mathsf{x}, \mathsf{nil}))\!) \approx_{=_{\{\mathsf{x}\}}} \mathit{traces}(\!(P' \mid \mathit{ls}(\mathsf{n}, \mathsf{x}, \mathsf{nil}))\!)$$

Note that the precondition assumes the existence of a program variable n which initially contains the length of the list at x. We can prove that the following LTSL property holds of $(\!(P' \mid \mathit{ls}(\mathsf{n}, \mathsf{x}, \mathsf{nil}))\!)$.

$$\mathbf{G}\Big(\mathit{atloc}(\mathsf{L}_1) \supset \big(\exists \mathsf{x}'.\ (\mathit{ls}(\mathsf{a}, \mathsf{x}', \mathsf{x}) * \mathit{ls}(\mathsf{b}, \mathsf{x}, \mathsf{nil})) \wedge \mathsf{a} + \mathsf{b} = \mathsf{n}\big)\Big)$$

By Corollary 2 we then have that the following property holds of $(\!(P \mid \mathit{ls}(\mathsf{n}, \mathsf{x}, \mathsf{nil}))\!)$.

$$\mathbf{G}\Big(\mathit{atloc}(\mathsf{L}_1) \supset \big(\exists \mathsf{a}, \mathsf{b}, \mathsf{x}'.\ (\mathit{ls}(\mathsf{a}, \mathsf{x}', \mathsf{x}) * \mathit{ls}(\mathsf{b}, \mathsf{x}, \mathsf{nil})) \wedge \mathsf{a} + \mathsf{b} = \mathsf{n}\big)\Big)$$

The invariant at L_1 now expresses that the sum of the lengths of the list segments $(\mathsf{a} + \mathsf{b})$ is always equal to n.

In Chapter 5 we will show that by using this approach to verification, we can easily extend a shape-focused analysis to an analysis that also supports reasoning about integer invariants. Furthermore, we can decompose the verification process in a way that allows the integer reasoning to occur independently of the shape reasoning.

3.4 Stuttering Simulation

In the previous sections, we presented some examples of programs that produce stuttering equivalent traces, as well as programs whose trace sets obey a stuttering containment relation. But we have not shown how to *prove* that the trace set of one program stuttering contains that of another. In this section, we introduce the concept of *stuttering simulation relations* and show how these can be used to prove that one program is an abstraction of another with respect to some equality relation on states. The definition below is based on Definition 4 from [Manolios, 2001] and corresponds to the concept of *well-founded simulation* (the well-foundedness referring to the rank functions that are involved in the definition).

Definition 29. *Given transition systems $S_1 = (A_1, I_1, F_1, \overset{1}{\dashrightarrow})$ and $S_2 = (A_2, I_2, F_2, \overset{2}{\dashrightarrow})$, we say that S_2 **E-stuttering simulates** S_1 iff there exists a relation R between the states of S_1 and S_2 that satisfies the following conditions*

1. *(Initial States Related)*
$$\forall a_1 \in I_1.\, \exists a_2 \in I_2.\, a_1\ R\ a_2$$
2. *(E-equivalent)* $\quad \forall a_1, a_2.\, (a_1\ R\ a_2) \Rightarrow (a_1\ E\ a_2)$
3. *(Transitions Match) There exist ranking functions $\mathit{rankt} : A_1 \times A_2 \to \mathbb{N}$ and $\mathit{rankl} : A_2 \times A_1 \times A_1 \to \mathbb{N}$ such that for all a_1, a_2, if $a_1\ R\ a_2$ and $a_1 \overset{1}{\dashrightarrow} a_1'$ then one of the following holds:*
   (a) *(S_2 Matches)* $\quad \exists a_2'.\, (a_2 \overset{2}{\dashrightarrow} a_2') \wedge (a_1'\ R\ a_2')$
   (b) *(S_1 Stutters)* $\quad (a_1'\ R\ a_2) \wedge (\mathit{rankt}(a_1', a_2) < \mathit{rankt}(a_1, a_2))$
   (c) *(S_2 Stutters)*
$$\exists a_2'.\, (a_2 \overset{2}{\dashrightarrow} a_2') \wedge (a_1\ R\ a_2') \wedge (\mathit{rankl}(a_2', a_1, a_1') < \mathit{rankl}(a_2, a_1, a_1'))$$
4. *(Final States Related) If $a_1\ R\ a_2$ then $a_1 \in F_1 \Leftrightarrow a_2 \in F_2$.*

*We call R an E-**stuttering simulation relation** and write $S_1 \underset{\sim}{\sqsubset}_{R,E} S_2$ to indicate that R is an E-stuttering simulation relation relating S_1 and S_2. We will also state the existence of such an R using the phrase "S_2 E-stuttering simulates S_1".*

Note that the definition allows three types of behavior when S_1 can take a step (conditions 3a, 3b, and 3c). The first corresponds to the standard requirement of simulation relations and specifies that the transition system on the right can match the step that the system on the left makes. The second and third conditions are what classifies this definition as stuttering simulation. These conditions allow for cases where only one of the systems takes a step. In such cases the system making the transition is said to "stutter," since the pre- and post-states of the transition are both E-equivalent. Thus, the state is repeated (with respect to the equivalence E), which is the connection with the common usage of "stutter" as the generation of repeated words or sounds. We include the conditions involving $rankt$ and $rankl$ to ensure that one system cannot stutter infinitely.

Given this definition of stuttering simulation, we can obtain the following theorem, which tells us that stuttering simulation implies stuttering trace containment. The fact that we prohibit infinite stuttering is important here, as this theorem would not hold without this restriction.

Theorem 18. *If $\exists R.\ S \underset{\sim}{\sqsubset}_{R,E} S'$ then traces$(S) \lesssim_E$ traces(S').*

Proof. (adapted from the proof of Proposition 1 in [Manolios, 2001]) We assume that $\exists R.\ S \underset{\sim}{\sqsubset}_{R,E} S'$ and $S = (A, I, F, \dashrightarrow)$ and $S' = (A', I', F', \dashrightarrow')$. We must show the following.

$$\forall T \in \textit{traces}(S).\ \exists T' \in \textit{traces}(S').\ T \sim_E T'$$

The definition of $\sim_E$ (Def. 19) states that this is equivalent to the following.

$$\forall T \in \textit{traces}(S).\ \exists T' \in \textit{traces}(S').\ \exists \alpha, \beta.\ matches(T, T', \alpha, \beta, E)$$

We will assume $T \in$ *traces*(S) and give a definition of T' such that $T' \in$ *traces*(S') and the following holds

$$\exists \alpha, \beta.\ matches(T, T', \alpha, \beta, E) \tag{3.25}$$

As we produce T', we also define α and β. Recall that α and β partition T and T' respectively into blocks of elements which are E-equivalent. Recall also that $\alpha(i)$ gives the index of the start of block i in trace T (and similarly for β and T'). Formally, we must provide an α and β satisfying the following (obtained by expanding (3.25) according to Definition 18).

$$\alpha(0) = \beta(0) = 0 \tag{3.26}$$

$$\begin{gathered}\forall i, j, k.\ \alpha(i) \leq j < \alpha(i+1) \wedge \beta(i) \leq k < \beta(i+1) \Rightarrow \\ \big(j < \mathit{len}(T) \Leftrightarrow k < \mathit{len}(T')\big) \wedge \big(j < \mathit{len}(T) \Rightarrow (T(j))\ E\ (T'(k))\big)\end{gathered} \tag{3.27}$$

The definition of α and β is by recursion on the block number. We assume we are given $\alpha(i)$, $\beta(i)$, and from these define $\alpha(i+1)$ and $\beta(i+1)$. We also assume that if $\alpha(i) < \mathit{len}(T)$ then we are provided with $T'(\beta(i))$ such that $\big(T(\alpha(i))\big)\ R\ \big(T(\beta(i))\big)$. If $\alpha(i) < \mathit{len}(T)$ we also build the i^{th} block of T'—that is, we define the elements $T'(k)$ where $\beta(i) \leq k < \beta(i+1)$. These are defined so as to establish (3.27) for block i, which can be split into the following two implications.

$$\begin{aligned}\forall j, k.\ \alpha(i) \leq j < \alpha(i+1) \wedge \beta(i) \leq k < \beta(i+1) \Rightarrow \\ (j < \mathit{len}(T)) \Leftrightarrow (k < \mathit{len}(T'))\end{aligned} \tag{3.28}$$

$$\begin{aligned}\forall j, k.\ \alpha(i) \leq j < \alpha(i+1) \wedge \beta(i) \leq k < \beta(i+1) \Rightarrow \\ \big(j < \mathit{len}(T) \Rightarrow \big(T(j)\big)\ E\ \big(T'(k)\big)\big)\end{aligned} \tag{3.29}$$

Finally, if $\alpha(i+1) < \mathit{len}(T)$ then we define $T'(\beta(i+1))$ such that it satisfies $\big(T(\alpha(i+1))\big)\ R\ \big(T'(\beta(i+1))\big)$, thus ensuring that the assumptions for generating the next block hold. We give a pictorial overview of the proof setup in Figure 3.9.

Base Case We start with the base case for T', α, and β. Condition (3.26) requires us to set $\alpha(0) = 0$ and $\beta(0) = 0$. Next we define $T'(0)$ given $T(0)$. We have from $S \sqsubseteq_{R,E} S'$ that $\forall a \in I.\ \exists a' \in I'.\ a\ R\ a'$. Since $T \in \mathit{traces}(S)$ we have that $T(0) \in I$. Thus, $\exists a' \in I'.\ T(0)\ R\ a'$. We set $T'(0)$ equal to this a', thus giving us $(T(0))\ R\ (T'(0))$.

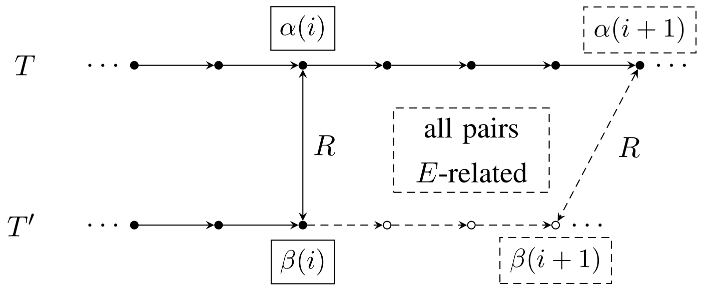


Figure 3.9: Pictorial overview of the proof of Theorem 3.9. The picture depicts how we build up T', α, and β. Solid elements of the figure are given. These include $\alpha(i)$, $\beta(i)$, the elements of T and the fact that $T(\alpha(i))\ R\ T'(\beta(i))$. The dashed elements are defined / proved in terms of these givens. Definitions must be provided for $\alpha(i+1)$, $\beta(i+1)$, and the elements of T' from index $\beta(i)$ to $\beta(i+1)$. It must then be proved that $\big(T(\alpha(i+1))\big)\ R\ \big(T'(\beta(i+1))\big)$ and that $\big(T(a)\ R\ T'(b)\big)$ for all a, b such that $\alpha(i) \leq a < \alpha(i+1)$ and $\beta(i) \leq b < \beta(i+1)$.

Recursive Case We break the proof of the recursive case into three sub-cases: either $\alpha(i) < \mathit{len}(T) - 1$ (the trace T contains at least two elements starting at $\alpha(i)$) or $\alpha(i) = \mathit{len}(T) - 1$ (the element at $\alpha(i)$ is the last element in the trace T) or $\alpha(i) > \mathit{len}(T) - 1$ (the index $\alpha(i)$ is past the end of the trace T).

CASE 1 [$\alpha(i) = \mathit{len}(T) - 1$] If $\alpha(i)$ is the index of the last element in the trace T, then we make T' end at $\beta(i)$. The constraints on well-formed traces ensure that since $\alpha(i)$ is the index of the last element in T, we have $T(\alpha(i)) \in F$. From condition 4 in the definition of simulation, and the fact that $\big(T(\alpha(i)\big)\ R\ \big(T'(\beta(i))\big)$, we have that $T'(\beta(i)) \in F'$, which ensures that taking $T'(\beta(i))$ to be the last element of trace T' results in a well-formed trace. We set $\alpha(i+1) = \alpha(i) + 1$ and $\beta(i+1) = \beta(i) + 1$. We now must check (3.28) and (3.29). We have $\big(T(\alpha(i)\big)\ R\ \big(T'(\beta(i))\big)$ which, by condition 2 of Definition 29, implies $\big(T(\alpha(i)\big)\ E\ \big(T'(\beta(i))\big)$. This establishes (3.29). For equation (3.28), we note that $\alpha(i) < \mathit{len}(T)$ and $\beta(i) < \mathit{len}(T')$ while $\alpha(i+1) \geq \mathit{len}(T)$ and $\beta(i+1) \geq \mathit{len}(T')$. This, combined with the fact that $\alpha(i+1) = \alpha(i) + 1$ and $\beta(i+1) = \beta(i) + 1$ is sufficient to establish (3.28).

CASE 2 [$\alpha(i) > \mathit{len}(T)-1$] In this case, we cannot satisfy the antecedent $j < \mathit{len}(T)$ in 3.29. Thus, that formula holds vacuously. Our rule above for ending the trace T' ensured that $\alpha(i) \geq \mathit{len}(T) \Rightarrow \beta(i) \geq \mathit{len}(T')$, so we can establish 3.28 regardless of what $\alpha(i+1)$ and $\beta(i+1)$ are set to (j and k in that formula will both index past the end of the trace). Essentially, we are past the end of both traces, so the values of α and β at this point are not relevant. Since we are free to set them to any values provided the functions remain strictly increasing, we choose $\alpha(i+1) = \alpha(i) + 1$ and $\beta(i+1) = \beta(i) + 1$.

CASE 3 [$\alpha(i) < \mathit{len}(T) - 1$] If T contains at least two elements at $\alpha(i)$, then we have $T(\alpha(i)) \dashrightarrow T(\alpha(i) + 1)$. Since we also have $S \sqsubseteq_{R,E} S'$ and $\big(T(\alpha(i))\ R\ T'(\beta(i))\big)$, then by Definition 29, we know that either condition 3a, 3b, or 3c holds. We now case split on these possibilities.

CASE 3.1 [Condition 3a *(S' Matches)*] In this case, we have that there exists an a' such that $(T'(\beta(i)) \dashrightarrow' a') \wedge (T(\alpha(i) + 1)\ R\ a')$. Since each transition system takes a step to new states which are related, we start a new block in each trace. We set $\alpha(i+1) = \alpha(i)+1$ and $\beta(i + 1) = \beta(i) + 1$. We set $T'(\beta(i + 1)) = a'$. Applying these definitions to $T(\alpha(i) + 1)\ R\ a'$, we obtain $\big(T(\alpha(i + 1))\big)\ R\ \big(T'(\beta(i + 1))\big)$. Note that $T(\alpha(i))$ and $T'(\beta(i))$ are the only elements in the i^{th} block of T and T', respectively. We also have $\big(T(\alpha(i))\big)\ R\ \big(T'(\beta(i))\big)$, and that R-relation implies E-equivalence (condition 2 of Definition 29). These facts together are sufficient to prove (3.29). Equation (3.28) follows from the fact that neither $T(\alpha(i))$ nor $T(\beta(i))$ are the last elements in their respective traces.

CASE 3.2 [Condition 3b *(S Stutters)*] We further assume that condition 3a does not hold (otherwise, this situation would be handled by the case above). In this case, we have $\big(T(\alpha(i){+}1)\big)\ R\ \big(T'(\beta(i))\big)$ and $\mathit{rankt}(T(\alpha(i){+}1), T'(\beta(i))) < \mathit{rankt}(T(\alpha(i)), T'(\beta(i)))$. We will consider the longest sub-sequence of T starting at index $\alpha(i)$ such that condition 3b holds for consecutive elements, but condition 3a does not. This will be used to define the i^{th} block of T'.

Let n be the maximum integer such that

$$\begin{aligned}&\forall l.\ 1 \leq l \leq n \Rightarrow\\ &\big(T(\alpha(i) + l)\big)\ R\ \big(T'(\beta(i))\big) \wedge \Big(\nexists a'.\ \big(T'(\beta(i)) \dashrightarrow' a'\big) \wedge \big(T(\alpha(i) + l)\ R\ a'\big)\Big)\end{aligned} \tag{3.30}$$

Note that $n \geq 1$ since the above holds for the current step of T. Also, n must be finite due to the well-foundedness of $rankt$. We set $\alpha(i+1) = \alpha(i)+n+1$ and $\beta(i+1) = \beta(i)+1$. The value of $T'(\beta(i+1))$ depends on whether $T(\alpha(i)+n)$ is the last element of T.

CASE 3.2.1 [$T(\alpha(i)+n)$ is the last element of T] In this case, $T'(\beta(i))$ will be the last element of T' and we proceed as in CASE 1. From Definition 12 we have $T(\alpha(i)+n) \in F$. We have $\big(T(\alpha(i)+n)\big)\ R\ \big(T'(\beta(i))\big)$ from (3.30). By condition 4 of Definition 29 we then have $T'(\beta(i)) \in F'$ and thus $T'(\beta(i))$ is a valid last state for T', so we leave T' undefined past $\beta(i)$. We set $\alpha(i+1) = \alpha(i)+n+1$ and $\beta(i+1) = \beta(i)+1$. By (3.30) we have $\big(T(\alpha(i)+l)\big)\ R\ \big(T'(\beta(i))\big)$ for $1 \leq l \leq n$ and thus $\big(T(\alpha(i)+l)\big)\ E\ \big(T'(\beta(i))\big)$, thus satisfying (3.29). Equation 3.28 follows from the fact that $\alpha(i)+n$ is the last index of T and $\beta(i)$ is the index of the last element of T'.

CASE 3.2.2 [$T(\alpha(i)+n)$ is not the last element of T] In this case, we let $\alpha(i+1) = \alpha(i)+n+1$ and we have that $T(\alpha(i)+n) \dashrightarrow T(\alpha(i)+n+1)$. By (3.30) and the maximality of n, we have that the consequent of (3.30) does not hold for $l = n+1$. Thus, we have the following.

$$\neg\Big(\big(T(\alpha(i)+n+1)\big)\ R\ \big(T'(\beta(i))\big)\Big) \vee \\ \Big(\exists a'.\ \big(T'(\beta(i)) \dashrightarrow' a'\big) \wedge \big(T(\alpha(i)+n+1)\big)\ R\ a'\Big) \quad (3.31)$$

We can show that the second disjunct must be the one that holds. Because we have $\big(T(\alpha(i)+n)\big)\ R\ \big(T'(\beta(i))\big)$ and $T(\alpha(i)+n) \dashrightarrow T(\alpha(i)+n+1)$, then by Definition 29 either 3a, 3b, or 3c must hold for the transition $T(\alpha(i)+n) \dashrightarrow T(\alpha(i)+n+1)$ and $T'(\beta(i))$.

- Condition (3a) corresponds exactly to the second disjunct in (3.31).
- Condition (3b) contradicts the first disjunct in (3.31), from which we conclude that the second disjunct must hold in this case.
- Condition (3c) cannot hold. If it did, we would have $\exists a'.\ T'(\beta(i)) \dashrightarrow' a' \wedge \big(T(\alpha(i)+n)\ R\ a'\big)$, which contradicts (3.30).

Thus, we have

$$\Big(\exists a'.\ \big(T'(\beta(i)) \dashrightarrow' a'\big) \wedge \big(T(\alpha(i)+n+1)\big)\ R\ a'\Big)$$

Let a' be the element described by the formula above. We set $\alpha(i+1) = \alpha(i)+n+1$ and set $\beta(i+1) = \beta(i)+1$. We set $T'(\beta(i+1)) = a'$. We have (3.28) since neither sequence is ending. We have (3.29) from assumption (3.30) and the fact that R-relation implies E-equivalence. We have $\big(T(\alpha(i+1))\big)\ R\ \big(T'(\beta(i+1))\big)$ from the assumption that $\big(T(\alpha(i)+n+1)\big)\ R\ a'$.

CASE 3.3 [Only condition 3c *(S' Stutters)* applies] This proceeds similarly to CASE 3.2. We again consider a maximal sequence (maximal with respect to prefix order) where only condition 3c applies. Formally, T'' is a maximal sequence with $T''(0) = T'(\beta(i))$ such that

$$\forall j.\ 0 \le j < \mathit{len}(T'') \Rightarrow \Big(\big(T(\alpha(i))\big)\ R\ \big(T''(j)\big)\Big) \tag{3.32}$$

and

$$\forall j.\ 0 \le j < (\mathit{len}(T'')-1) \Rightarrow \big(T''(j) \dashrightarrow' T''(j+1)\big) \tag{3.33}$$

and for each j such that $0 \le j < (\mathit{len}(T'')-1)$ we have

$$\nexists a.\ \big(T(\alpha(i)) \dashrightarrow a\big) \wedge a\ R\ \big(T''(j+1)\big) \tag{3.34}$$

(which states that condition 3a does not hold) and

$$\nexists a.\ \big(T(\alpha(i)) \dashrightarrow a\big) \wedge a\ R\ (T''(j))$$

(which states that condition 3b does not hold). There may be several choices for the sequence T''. Any choice satisfying the stated conditions is acceptable.

Note that T'' contains at least two elements since condition 3c (the assumption in this case) states that there is an a' such that $\big(T'(\beta(i)) \dashrightarrow' a'\big) \wedge \big(T(\alpha(i))\ R\ a'\big)$. This implies that there is a sequence satisfying these conditions with $T''(0) = T'(\beta(i))$ and $T''(1) = a'$. Let $n+1$ be the length of this sequence (thus making $T''(n)$ the last element in the sequence).

We have $\big(T(\alpha(i))\big)\ R\ \big(T''(n)\big)$ from (3.32) and we have $T(\alpha(i)) \dashrightarrow T(\alpha(i)+1)$ due to the fact that we are in CASE 3. Thus, condition 3 of Definition 29 states that either condition 3a, 3b, or 3c holds for the transition $T(\alpha(i)) \dashrightarrow T(\alpha(i)+1)$ and $T''(n)$.

Due to the maximality of T'', we cannot have that only condition 3c holds. If this were the case, then we would have $T''(n) \dashrightarrow' a'$ for some a' and T'' could be extended by setting $T''(n+1) = a'$, thus contradicting the maximality of T''.

Condition 3b also cannot hold. Suppose it did. Then we would have

$$T(\alpha(i)) \dashrightarrow T(\alpha(i)+1) \text{ and } \big(T(\alpha(i)+1)\big)\ R\ \big(T''(n)\big)$$

Since we already have $\big(T(\alpha(i))\big)\ R\ \big(T''(n-1)\big)$ and $T''(n-1) \dashrightarrow' T''(n)$ by (3.32) and (3.33), this implies that condition 3a holds of the transition $T(\alpha(i)) \dashrightarrow T(\alpha(i)+1)$ and $T''(n-1)$. This contradicts (3.34).

Thus, 3a must hold for $T(\alpha(i)) \dashrightarrow T(\alpha(i)+1)$ and $T''(n)$, implying that there is a b such that $T''(n) \dashrightarrow' b$ and $T(\alpha(i)+1)\ R\ b$. We handle this case similarly to CASE 3.1. We set $\alpha(i+1) = \alpha(i)+1$ and $\beta(i+1) = \beta(i)+n+1$. We let $T'(j) = T''(j-\beta(i))$ for $0 \le j \le n$. We set $T'(\beta(i+1))$ equal to b. Since T contains elements at least through index $\alpha(i+1)$ and T' contains indices at least through $\beta(i+1)$, we have (3.28). From 3.32 and the fact that R-relation implies E-equivalence, we have (3.29). We also have $T(\alpha(i+1))\ R\ b$ which implies $\big(T(\alpha(i+1))\big)\ R\ \big(T'(\beta(i+1))\big)$, completing our proof requirements. □

Simulation gives us a method of proving E-stuttering trace containment that only involves examining local transitions. Stuttering simulation is a stronger property than stuttering trace containment and actually preserves all ACTL$^*\backslash$X properties [Manolios, 2001]. Though we are only interested in LTSL, which is a subset of ACTL$^*\backslash$X, we will nevertheless use stuttering simulation as our main proof method, as its local character makes reasoning much easier.

3.5 Properties of Interest

While we have shown that stuttering equivalence preserves all LTSL properties, there are certain specific properties that we will focus on in our examples and experiments.

Definition 30.

1. *A program P is **safe** iff $P \models_X {\sim}(\mathbf{F}(\mathit{err}))$.*
2. *A program P is **terminating** iff $P \models_X \mathbf{F}(\mathit{final} \vee \mathit{err})$.*
3. *A formula Q is **invariant for** P **at** l iff $P \models_X \mathbf{G}(\mathit{atloc}(l) \supset Q)$.*
4. *An expression e^{i}_B **bounds** an expression e^{i} iff $P \models_X \mathbf{G}(e^{\mathrm{i}} \leq e^{\mathrm{i}}_B)$.*

In less formal terms, the *safe* property states that the execution state $\mathbf{error}$ is never reached. The *terminating* property holds exactly when the program has no infinite traces. The reason this statement is equivalent to the LTSL formula given above is that neither of the states $\mathbf{error}$ nor $\mathbf{final}(s, h)$ can ever make a transition. Thus, any trace containing $\mathbf{error}$ must be a finite trace with final state $\mathbf{error}$ (and similarly for $\mathbf{final}(s, h)$).

The *invariant at l* property holds exactly when Q is an invariant at location l. This means that whenever the program jumps to label l, the current store and heap satisfy Q. The *bounds* property states that at every step in the execution of program P, the value of the expression e^{i}_B (as evaluated in the current state) is greater than or equal to the value of the expression e^{i} (in other words, e^{i}_B is an upper bound of e^{i}). In general, when we consider bounds we will be interested in finding a bound for a variable in terms of specific other, designated values. For example, we may be interested in finding a bound on the size of a function's outputs in terms of its inputs.

Chapter 4

Instrumented Programs

The translation from heap-manipulating programs to numeric abstractions proceeds via an intermediate step that we call *instrumented programs*. These are programs that include the original program commands along with commands that update a set of *instrumentation variables* V, drawn from a set that is disjoint from the set of program variables. The additional commands describe how numeric counts, such as the size of a data structure, change during execution of the program. We call such additional commands *instrumentation commands*. The instrumentation commands are added to the instrumented program as a proof of memory safety is constructed and make use of the intermediate results of this safety analysis. Once the instrumented program has been constructed, the numeric abstraction is extracted from it by a simple syntax-directed translation. This step is discussed in Section 4.4. The end result is that the numeric abstraction $\overset{\mathrm{s}}{=}_{V'}$-stuttering simulates the original program, where V' is a subset of the program and instrumentation variables that depends on the details of the construction of the abstraction. This results in a numeric abstraction that is sound for both safety and liveness properties over variables in V'.

4.1 Theory

Informally, an instrumented program for program P is a program $\widehat{P}$ that contains all the commands and control-flow of P, but with the addition of some commands and branches that make use of a set of *instrumentation variables* that are separate from the program variables. These instrumentation variables play a role similar to that of auxiliary variables in program logics for concurrency [Owicki and Gries, 1976].

In Figure 4.2 we give a set of inference rules for establishing the judgment $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ which is read "$\widehat{P}$ is an instrumented version of P" and also explicitly lists V, the set of instrumentation variables and Γ, a mapping from labels to separation logic formulae that specifies program invariants for each label. This judgement is intended to capture the fact that $\widehat{P}$ simulates P when both are started from states satisfying $\Gamma(\mathit{initloc}(P))$ (the invariant for the initial location). The soundness theorem for the system, proved in Section 4.3, states that the proof rules described in this chapter do ensure the existence of such a simulation.

Figure 4.1 defines a similar judgment at the level of continuations. The judgment for continuations, which has the form $\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k$, should be provable only if, when started from a state satisfying Q, the continuation $\widehat{k}$ simulates the continuation k. For continuations, this simulation means that $\widehat{k}$ can match any transition k makes and the continuations eventually either both halt, both reach an error, or both jump to the same label.

The simulation relation we obtain in Section 4.3 enforces a relationship between the memory states of the two programs. The instrumented program $\widehat{P}$ modifies variables in V, but the original program P does not. The simulation relation ensures that, despite these extra commands involving new variables, for every execution trace T of the original program, there is a matching execution trace T' in the instrumented program such that T and T' agree on the values of the non-instrumentation variables (that is, all variables in the original program). This connection lets us check properties of P by instead checking them on $\widehat{P}$. For example, if x is a program variable and x is never assigned the value

0 in executions of $\widehat{P}$ then we can conclude that it is also never assigned the value 0 in executions of P.

Note that the property of being a valid instrumentation is defined with respect to program invariants Γ and, in the case of continuations, with respect to a precondition Q. If we view the construction of a proof in the system given in Figure 4.1 as proceeding in a bottom-up manner, then instrumentation proceeds in lock-step with the derivation of a partial correctness proof of the program. The rules COMMAND and BRANCH tell us how to update the precondition to reflect the results of executing an existing command and rules INST-ASSIGN, INST-DISJ, INST-EXISTS, and INST-ASSUME tell us which new commands can be inserted. The triple $\{Q\}\ c\ \{Q'\}$ in the COMMAND rule is a partial correctness triple and holds iff

$$\forall s, h.\ ((s,h) \models Q) \Rightarrow \big(\mathbf{error} \notin (\llbracket c \rrbracket\,(s,h))\big) \wedge \big(\forall (s',h') \in (\llbracket c \rrbracket\,(s,h)).\ (s',h') \models Q'\big)$$

Note that such triples can be found only if c is memory safe under precondition Q (this is required due to the clause $\mathbf{error} \notin (\llbracket c \rrbracket\,(s,h))$ and the fact that $\mathbf{error}$ is the result of any command that violates memory safety). For this reason, the rules in Figures 4.1 and 4.2 will only let us derive instrumented versions of a program if the original program is memory safe.

A key difference between this approach to command insertion and the auxiliary variable approach lies with the INST-EXISTS rule. This rule tells us that if we insert an assignment $x := ?$, then we can remove an existential quantifier on x. This may seem odd, since $\{\exists x.\ Q\}\ x := ?\ \{Q\}$ is not a valid partial correctness triple. However, inserting such a command and reasoning from the unquantified formula is sound because our soundness result is based on simulation. To maintain soundness, we must show that if the original program can take a step, then there exists a step in the instrumented program that takes us to a related state. The fact that the semantics of $x := ?$ includes all possible updates to x allows us to find such a step. Similarly, the INST-DISJ rule allows us to reason separately about each side of a disjunction. Again, this is valid because we are targeting a correspondence between the two programs that is based on simulation. We say more about these connections in Section 4.7.

Halt

$$\frac{}{\Gamma \vdash \{Q\}\ \mathsf{halt} \blacktriangleright_V \mathsf{halt}}$$

Abort

$$\frac{}{\Gamma \vdash \{Q\}\ \mathsf{abort} \blacktriangleright_V \mathsf{abort}}$$

Goto

$$\frac{\Gamma(l) = Q}{\Gamma \vdash \{Q\}\ \mathsf{goto}\ l \blacktriangleright_V \mathsf{goto}\ l}$$

Command

$$\frac{\{Q\}\ c\ \{Q'\} \qquad \Gamma \vdash \{Q'\}\ \widehat{k} \blacktriangleright_V k}{\Gamma \vdash \{Q\}\ (c\,\mathbf{;}\,\widehat{k}) \blacktriangleright_V (c\,\mathbf{;}\,k)}$$

Strengthening

$$\frac{Q \Rightarrow Q' \qquad \Gamma \vdash \{Q'\}\ \widehat{k} \blacktriangleright_V k}{\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k}$$

Branch

$$\frac{\forall i.\ (\Gamma \vdash \{Q \wedge e_i^{\mathrm{b}}\}\ \widehat{k_i} \blacktriangleright_V k_i)}{\Gamma \vdash \{Q\}\ \mathsf{branch}\ \ldots, e_i^{\mathrm{b}} \Rightarrow \widehat{k_i}, \ldots\ \mathsf{end} \blacktriangleright_V \mathsf{branch}\ \ldots, e_i^{\mathrm{b}} \Rightarrow k_i, \ldots\ \mathsf{end}}$$

False

$$\frac{}{\Gamma \vdash \{\mathsf{false}\}\ \mathsf{halt} \blacktriangleright_V k}$$

Inst-Assign

$$\frac{\{Q\}\ x^\tau := e^\tau\ \{Q'\} \qquad \Gamma \vdash \{Q'\}\ \widehat{k} \blacktriangleright_V k}{\Gamma \vdash \{Q\}\ (x^\tau := e^\tau\,\mathbf{;}\,\widehat{k}) \blacktriangleright_V k}\ x^\tau \in V$$

Inst-Disj

$$\frac{\Gamma \vdash \{Q_1\}\ \widehat{k_1} \blacktriangleright_V k \qquad \Gamma \vdash \{Q_2\}\ \widehat{k_2} \blacktriangleright_V k}{\Gamma \vdash \{Q_1 \vee Q_2\}\ \mathsf{branch}\ \mathsf{true} \Rightarrow \widehat{k_1}, \mathsf{true} \Rightarrow \widehat{k_2}\ \mathsf{end} \blacktriangleright_V k}$$

Inst-Exists

$$\frac{\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k}{\Gamma \vdash \{\exists x^\tau.\ Q\}\ (x^\tau := ?^\tau\,\mathbf{;}\,\widehat{k}) \blacktriangleright_V k}\ x^\tau \in V$$

Inst-Assume

$$\frac{Q \Rightarrow e^{\mathrm{b}} \qquad \Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k}{\Gamma \vdash \{Q\}\ \mathsf{assume}(e^{\mathrm{b}})\,\mathbf{;}\,\widehat{k} \blacktriangleright_V k}$$

Figure 4.1: Rules for establishing that $\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k$, read "under precondition Q, with label invariants Γ, the continuation $\widehat{k}$ is an instrumented version of k with instrumentation variables V." Premises of the form $\{Q\}\ c\ \{Q'\}$ are partial correctness triples and hold iff for all s, h, $(s, h) \models Q$ implies $(\forall(s', h') \in (\llbracket c \rrbracket\,(s, h)).\ (s', h') \models Q')$. Premises of the form $Q \Rightarrow Q'$ hold iff $Q \Rightarrow Q'$ is valid (that is, $(s, h) \models (Q \Rightarrow Q')$ for all s, h).

INST-PROG

$$\frac{\begin{array}{c} dom(\widehat{P}) = dom(P) \\ fv(P) \cap V = \emptyset \qquad initloc(\widehat{P}) = initloc(P) \qquad \forall l \in dom(P).\, (\Gamma \vdash \{\Gamma(l)\}\ \widehat{P}(l) \blacktriangleright_V P(l)) \end{array}}{\Gamma \vdash \widehat{P} \blacktriangleright_V P}$$

Figure 4.2: Rule for proving that $\widehat{P}$ is an instrumented version of P. The function $fv(P)$ gives the set of variables occurring free in P. Since there are no binding constructs in our language, this is just the set of all variables appearing in P.

Notation As before, we will use circled numbers to label continuations in our examples. To help distinguish between the instrumented program and the original program, we will adopt the convention of using black numbers in white circles (①, ②, . . .) to represent control points in the original program and white numbers in black circles (❶, ❷, . . .) to represent control points in the instrumented program. We will also assign numbers such that if the original program contains a continuation labeled ② and the instrumented program contains a continuation labeled ❷ then we will have $\Gamma \vdash \{Q\}$ ❷ $\blacktriangleright_V$ ② for some Γ, V, and Q. Intuitively, this indicates that the control points ② and ❷ are related by the simulation relation used to demonstrate soundness.

4.1.1 Common Cases

The rules INST-ASSIGN, INST-DISJ, INST-EXISTS and INST-ASSUME allow us to expresses various facts about the behavior of numeric properties of data structures. These facts generally fall into four categories.

Deterministic Size Changes

We can record deterministic size changes using the INST-ASSIGN rule. Suppose we have the following definition of singly-linked list segments.

$$\begin{aligned}
&\mathit{ls}(n, \mathit{start}, \mathit{end}) \equiv\\
&\qquad (\mathbf{emp} \wedge \mathit{start} = \mathit{end} \wedge n = 0)\\
&\qquad \vee (n > 0 \wedge (\exists z.\ (\mathit{start} \mapsto [\mathsf{next} : z]) * \mathit{ls}(n-1, z, \mathit{end})))
\end{aligned}$$

and execute the code given below.

$$\begin{aligned}
L_1 : \text{①}\ \mathsf{branch}\ x \neq \mathsf{nil} \Rightarrow \text{②}\ x := x.\mathsf{next}\texttt{;}\ \text{③}\ \mathsf{goto}\ L_1,&\\
x = \mathsf{nil} \Rightarrow \text{④}\ \mathsf{halt\ end}&
\end{aligned}$$

An invariant of this code at label L_1 is $\exists n_1, n_2, x'.\ \mathit{ls}(n_1, x', x) * \mathit{ls}(n_2, x, \mathsf{nil})$. In order to track how the sizes of the segments are changing, we can generate an instrumented program for the code above. Let $\Gamma(L_1) = \exists x'.\ \mathit{ls}(n_1, x', x) * \mathit{ls}(n_2, x, \mathsf{nil})$. Then the code below is an instrumented version of the code above with instrumentation variables n_1 and n_2 (the assignments to n_1 and n_2 are added with the INST-ASSIGN rule). The variable n_2 tracks the quantity "length of the list segment from x to nil" and n_1 tracks the quantity "length of the list segment ending at x."

$$\begin{aligned}
L_1 : \text{❶}\ \mathsf{branch}\ x \neq \mathsf{nil} \Rightarrow \text{❷}\ x := x.\mathsf{next}\texttt{;}\ \text{❸}\ n_1 := n_1 + 1\texttt{;}&\\
n_2 := n_2 - 1\texttt{;}\ \mathsf{goto}\ L_1,&\\
x = \mathsf{nil} \Rightarrow \text{❹}\ \mathsf{halt\ end}&
\end{aligned}$$

Note that the existential quantification is dropped in the invariant used for the instrumented program (in $\Gamma(L_1)$ the variables n_1 and n_2 appear unquantified). This is possible because we are now updating n_1 and n_2 in the body of the loop. Viewed another way, it is by committing to an invariant in which n_1 and n_2 are unquantified that we are forced to write the appropriate updates to n_1 and n_2 in the body (if we update n_1 or n_2 incorrectly, we will not be able to show that $\Gamma(L_1)$ is an invariant). Figure 4.3 gives a derivation showing that the instrumentation we presented is a valid instrumented version of the original program according to the rules in Figures 4.1 and 4.2.

$$\dfrac{\begin{array}{c}\vdots\\ \{Q_3\}\ n_2 := n_2 - 1\ \{Q_4\}\end{array} \qquad \dfrac{\Gamma(L_1) = Q_4}{\Gamma \vdash \{Q_4\}\ \mathsf{goto}\ L_1 \blacktriangleright_{n_1,n_2} \mathsf{goto}\ L_1}\ \textsc{Goto}}{\Gamma \vdash \{Q_3\}\ n_2 := n_2 - 1\texttt{;}\ \mathsf{goto}\ L_1 \blacktriangleright_{n_1,n_2} \mathsf{goto}\ L_1}\ \text{I-A}$$

$$\dfrac{\begin{array}{c}\vdots\\ \{Q_2\}\ n_1 := n_1 + 1\ \{Q_3\}\end{array} \qquad \Gamma \vdash \{Q_3\}\ n_2 := n_2 - 1\texttt{;}\ \mathsf{goto}\ L_1 \blacktriangleright_{n_1,n_2} \mathsf{goto}\ L_1}{\Gamma \vdash \{Q_2\}\ \text{❸} \blacktriangleright_{n_1,n_2} \text{③}}\ \text{I-A}$$

$$\dfrac{\begin{array}{c}\vdots\\ \{Q_1 \wedge x \neq \mathsf{nil}\}\ x := x.\mathsf{next}\ \{Q_2\}\end{array} \qquad \Gamma \vdash \{Q_2\}\ \text{❸} \blacktriangleright_{n_1,n_2} \text{③}}{\Gamma \vdash \{Q_1 \wedge x \neq \mathsf{nil}\}\ \text{❷} \blacktriangleright_{n_1,n_2} \text{②}}\ \textsc{Cmd}$$

$$\dfrac{\Gamma \vdash \{Q_1 \wedge x \neq \mathsf{nil}\}\ \text{❷} \blacktriangleright_{n_1,n_2} \text{②} \qquad \dfrac{}{\Gamma \vdash \{Q_1 \wedge x = \mathsf{nil}\}\ \mathsf{halt} \blacktriangleright_{n_1,n_2} \mathsf{halt}}\ \textsc{Halt}}{\Gamma \vdash \{Q_1\}\ \text{❶} \blacktriangleright_{n_1,n_2} \text{①}}\ \textsc{Branch}$$

$$\begin{array}{rcl}
\Gamma(L_1) & = & \exists x'.\ ls(n_1, x', x) * ls(n_2, x, \mathsf{nil})\\
Q_1 & = & \exists x'.\ ls(n_1, x', x) * ls(n_2, x, \mathsf{nil})\\
Q_2 & = & \exists x'.\ ls(n_1 + 1, x', x) * ls(n_2 - 1, x, \mathsf{nil})\\
Q_3 & = & \exists x'.\ ls(n_1, x', x) * ls(n_2 - 1, x, \mathsf{nil})\\
Q_4 & = & \exists x'.\ ls(n_1, x', x) * ls(n_2, x, \mathsf{nil})
\end{array}$$

Figure 4.3: Derivation showing an instrumented program that performs a deterministic update of a variable representing the length of a linked list. I-A stands for INST-ASSIGN.

Non-deterministic Size Changes

Suppose we have the following definition of a binary tree, where n represents the number of nodes in the tree.

$$\begin{aligned} \mathit{tree}(n, r) \equiv\ & (n = 0 \wedge r = \mathsf{nil}) \\ & \vee (n > 0 \wedge \exists n_1, n_2.\ n = n_1 + n_2 + 1 \wedge \\ & \qquad \exists lc, rc.\ r \mapsto [\mathsf{left} : lc, \mathsf{right} : rc] \\ & \qquad\quad * \ \mathit{tree}(n_1, lc) * \mathit{tree}(n_2, rc)) \end{aligned}$$

If we now consider code for descending through the tree, we can obtain update commands similar to those obtained for the linked list example above. However, when a pointer p is advanced through a list, the change in the size of the list at p is deterministic (it always decreases by one). In the case of trees, if some pointer p descends to the left child, we do not have a deterministic function that describes how the number of nodes reachable from p changes. Instead, there is a relation between the two quantities which specifies that the number of nodes in the left sub-tree can range from zero to one less than the number of nodes in the full tree. We will use non-deterministic assignment to capture this update relation.

The original program we consider is given below. The program checks whether the tree at r is empty and, if it is not, it non-deterministically chooses a child to descend to. We have marked with ① a location of interest during creation of the instrumented program.

```
$L_1$ : branch $r \neq$ nil $\Rightarrow$ ① branch true $\Rightarrow r := r$.left;
                                               goto $L_1$,
                                          true $\Rightarrow r := r$.right;
                                               goto $L_1$ end
               $r =$ nil $\Rightarrow$ halt end
```

Let $\Gamma(L_1) = (\mathit{tree}(n, r)) * \mathsf{true}$ (where true is used to capture the part of the heap no longer below r in the tree) and let Q be the following formula

$$Q \overset{\text{def}}{=} (n > 0 \land n = n_1 + n_2 + 1) \land \\ \exists lc, rc.\ r \mapsto [\mathsf{left} : lc, \mathsf{right} : rc] * \mathit{tree}(n_1, lc) * \mathit{tree}(n_2, rc) * \mathsf{true}$$

We will now construct an instrumented version of this program using the following process, obtained by taking an algorithmic, bottom-up reading of the inference rules given in Figure 4.1.

1. Start with the continuation at L_1 and the invariant $\Gamma(L_1)$.
2. Copy commands from the original program over to the instrumented program, updating the current invariant using the rules Branch and Command.
3. If a halt or abort is encountered, then we can stop analyzing this branch.
4. If a $\mathsf{goto}\ L$ command is encountered, then we insert instrumentation commands using rules Inst-Exists, Inst-Assume, and Inst-Assign in order to establish the invariant $\Gamma(L)$.

This process is not general enough to give us the instrumentation we want in all cases (for example it will never insert new branches using the Inst-Branch rule) but it will suffice for this example. We give a more general procedure in Chapter 5.

Following steps 1 and 2 we can obtain the formula $\exists n_1, n_2.\ Q$ for the invariant at the position labeled with ① in the original program. We now must give an instrumentation of each case of the branch at this location. Let us consider first the case that chooses the left child. This case executes the continuation $r := r.\mathsf{left};\ \mathsf{goto}\ L_1$. A valid post-condition after executing $r := r.\mathsf{left}$ is the following

$$Q' \overset{\text{def}}{=} \exists n_1, n_2.\ n > 0 \land (n = n_1 + n_2 + 1) \land \\ \exists r', rc.\ r' \mapsto [\mathsf{left} : r, \mathsf{right} : rc] * \mathit{tree}(n_1, r) * \mathit{tree}(n_2, rc) * \mathsf{true}$$

We now need to add instrumentation commands that allow us to re-establish the invariant $\Gamma(L_1)$ which is $(\mathit{tree}(n, r)) * \mathsf{true}$. The commands we will add are the following, which are justified using the INST-EXISTS, INST-ASSUME, and INST-ASSIGN rules. A full derivation is given in Figure 4.4.

$$n_1 := ?\texttt{;}\ n_2 := ?\texttt{;}\ \mathsf{assume}(n = n_1 + n_2 + 1)\texttt{;}\ n := n_1$$

Executing these leads us to the invariant

$$\exists r', rc.\ r' \mapsto [\mathsf{left} : r, \mathsf{right} : rc] * \mathit{tree}(n, r) * \mathit{tree}(n_2, rc) * \mathsf{true}$$

which is labeled Q_2 in Figure 4.4. This formula implies $(\mathit{tree}(n, r)) * \mathsf{true}$ which is $\Gamma(L_1)$. This allows us to finish the processing of this branch by using the STRENGTHENING rule to show that we have the invariant $(\mathit{tree}(n, r)) * \mathsf{true}$ here. As this is equal to $\Gamma(L_1)$, this lets us use the GOTO rule to process the goto L_1 command.

We can perform the same analysis of the branch that descends into the right sub-tree and obtain the instrumentation commands below.

$$n_1 := ?\texttt{;}\ n_2 := ?\texttt{;}\ \mathsf{assume}(n = n_1 + n_2 + 1)\texttt{;}\ n := n_2$$

Putting this all together, the full instrumented version of this program is given below.

```
$L_1$ : branch $r \neq$ nil $\Rightarrow$ ❶ branch true $\Rightarrow r := r$.left; $n_1 := ?$; $n_2 := ?$;
                                                 assume($n = n_1 + n_2 + 1$);
                                                 $n := n_1$; goto $L_1$,
                                          true $\Rightarrow r := r$.right; $n_1 := ?$; $n_2 := ?$;
                                                 assume($n = n_1 + n_2 + 1$);
                                                 $n := n_2$; goto $L_1$ end,
             $r =$ nil $\Rightarrow$ halt end
```

Recall that we generated this program in a fairly directed manner. We copied commands from the original program into the instrumented program and only inserted instrumentation commands when this was necessary to establish an invariant in Γ. It still required some ingenuity to derive the post-conditions of commands and determine which

$$
\dfrac{\dfrac{\Gamma \vdash \{\exists n_2.\ Q_1\}\ \begin{array}{l} n_2 := ?\mathtt{;} \\ \mathsf{assume}(n = n_1 + n_2 + 1)\mathtt{;}\ n := n_1\ \mathtt{;}\mathsf{goto}\ L_1 \end{array} \blacktriangleright_{n,n_1,n_2} \mathsf{goto}\ L_1}{\dfrac{\Gamma \vdash \{Q_1\}\ \mathsf{assume}(n = n_1 + n_2 + 1)\mathtt{;}\ n := n_1\mathtt{;}\ \mathsf{goto}\ L_1 \blacktriangleright_{n,n_1,n_2} \mathsf{goto}\ L_1}{\dfrac{\Gamma \vdash \{Q_1\}\ n := n_1\mathtt{;}\ \mathsf{goto}\ L_1 \blacktriangleright_{n,n_1,n_2} \mathsf{goto}\ L_1}{\{Q_1\}\ n := n_1\ \{Q_2\} \qquad \dfrac{Q_2 \Rightarrow Q_3 \qquad \dfrac{\Gamma(L_1) = Q_3}{\Gamma \vdash \{Q_3\}\ \mathsf{goto}\ L_1 \blacktriangleright_{n,n_1,n_2} \mathsf{goto}\ L_1}\ \textsc{Goto}}{\Gamma \vdash \{Q_2\}\ \mathsf{goto}\ L_1 \blacktriangleright_{n,n_1,n_2} \mathsf{goto}\ L_1}\ \textsc{Strengthen}}\ \textsc{Inst-Assign}}\ \textsc{Inst-Assume}}\ \text{I-E}}{\Gamma \vdash \{\exists n_1, n_2.\ Q_1\}\ \begin{array}{l} n_1 := ?\mathtt{;}\ n_2 := ?\mathtt{;} \\ \mathsf{assume}(n = n_1 + n_2 + 1)\mathtt{;}\ n := n_1 \end{array} \blacktriangleright_{n,n_1,n_2} \mathsf{goto}\ L_1}\ \text{I-E}
$$

$$
\begin{array}{rcl}
Q_1 & = & n > 0 \wedge (n = n_1 + n_2 + 1) \wedge \\
 & & \quad \exists r', rc.\ r' \mapsto [\mathsf{left} : r, \mathsf{right} : rc] * \mathit{tree}(n_1, r) * \mathit{tree}(n_2, rc) * \mathsf{true} \\
Q_2 & = & \exists r', rc.\ r' \mapsto [\mathsf{left} : r, \mathsf{right} : rc] * \mathit{tree}(n, r) * \mathit{tree}(n_2, rc) * \mathsf{true} \\
Q_3 & = & \mathit{tree}(n, r) * \mathsf{true} \\
\Gamma(L_1) & = & \mathit{tree}(n, r) * \mathsf{true}
\end{array}
$$

Figure 4.4: Derivation showing that, for the tree traversal program on page 136, the commands given re-establish the invariant $\Gamma(L_1)$. We write I-E as an abbreviation for INST-EXISTS and abbreviate STRENGTHENING as STRENGTHEN.

instrumentation commands to insert (although the former could be handled by using strongest post-conditions). In Chapter 5 we will describe how to automate all portions of the instrumentation process.

Our semi-automated process had us insert instrumentation commands only immediately before goto commands. If we had chosen different points at which to insert the instrumentation commands, we could have obtained the code below, which places the commands that affect n_1 and n_2 before the branch instead of replicating them in each branch case.

```
$L_1$ : branch $r \neq$ nil ⇒ $n_1 := ?$; $n_2 := ?$;
                      assume($n = n_1 + n_2 + 1$);
                      branch true ⇒ $r := r$.left;
                                    $n = n_1$; goto $L_1$,
                             true ⇒ $r := r$.right;
                                    $n = n_2$; goto $L_1$ end
        $r =$ nil ⇒ halt end
```

Both this code and our previously derived code are valid instrumentations of the original program, as can be verified using the rules in Figure 4.1. However, the second, shorter program may be easier to verify using automated tools. In general, the less statements, variables, and branching a program contains, the easier it is for automated tools to handle. We say more about this in Section 5.11, which discusses our experimental results.

Branch Condition Translation

Let us return to the linked-list example from before. The instrumented code that we generated is replicated below.

```
$L_1$ : ❶ branch $x \neq$ nil ⇒ ❷ $x := x$.next; ❸ $n_1 := n_1 + 1$;
                              $n_2 := n_2 - 1$; goto $L_1$,
              $x =$ nil ⇒ ❹ halt end
```

This summarizes how n_1 and n_2 change during each iteration. Recall that n_1 and n_2 are the lengths of the list segments in the invariant $\exists x'.\ ls(n_1, x', x) * ls(n_2, x, \mathsf{nil})$. The instrumentation commands in the program above are sufficient to prove some properties of the list lengths. For example, we can show that the sum $n_1 + n_2$ is invariant at location L_1. However, we have not added any commands to indicate how n_1 and n_2 influence the truth of the branch condition. Thus, though we would like to use n_1 and n_2 to reason about termination of the code, we cannot obtain a ranking function because n_1 and n_2 are not bounded.

To obtain a more precise numeric abstraction that will be useful for termination reasoning, we need to notice that only certain values of n_2 are possible when the branch condition $x = \mathsf{nil}$ is true. Similarly, when $x \neq \mathsf{nil}$ is true, this also gives us information on the possible values of n_2. Specifically, if $x = \mathsf{nil}$ then $n_2 = 0$ and if $x \neq \mathsf{nil}$ then $n_2 > 0$. To record this information and make it available to subsequent analyses, we can use the INST-ASSUME rule to insert an assumption on n_2. The final instrumented program then becomes the following.

$$\begin{aligned}\mathsf{L}_1 : \mathsf{branch}\ x \neq \mathsf{nil} \Rightarrow \mathsf{assume}(n_2 > 0)\texttt{;}\ x := x.next\texttt{;}\\ n_1 := n_1 + 1\texttt{;}\ n_2 := n_2 - 1\texttt{;}\ \mathsf{goto}\ \mathsf{L}_1,\\ x = \mathsf{nil} \Rightarrow \mathsf{assume}(n_2 = 0)\texttt{;}\ \mathsf{halt\ end}\end{aligned}$$

It is now clear that, for any n_2, the program terminates. This is the case because n_2 decreases by one during each iteration and once $n_2 = 0$, the first assume statement prevents us from executing the loop body again. Values of n_2 such that $n_2 < 0$ are not possible as the two assume conditions together ensure that the only valid executions are those for which $n_2 \geq 0$ in the initial state. Ruling out the states where $n_2 < 0$ does not pose a problem for soundness since the precondition $\exists x'.\ ls(n_1, x', x) * ls(n_2, x, \mathsf{nil})$ implies that $n_2 \geq 0$.

Alternate Translation We could also have inserted a branch on n_2 using the INST-DISJ rule and then pruned inconsistent cases using the FALSE rule. Recall that the original code was as below.

$$\begin{aligned}L_1 : \textcircled{1}\ \mathsf{branch}\ x \neq \mathsf{nil} \Rightarrow \textcircled{2}\ x := x.\mathsf{next}\texttt{;}\ \textcircled{3}\ \mathsf{goto}\ L_1,\\ x = \mathsf{nil} \Rightarrow \textcircled{4}\ \mathsf{halt\ end}\end{aligned}$$

We start by noting that $\Gamma(L_1) = \exists x'.\ ls(n_1, x', x) * ls(n_2, x, \mathsf{nil})$ and this implies $Q_1 \vee Q_2$ where Q_1 and Q_2 are defined as follows.

$$\begin{aligned}Q_1 &\equiv \exists x'.\ ls(n_1, x', x) \wedge x = \mathsf{nil} \wedge n_2 = 0\\ Q_2 &\equiv \exists x', z.\ ls(n_1, x', x) * (x \mapsto [\mathsf{next} : z]) * ls(n_2 - 1, z, \mathsf{nil}) \wedge n_2 > 0\end{aligned}$$

This was obtained by replacing $ls(n_2, x, \mathsf{nil})$ with its definition and distributing $\wedge$ and $*$ over disjunction. We can then use the INST-DISJ rule to insert a non-deterministic branch

$$\mathsf{branch\ true} \Rightarrow \widehat{k_1}, \mathsf{true} \Rightarrow \widehat{k_2}\ \mathsf{end}$$

where $\widehat{k_1}$ and $\widehat{k_2}$ are chosen such that $\Gamma \vdash \{Q_1\}\ \widehat{k_1} \blacktriangleright_{n_1,n_2} \textcircled{1}$ and $\Gamma \vdash \{Q_2\}\ \widehat{k_2} \blacktriangleright_{n_1,n_2} \textcircled{1}$. Our next step is to copy over the branch from the original program, obtaining the following partial instrumented program. In each branch case, we have indicated what the precondition at that location will be during the proof that this program is a valid instrumentation.

$$\begin{aligned}
L_1 : \{Q_1 \vee Q_2\}\ \mathsf{branch\ true} \Rightarrow\ & \{Q_1\}\ \mathsf{branch}\ x \neq \mathsf{nil} \Rightarrow \{Q_1 \wedge x \neq \mathsf{nil}\} \ldots, \\
& \qquad\qquad\quad x = \mathsf{nil} \Rightarrow \{Q_1 \wedge x = \mathsf{nil}\} \ldots\ \mathsf{end}, \\
\mathsf{true} \Rightarrow\ & \{Q_2\}\ \mathsf{branch}\ x \neq \mathsf{nil} \Rightarrow \{Q_2 \wedge x \neq \mathsf{nil}\} \ldots, \\
& \qquad\qquad\quad x = \mathsf{nil} \Rightarrow \{Q_2 \wedge x = \mathsf{nil}\} \ldots\ \mathsf{end\ end}
\end{aligned}$$

Thus, we get four cases, one for each combination of conditions from the two branches. Since the formulas $Q_1 \wedge x \neq \mathsf{nil}$ and $Q_2 \wedge x = \mathsf{nil}$ are both equivalent to false, we can prune those branches with the FALSE rule, obtaining the following.

$$\begin{aligned}
L_1 : \{Q_1 \vee Q_2\}\ \mathsf{branch\ true} \Rightarrow\ & \{Q_1\}\ \mathsf{branch}\ x \neq \mathsf{nil} \Rightarrow \{\mathsf{false}\}\ \mathsf{halt}, \\
& \qquad\qquad\quad x = \mathsf{nil} \Rightarrow \{Q_1 \wedge x = \mathsf{nil}\} \ldots\ \mathsf{end}, \\
\mathsf{true} \Rightarrow\ & \{Q_2\}\ \mathsf{branch}\ x \neq \mathsf{nil} \Rightarrow \{Q_2 \wedge x \neq \mathsf{nil}\} \ldots, \\
& \qquad\qquad\quad x = \mathsf{nil} \Rightarrow \{\mathsf{false}\}\ \mathsf{halt\ end\ end}
\end{aligned}$$

We can then use INST-ASSUME to record facts about n_2, obtaining

$$\begin{aligned}
L_1 : \{Q_1 \vee Q_2\}\ \mathsf{branch\ true} \Rightarrow\ & \{Q_1\}\ \mathsf{branch}\ x \neq \mathsf{nil} \Rightarrow \{\mathsf{false}\}\ \mathsf{halt}, \\
& \qquad\qquad\quad x = \mathsf{nil} \Rightarrow \{Q_1 \wedge x = \mathsf{nil}\} \\
& \qquad\qquad\qquad\qquad\quad \mathsf{assume}(n_2 = 0);\ \ldots\ \mathsf{end}, \\
\mathsf{true} \Rightarrow\ & \{Q_2\}\ \mathsf{branch}\ x \neq \mathsf{nil} \Rightarrow \{Q_2 \wedge x \neq \mathsf{nil}\} \\
& \qquad\qquad\qquad\qquad\quad \mathsf{assume}(n_2 > 0);\ \ldots, \\
& \qquad\qquad\quad x = \mathsf{nil} \Rightarrow \{\mathsf{false}\}\ \mathsf{halt\ end\ end}
\end{aligned}$$

In this case, the use of Inst-Disj just described yields an instrumented program which is equivalent to the program we previously obtained from the simpler and more succinct method of inserting assume() statements with Inst-Assume. This will be the case whenever there are expressions over instrumented variables that are equivalent to each of the original branch conditions (as is the case with the expressions $n_2 = 0$ and $n_2 > 0$ and the branch conditions $x = \mathsf{nil}$ and $x \neq \mathsf{nil}$).

However, there are cases where Inst-Disj is necessary and the simpler method does not yield satisfactory results. This happens when the instrumented variables only allow us to express an under- or over-approximation of the original branch condition. For example, consider the condition $x = y$ in a state satisfying $ls(n, x, y)$. If $n = 0$ in this state, then $x = y$. But if $n > 0$ then x and y can still be equal if the list is cyclic. As such, $n = 0$ is an under-approximation of the condition $x = y$, but we have no corresponding under-approximation for $x \neq y$. An instrumentation of a branch on $x = y$ might then look like the following (we have added the assume() statements on n in a different location, but the procedure is otherwise the same as in the previous example). As before, we mark the inconsistent branch with the precondition {false}.

```
$L_1 : \{ls(n, x, y)\}$ branch true $\Rightarrow$ assume($n = 0$); branch $x = y \Rightarrow \ldots,$
                                                                  $x \neq y \Rightarrow \{\mathsf{false}\}$ halt end,
                              true $\Rightarrow$ assume($n > 0$); branch $x = y \Rightarrow \ldots$
                                                                  $x \neq y \Rightarrow \ldots$ end end
```

In all of these examples, we used Inst-Disj to split on a disjunction that arose naturally from the disjunctive form of the definition of ls. We can also use Inst-Disj to case split on any predicate. Since the standard (non-separating) logical connectives in separation logic are classical in nature, we have the law of excluded middle and thus can always introduce the disjunction $Q \vee \neg Q$ for any formula Q. This then allows us to case split on an arbitrary Q at any point in the instrumented program. For example, we can branch on whether two variables are equal even if such an expression does not appear in the precondition or in the program text.

4.1.2 Properties

We note here a few useful properties of the proof system given in Figure 4.1. Of course soundness is the property in which we are most interested. However as its proof is the most complex, we save it for Section 4.3.

Choice of Instrumentation Variables

The proof system in Figure 4.1 asks us to choose a set V of instrumentation variables which must contain all the variables that appear free in the instrumentation commands. Intuitively, this set need only mention the instrumentation variables that are actually used by the instrumented program. This is captured by the following theorem.

Theorem 19. *If* $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ *then* $\Gamma \vdash \widehat{P} \blacktriangleright_{V'} P$ *for* $V' = (\mathit{fv}(\widehat{P}) - \mathit{fv}(P))$.

Proof. We will show that any derivation of $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ can be transformed into a derivation of $\Gamma \vdash \widehat{P} \blacktriangleright_{V'} P$. The INST-PROG rule ensures that $\mathit{fv}(P) \cap V = \emptyset$ and we proceed to transform the derivation of each $\Gamma \vdash \{\Gamma(l)\}\ \widehat{P}(l) \blacktriangleright_V P(l)$ premise in INST-PROG. The set V only participates in side conditions of rules and is unchanged as we move up the proof tree. We want to show that for each rule, replacing V by V' in the side condition still results in a valid derivation.

To take a representative case, consider the INST-EXISTS rule. We have $x \in V$. We must show that $x \in V'$. Clearly $x \in \mathit{fv}(\widehat{P})$ as $(x := ?\texttt{;}\ \widehat{k})$ is a sub-term of $\widehat{P}$. Then $x \in V'$ provided that $x \notin \mathit{fv}(P)$. But we have that $\mathit{fv}(P) \cap V = \emptyset$, thus $x \in V$ implies $x \notin \mathit{fv}(P)$. The other cases are similar. □

We also have that if V is sufficient to show instrumentation, then any extension of V is also sufficient.

Theorem 20. *If* $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ *then for all* $V' \supseteq V$ *such that* $V' \cap \mathit{fv}(P) = \emptyset$ *we have* $\Gamma \vdash \widehat{P} \blacktriangleright_{V'} P$.

Proof. The proof is by induction on the derivation of $\Gamma \vdash \widehat{P} \blacktriangleright_V P$. For the rule INST-PROG we need to show that $\mathit{fv}(P) \cap V' = \emptyset$ and $\forall l \in \mathit{dom}(P).\ (\Gamma \vdash \{Q\}\ \widehat{P}(l) \blacktriangleright_{V'} P(l))$. The first is given as an assumption, the second is proved by induction on the derivation. Specifically, we show that for all k and $V' \supseteq V$, if $\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k$ holds, then so does $\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_{V'} k$.

Examining the rules in Figure 4.1 we see that only INST-ASSIGN and INST-EXISTS involve conditions on the set of variables V'. For the other rules, our goal will follow immediately from the inductive hypothesis. Suppose that INST-ASSIGN was the last rule applied in the derivation of $\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k$. Then we have $\{Q\}\ x^\tau := e^\tau\ \{Q'\}, \Gamma \vdash \{Q'\}\ \widehat{k} \blacktriangleright_V k$ and $x^\tau \in V$. From the last condition and $V' \supseteq V$ we have $x^\tau \in V'$. The inductive hypothesis gives us $\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_{V'} k$. These last two together with $\{Q\}\ x^\tau := e^\tau\ \{Q'\}$ are then sufficient to apply INST-ASSIGN with V' as the set of instrumentation variables, obtaining $\Gamma \vdash \{Q\}\ (x^\tau := e^\tau\, ;\, \widehat{k}) \blacktriangleright_{V'} k$, which is our goal.

The case for INST-EXISTS is similar, as again the only condition on V is the side condition that $x^\tau \in V$. □

Combined, these theorems indicate that the use of V in the inference system is merely a notational convenience. It could be derived, up to extension, from the free variables of P and P'.

Weakening Γ

For an instrumentation of a given continuation, Γ can always be weakened (this is not the case at the level of programs, however).

Lemma 12. *If* $\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k$ *and* $\forall l.\ \Gamma(l) \Rightarrow \Gamma'(l)$ *then*

$$\Gamma' \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k$$

Proof. We show how to transform a derivation of $\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k$ into a derivation of $\Gamma' \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k$. For all the rules in the derivation except GOTO, we can simply replace Γ by Γ'. The rule will still be valid. For GOTO, which is the only rule in Figure 4.1 that involves a condition on Γ, we make the following change. The GOTO rule is reproduced below.

$$\frac{\Gamma(l) = Q}{\Gamma \vdash \{Q\}\ \mathsf{goto}\ l \blacktriangleright_V \mathsf{goto}\ l}\ \textsc{Goto}$$

As the equality in $\Gamma(l) = Q$ is syntactic equality, any instance of GOTO has the form below.

$$\frac{}{\Gamma \vdash \{\Gamma(l)\}\ \mathsf{goto}\ l \blacktriangleright_V \mathsf{goto}\ l}\ \textsc{Goto}$$

These rule instances are each replaced with the following derivation, which uses our assumption $\Gamma(l) \Rightarrow \Gamma'(l)$.

$$\frac{\Gamma(l) \Rightarrow \Gamma'(l) \qquad \dfrac{}{\Gamma' \vdash \{\Gamma'(l)\}\ \mathsf{goto}\ l \blacktriangleright_V \mathsf{goto}\ l}\ \textsc{Goto}}{\Gamma' \vdash \{\Gamma(l)\}\ \mathsf{goto}\ l \blacktriangleright_V \mathsf{goto}\ l}\ \textsc{Strengthening}$$

□

Over-approximation of Reachable States

The manner in which the preconditions in Figure 4.1 are transformed is reminiscent of Hoare-logic reasoning. And in fact, it is the case that these formulae always over-approximate the reachable states at the corresponding point in the execution of the instrumented program, just as Hoare-style pre- and post-conditions do. We show this now, beginning with the following lemma.

Lemma 13. *Suppose that $\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k$ holds and $(s, h) \models Q$. Then for all s', h', l' we have $\langle \widehat{k}, (s, h)\rangle \underset{\widehat{P}}{\longrightarrow}^{+} \mathbf{goto}(l', (s', h'))$ implies $(s', h') \models \Gamma(l')$.*

The proof is by induction on the derivation of $\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k$ and in each inductive case involves checking that if the instrumented command in the conclusion of a rule takes a single step from a state satisfying the precondition, then the precondition in the premise holds of the post-state. We do not give a full proof here since the proof of soundness also involves checking this property of the rules. For details, see Section 4.3.

We can now show that the preconditions over-approximate the reachable states.

Theorem 21. *If* $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ *and* $(s, h) \models \Gamma(\mathit{initloc}(\widehat{P}))$ *and*

$$\mathbf{goto}(\mathit{initloc}(\widehat{P}), (s, h)) \underset{\widehat{P}}{\longrightarrow}^{+} \mathbf{goto}(l', (s', h'))$$

then $(s', h') \models \Gamma(l')$.

Let $l_0 = \mathit{initloc}(\widehat{P})$. If $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ holds, then we have $\Gamma \vdash \{\Gamma(l_0)\}\ \widehat{P}(l_0) \blacktriangleright_V P(l_0)$. This together with our assumption $(s, h) \models \Gamma(l_0)$ allows us to apply Lemma 13, thus obtaining that $\mathbf{goto}(\mathit{initloc}(\widehat{P}), (s, h)) \underset{\widehat{P}}{\longrightarrow}^{+} \mathbf{goto}(l', (s', h'))$ implies $(s', h') \models \Gamma(l')$, as desired.

Inversion

Since there is only one rule for proving $\Gamma \vdash \widehat{P} \blacktriangleright_V P$, we have the following inversion lemma.

Lemma 14. *If* $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ *then all the following hold*

1. $\mathit{dom}(\widehat{P}) = \mathit{dom}(P)$
2. $\mathit{fv}(P) \cap V = \emptyset$
3. $\mathit{initloc}(\widehat{P}) = \mathit{initloc}(P)$
4. $\forall l \in \mathit{dom}(P).\ (\Gamma \vdash \{\Gamma(l)\}\ \widehat{P}(l) \blacktriangleright_V P(l))$

We also have that all judgments appearing in the proof involve sub-terms of the program P in the position following the $\blacktriangleright$ symbol.

Lemma 15. *If* D *is a sub-derivation of* $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ *with conclusion* $\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k$ *then* k *is a sub-term of* P.

Proof. The proof is by induction on the derivation of $\Gamma \vdash \widehat{P} \blacktriangleright_V P$. We check each rule in the system given in Figures 4.1 and 4.2 and verify that if the conclusion has the form $\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k$ and a premise has the form $\Gamma \vdash \{Q'\}\ \widehat{k}' \blacktriangleright_V k'$ then k' is a sub-term of k. □

Corollary 3. *If D is a sub-derivation of $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ with conclusion $\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k$ then $V \cap \mathit{fv}(k) = \emptyset$.*

Proof. Since $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ holds, we have $V \cap \mathit{fv}(P) = \emptyset$ from Lemma 14. By Lemma 15 we have that k is a sub-term of P. Thus, $\mathit{fv}(k) \subseteq \mathit{fv}(P)$. Combining these facts gives us that $V \cap \mathit{fv}(k) = \emptyset$. □

4.1.3 Derived Rules

We now discuss certain rules which are *derived* in the sense that, given their premises, their conclusion can be constructed by the use of existing rules. Such rules capture common reasoning patterns and thus we will often use them directly in proofs. Often the instrumented program in the conclusion of the rule is equivalent to another, simpler, instrumented program in the sense that they produce sets of execution traces that are stuttering equivalent. In such cases we will note this and adopt the rule with the simplified conclusion. Note that this simplification step is not usually part of the process of generating derived rules. Thus, these are more accurately described as "simplifications of derived rules," however we adopt the term "derived rule" for conciseness.

Case Split with Conditions In the previous section, we repeatedly encountered continuations with the following structure.

$$\begin{aligned} k \overset{\text{def}}{=} \ & \mathsf{branch\ true} \Rightarrow \mathsf{assume}(e_1)\texttt{;}\ \widehat{k_1}, \\ & \qquad \mathsf{true} \Rightarrow \mathsf{assume}(e_2)\texttt{;}\ \widehat{k_2}\ \mathsf{end} \end{aligned}$$

Such a pattern corresponds to the derivation given in Figure 4.5. The code above is equivalent to the following.

$$\begin{aligned} k' \overset{\text{def}}{=} \mathsf{branch}\ & e_1 \Rightarrow \widehat{k_1}, \\ & e_2 \Rightarrow \widehat{k_2}\ \mathsf{end} \end{aligned}$$

To see why, consider the traces of k. These have one of two forms. Either they fit the pattern

$$\langle k, (s,h)\rangle\, \langle(\mathsf{assume}(e_1)\texttt{;}\ \widehat{k_1}), (s,h)\rangle\ T_1$$

where $(s,h) \models e_1$ and T_1 is a trace of $\widehat{k_1}$ starting from s, h, or they are of the form

$$\langle k, (s,h)\rangle\, \langle(\mathsf{assume}(e_2)\texttt{;}\ \widehat{k_2}), (s,h)\rangle\ T_2$$

where $(s,h) \models e_2$ and T_2 is a trace of $\widehat{k_2}$ starting from s, h.

The traces of k' are stuttering equivalent to these with respect to the equivalence relation $\doteq$, which is the equivalence relation on states that allows the current continuation to differ but otherwise requires the states to match (a full definition is given on page 89). The traces of k' have the form

$$\langle k', (s,h)\rangle\, T_1$$

and

$$\langle k', (s,h)\rangle\, T_2$$

These differ from the trace of k only in that the traces of k contain one more repetition of the memory state s, h.

Collecting the premises in the derivation in Figure 4.5 and using the simplified continuation k' as the conclusion gives us the following derived rule.

Inst-Branch

$$\frac{Q \Rightarrow e_1 \vee e_2 \qquad \Gamma \vdash \{Q \wedge e_1\}\ \widehat{k_1} \blacktriangleright_V k \qquad \Gamma \vdash \{Q \wedge e_2\}\ \widehat{k_2} \blacktriangleright_V k}{\Gamma \vdash \{Q\}\ \mathsf{branch}\ e_1 \Rightarrow \widehat{k_1}, e_2 \Rightarrow \widehat{k_2}\ \mathsf{end} \blacktriangleright_V k}$$

This lets us directly branch on pure conditions present in a disjunctive precondition.

$$\dfrac{\dfrac{Q \wedge e_1 \Rightarrow e_1 \qquad \boxed{\Gamma \vdash \{Q \wedge e_1\}\ \widehat{k_1} \blacktriangleright_V k}}{\Gamma \vdash \{Q \wedge e_1\}\ (\mathsf{assume}(e_1)\mathsf{;}\ \widehat{k_1}) \blacktriangleright_V k}\ \text{I-A} \qquad \dfrac{Q \wedge e_2 \Rightarrow e_2 \qquad \boxed{\Gamma \vdash \{Q \wedge e_2\}\ \widehat{k_2} \blacktriangleright_V k}}{\Gamma \vdash \{Q \wedge e_2\}\ (\mathsf{assume}(e_2)\mathsf{;}\ \widehat{k_2}) \blacktriangleright_V k}\ \text{I-A}}{\Gamma \vdash \{(Q \wedge e_1) \vee (Q \wedge e_2)\}\ \mathsf{branch\ true} \Rightarrow \mathsf{assume}(e_1)\mathsf{;}\ \widehat{k_1}, \mathsf{true} \Rightarrow \mathsf{assume}(e_2)\mathsf{;}\ \widehat{k_2}\ \mathsf{end} \blacktriangleright_V k}$$

$$\dfrac{\begin{array}{c}\boxed{Q \Rightarrow e_1 \vee e_2}\\ \vdots \\ Q \Rightarrow (Q \wedge e_1) \vee (Q \wedge e_2)\end{array} \qquad \uparrow}{\Gamma \vdash \{Q\}\ \mathsf{branch\ true} \Rightarrow \mathsf{assume}(e_1)\mathsf{;}\ \widehat{k_1}, \mathsf{true} \Rightarrow \mathsf{assume}(e_2)\mathsf{;}\ \widehat{k_2}\ \mathsf{end} \blacktriangleright_V k}\ \textsc{Str}$$

Figure 4.5: Derivation corresponding to the insertion of a case split on $e_1 \vee e_2$. The premises that become premises of the derived rule are boxed (the other two premises are tautologies). We abbreviate STRENGTHENING as STR and INST-ASSUME as I-A. The unlabeled rule is an instance of INST-DISJ.

Branch Translation We can build on the INST-BRANCH rule given previously to derive a rule that lets us translate branch conditions in one step when the conditions have an exact analogue in terms of instrumentation variables. To take an example, in the case of complete lists of the form $ls(n, x, \mathsf{nil})$—that is, lists of length n starting at x and ending at nil—we have that $ls(n, x, \mathsf{nil}) \wedge n = 0 \Leftrightarrow ls(n, x, \mathsf{nil}) \wedge x = \mathsf{nil}$. Thus, in a state in which we have $ls(n, x, \mathsf{nil})$, knowing that $n = 0$ tells us just as much as knowing that $x = \mathsf{nil}$.

The derivation given in Figure 4.6 forms the basis of the derived rule. We then, as in the previous case, simplify the conclusion. However, the argument that such a simplification is permitted is more complicated in this case. We would like to take the following

$$k \overset{\text{def}}{=} \mathsf{branch}\ e_1 \Rightarrow \mathsf{assume}(e'_1)\mathsf{;}\ \widehat{k_1}, \ldots, e_n \Rightarrow \mathsf{assume}(e'_n)\mathsf{;}\ \widehat{k_n}\ \mathsf{end}$$

and reduce it to the continuation below.

$$k' \overset{\text{def}}{=} \mathsf{branch}\ e'_1 \Rightarrow \widehat{k_1}, \ldots, e'_n \Rightarrow \widehat{k_n}\ \mathsf{end}$$

The problem is that these two continuations are only equivalent for initial states (s, h) in which $(s, h) \models e'_i$ implies $(s, h) \models e_i$.

If this implication holds, then the traces of k have the following form

$$\langle k, (s,h)\rangle \, \langle (\mathsf{assume}(e'_i)\texttt{;}\ \widehat{k_i}), (s,h)\rangle \, T_i$$

where $(s,h) \models e_i$ and $(s,h) \models e'_i$ and T_i is a trace of $\widehat{k_i}$. The traces of k' have the form

$$\langle k', (s,h)\rangle \, T_i$$

where $(s,h) \models e'_i$. If $(s,h) \models e'_i$ implies $(s,h) \models e_i$, then these two sets of traces are related by $\sim_{\doteq}$ (for each trace of k there is a matching trace of k' and vice-versa).

To ensure that the above simplification is always valid then, we require that $Q \wedge e'_i \Rightarrow e_i$. This, combined with the fact that Q is an over-approximation of the reachable states at this point in the execution, ensures that the continuation will only be executing in contexts in which for all s, h we have $(s,h) \models e'_i$ implies $(s,h) \models e_i$ and the replacement is valid. This leaves us with the rule below. Note that since the derivation in Figure 4.6 requires that $(Q \wedge e_i) \Rightarrow e'_i$ and the rule for simplifying the conclusion requires that $(Q \wedge e'_i) \Rightarrow e_i$, this forces the assumption that $(Q \wedge e_i) \Leftrightarrow (Q \wedge e'_i)$ in the final rule.

$$\textsc{Inst-BranchTrans}\quad \frac{(Q \wedge e_i) \Leftrightarrow (Q \wedge e'_i) \qquad \forall i.\ (\Gamma \vdash \{Q \wedge e_i\}\ \widehat{k_i} \blacktriangleright_V k_i)}{\Gamma \vdash \{Q\} \begin{pmatrix} \mathsf{branch}\ e'_1 \Rightarrow \widehat{k_1}, \ldots, \\ e'_n \Rightarrow \widehat{k_n}\ \mathsf{end} \end{pmatrix} \blacktriangleright_V \begin{pmatrix} \mathsf{branch}\ e_1 \Rightarrow \widehat{k_1}, \ldots, \\ e_n \Rightarrow \widehat{k_n}\ \mathsf{end} \end{pmatrix}}$$

Assignment We took as primitive the Inst-Assign rule. Having a succinct rule for updating instrumentation variables is useful, as this operation occurs quite frequently. However, as we will see in this section, this rule is actually derivable from the others. Figure 4.7 gives the derivation for the simpler case where we are inserting the instrumentation command $x := e$ and $x \notin fv(e)$. We can then derive the more general rule with the commonly-used trick of inserting a temporary variable (transforming $x := e$ into $y := e\texttt{;}\ x := y$ where y is a fresh variable).

Essentially, the derivation relies on the fact that we can use the Strengthening rule to reason forward from our precondition Q, obtaining the sequence of implications

$$\forall i\ \dfrac{\dfrac{\boxed{(Q \wedge e_i) \Rightarrow e'_i} \qquad \boxed{\Gamma \vdash \{Q \wedge e_i\}\ \widehat{k_i} \blacktriangleright_V k_i}}{\Gamma \vdash \{Q \wedge e_i\}\ (\mathsf{assume}(e'_i)\texttt{;}\ \widehat{k_i}) \blacktriangleright_V k_i}\ \textsc{Inst-Assume}}{\Gamma \vdash \{Q\} \begin{pmatrix} \mathsf{branch}\ e_1 \Rightarrow \mathsf{assume}(e'_1)\texttt{;}\ \widehat{k_1}, \ldots, \\ e_n \Rightarrow \mathsf{assume}(e'_n)\texttt{;}\ \widehat{k_n}\ \mathsf{end} \end{pmatrix} \blacktriangleright_V \begin{pmatrix} \mathsf{branch}\ e_1 \Rightarrow k_1, \ldots, \\ e_n \Rightarrow k_n\ \mathsf{end} \end{pmatrix}}\ \textsc{Branch}$$

Figure 4.6: Derivation corresponding to the translation of branch conditions into conditions on instrumentation variables. In the rule labeled $\forall i$, the premise holds for each value of i. The premises that become premises of the derived rule are boxed. We require that they hold for each $i \in \{1, \ldots, n\}$.

$Q \Rightarrow \exists x.\ Q \Rightarrow \exists x'.\ Q[x'/x]$. This allows us to perform the quantification of the previous value of x that occurs in the forward reasoning rule for $x := e$ in Hoare logic. We then note that, since our semantics of expressions is total, if e does not contain x then $\exists x.\ x = e$ is a tautology, allowing us to conclude

$$(\exists x'.\ Q[x'/x]) \wedge (\exists x.\ x = e)$$

Since x is not free in $\exists x'.\ Q[x'/x]$, we can extend the scope of the quantifier on x, obtaining

$$\exists x.\ (\exists x'.\ Q[x'/x]) \wedge x = e$$

We can the use the Inst-Exists rule to add the command $x := ?$ and obtain the precondition

$$(\exists x'.\ Q[x'/x]) \wedge x = e$$

which allows us to insert $\mathsf{assume}(x = e)$ with the Inst-Assume rule.

The derivation in Figure 4.7 also makes use of the fact that $\{Q\}\ x := e\ \{Q'\}$ implies $\exists x'.\ \big(Q[x'/x] \wedge (x = e[x'/x])\big) \Rightarrow Q'$. This holds because $\exists x'.\ \big(Q[x'/x] \wedge (x = e[x'/x])\big)$ is the strongest post-condition of $x := e$ with respect to the precondition Q. If $x \notin fv(e)$ then $e[x'/x] = e$ and the strongest post-condition is simply $\exists x'.\ Q[x'/x] \wedge x = e$.

Collecting the premises and side-conditions from the derivation in Figure 4.7 we obtain the following derived rule for assignments (note that we have also simplified

$x \notin \mathit{fv}(Q[x'/x], e)$ to $x \notin \mathit{fv}(e)$ since $Q[x'/x]$ cannot contain x).

$$\frac{\{Q\}\; x := e\; \{Q'\} \qquad \Gamma \vdash \{Q'\}\; \widehat{k} \blacktriangleright_V k}{\Gamma \vdash \{Q\}\; (x := ?\texttt{;}\ \mathsf{assume}(x = e)\texttt{;}\ \widehat{k}) \blacktriangleright_V k}\; x \in V, x \notin \mathit{fv}(e)$$

We can then prove that if $x \notin \mathit{fv}(e)$ then $(x := ?\texttt{;}\ \mathsf{assume}(x = e)\texttt{;}\ \widehat{k})$ is stuttering equivalent to $(x := e\texttt{;}\ \widehat{k})$. Let k be the first continuation and k' be the second. The traces of k have the form

$$\langle k, (s, h)\rangle\, \langle(\mathsf{assume}(x = e)\texttt{;}\ \widehat{k}), (s', h)\rangle\, T$$

where $s' = s[x \rightarrow v]$ for some v and $(s', h) \models (x = e)$ and T is a trace of $\widehat{k}$ starting from (s', h). The traces of k' have the form

$$\langle k', (s, h)\rangle\, \langle\widehat{k}, (s[x \rightarrow [\![e]\!]\, s], h)\rangle\, T$$

The traces are stuttering equivalent (with respect to $\doteq$) provided we can show that $s' = s[x \rightarrow [\![e]\!]\, s]$. The fact that $(s', h) \models (x = e)$ implies $s'(x) = [\![e]\!]\, s$. Combined with the fact that $s' = s[x \rightarrow v]$, this tells us that $v = [\![e]\!]\, s$ and thus $s' = s[x \rightarrow [\![e]\!]\, s]$ as desired.

The above argument allows us to simplify the instrumented continuation in the conclusion, obtaining the following rule.

INST-ASSIGN-NOTFREE

$$\frac{\{Q\}\; x := e\; \{Q'\} \qquad \Gamma \vdash \{Q'\}\; \widehat{k} \blacktriangleright_V k}{\Gamma \vdash \{Q\}\; (x := e\texttt{;}\ \widehat{k}) \blacktriangleright_V k}\; x \in V, x \notin \mathit{fv}(e)$$

This then gives us all the machinery necessary to replicate the INST-ASSIGN rule. Suppose we had the proof system in Figure 4.1, but without the INST-ASSIGN rule and we wanted to insert the assignment $x := e$, where x is an instrumentation variable. Then we could select an instrumentation variable y which is not otherwise used (by Theorem 20 this can always be done) and insert the commands $y := e\texttt{;}\ x := y$ using the INST-ASSIGN-NOTFREE rule.

$$\dfrac{\dfrac{((\exists x'.\,Q[x'/x]) \land x = e) \Rightarrow x = e \qquad \dfrac{\begin{array}{c}\boxed{\{Q\}\ x := e\ \{Q'\}} \quad \boxed{x \notin \mathit{fv}(e)} \\ \vdots \\ ((\exists x'.\,Q[x'/x]) \land x = e) \Rightarrow Q'\end{array} \qquad \boxed{\Gamma \vdash \{Q'\}\ \widehat{k} \blacktriangleright_V k}}{\Gamma \vdash \{(\exists x'.\,Q[x'/x]) \land x = e\}\ \widehat{k} \blacktriangleright_V k}}{\Gamma \vdash \{(\exists x'.\,Q[x'/x]) \land x = e\}\ (\mathsf{assume}(x = e)\ \texttt{;}\ \widehat{k}) \blacktriangleright_V k}\ \text{I-A}}{\Gamma \vdash \{\exists x.\,(\exists x'.\,Q[x'/x]) \land x = e\}\ (x := ?\texttt{;}\ \mathsf{assume}(x = e)\ \texttt{;}\ \widehat{k}) \blacktriangleright_V k}\ \boxed{x \in V}\ \text{I-E}$$

$$\dfrac{\begin{array}{c}\dfrac{x' \notin \mathit{fv}(Q) \qquad \boxed{x \notin \mathit{fv}(e)}}{\vdots} \\ Q \Rightarrow \exists x.\,(\exists x'.\,Q[x'/x]) \land x = e\end{array} \qquad \uparrow}{\Gamma \vdash \{Q\}\ (x := ?\texttt{;}\ \mathsf{assume}(x = e)\ \texttt{;}\ \widehat{k}) \blacktriangleright_V k}$$

Figure 4.7: Derivation of the INST-ASSIGN rule for the case where $x \notin \mathit{fv}(e)$. The formulas and conditions that become premises and side conditions in the derived rule are boxed. The unboxed formulas can always be made to hold, either because they are tautologies or, in the case of $x' \notin \mathit{fv}(Q)$ because we get to choose x' when constructing the derivation. I-A stands for INST-ASSUME, I-E stands for INST-EXISTS. All other rules are instances of STRENGTHENING.

4.2 Example

Before examining in more detail the theory behind instrumented programs, we first consider a concrete example. Consider the C program in Figure 4.8. This program advances a pointer r through an ordered binary tree, searching for the value v. It returns 1 if the value is found and 0 otherwise. Suppose we want to verify that this program terminates.

The usual method for showing this is to produce a *ranking function*, which is a function from program states to some well-founded set (often a bounded subset of the integers). For programs not involving the heap, these ranking functions can be given as functions of the program variables. However, for programs that manipulate heap-based data structures, these functions may involve properties of the heap.

```
int mem(TreePointer r, int v) {
  int u;

  while(r != 0) {
    u = r->data;
    if (u == v)
      return 1;
    else if (u < v)
      r = r->right;
    else
      r = r->left;
  }
  return 0;
}
```

Figure 4.8: C code implementing a membership query for an ordered binary tree.

This is the case for our example. We cannot write a ranking function for the loop that is given solely in terms of program variables. The quantity that is decreasing at each iteration is the size of the sub-tree at r, which does not have an explicit representation in the program. As such, standard termination tools cannot be applied to this example and we might think that any method for constructing a ranking function for this example would have to be heap-aware.

What we show in this section (and in the thesis in general) is that by constructing an appropriate instrumented version of the code, we can provide explicit information regarding the counts involved in the termination argument. This provides a standard termination tool with the components it needs to construct a ranking function and allows the rank function synthesis to be done with no knowledge of the underlying heap-based data structures.

We begin by translating the C program into our program format. The result of this translation is given in Figure 4.9. We include a variable "*return*" that models the return value of the function.

```
loop : ① branch $r = \mathsf{nil} \Rightarrow$ ② $\mathit{return} := 0$; halt,
                $r \neq \mathsf{nil} \Rightarrow$ ③ $u := r.\mathsf{data}$;
                                ④ branch $u = v \Rightarrow$ ⑤ $\mathit{return} := 1$; halt,
                                         $u < v \Rightarrow$ ⑥ $r := r.\mathsf{right}$; goto loop,
                                         $u > v \Rightarrow$ ⑦ $r := r.\mathsf{left}$; goto loop
                                end
       end
```

Figure 4.9: The program from Figure 4.8 translated into our program notation, with control points numbered.

To produce the instrumented version, we need a means of describing the contents of the heap. This is provided by the following definition of binary trees. Here, n represents the number of nodes in the tree.

$$\begin{aligned}
&\mathit{tree}(n, r) \equiv \\
&\qquad (n = 0 \land r = \mathsf{nil} \land \mathbf{emp}) \\
&\quad \lor\ (n > 0 \land \exists n_1, n_2.\ (n = n_1 + n_2 + 1) \land \\
&\qquad\qquad\qquad (\exists lc, rc, m.\ (r \mapsto [\mathsf{left} : lc, \mathsf{right} : rc, \mathsf{data} : m]) * \\
&\qquad\qquad\qquad\quad \mathit{tree}(n_1, lc) * \mathit{tree}(n_2, rc)))
\end{aligned}$$

An instrumented version of the search program is given in Figure 4.10. The loop invariant is $\mathit{tree}(n, r) * \mathsf{true}$, which indicates that there is a binary search tree at r consisting of n separate nodes (where a "node" is a pointer cell of the form $x \mapsto [\mathsf{left} : a, \mathsf{right} : b, \mathsf{data} : c]$). The "$*$ true" portion indicates that the heap may also contain other cells. For a more complete analysis of this program, we would want to define a predicate describing a "tree with a hole" (similar to the approach taken in Calcagno et al. [2005]) in order to track these other cells more precisely, as this information is needed to conclude that the heap still contains a tree when the function returns.

We have annotated the instrumented program with invariants at key locations, showing the value of Q that would be used in the proof of $\Gamma \vdash \{Q\}\ \widehat{k}\ \blacktriangleright_V\ k$ at that point.

```
loop :  $\{\mathit{tree}(n, r) * \mathsf{true}\}$
        ❶ branch
            $n = 0 \Rightarrow$ ❷ $\mathit{return} := 0$; halt
            $n > 0 \Rightarrow$ ❸
                $\{\exists n_1, n_2.\ Q\}$ $n_1 := ?$; $n_2 := ?$;
                $\{Q\}$ assume($n = n_1 + n_2 + 1$);
                $\{Q\}$ $u := r$.data;
                ❹ branch
                    $u = v \Rightarrow$ ❺ $\mathit{return} := 1$; halt,
                    $u < v \Rightarrow$ ❻ $r := r$.left;
                                 $\{\mathit{tree}(n_1, r) * \mathsf{true}\}$ $n := n_1$;
                                 $\{\mathit{tree}(n, r) * \mathsf{true}\}$ goto loop
                    $u > v \Rightarrow$ ❼ $r := r$.right;
                                 $\{\mathit{tree}(n_2, r) * \mathsf{true}\}$ $n := n_2$;
                                 $\{\mathit{tree}(n, r) * \mathsf{true}\}$ goto loop
                end
        end
```

$$Q \overset{\text{def}}{=} \exists lc, rc, m.\ (r \mapsto [\mathsf{left} : lc, \mathsf{right} : rc, \mathsf{data} : m] * \\ \mathit{tree}(n_1, lc) * \mathit{tree}(n_2, rc) * \mathsf{true}) \wedge (n = n_1 + n_2 + 1)$$

Figure 4.10: Instrumented version of the program in Figure 4.9.

The main branch on $r = \mathsf{nil}$ is transformed into an equivalent branch on $n = 0$ by the INST-BRANCHTRANS derived rule from Section 4.1.3. Other commands are added via the INST-ASSUME, INST-EXISTS, and INST-ASSIGN rules.

The program first branches on the instrumentation variable n, which represents the number of nodes in the tree rooted at r. In the case where the tree is empty, we return. In the case where the tree is non-empty, it is expanded into its left and right child, whose sizes summed plus one equals n. When we reach the end of this case, having advanced r to the appropriate child, the instrumentation command $n := n_i$ is inserted (where $i = 1$ or $i = 2$ depending on the child that was chosen). This updates n to contain the number of nodes in the sub-tree that is now pointed to by r.

To show termination, we can focus on the changes to n. We see that in all paths through the loop, either we halt or n strictly decreases. As n is bounded below by 0, this ensures termination of the loop.

Note that the commands $n_1 := ?$, $n_2 := ?$, and $\mathsf{assume}(n = n_1 + n_2 + 1)$ have the effect of ensuring that, regardless of whether the left child (with size n_1) or the right child (size n_2) is chosen, the size of the tree at r decreases. The non-deterministic choice commands assign new, arbitrary values to n_1 and n_2 and then the assume statement ensures that only values that satisfy the relationship between the sizes are considered (the assume allows us to disregard executions where non-satisfactory values of n_1 and n_2 are chosen).

If the assume statement were not present, the program in Figure 4.10 would still be a valid instrumentation according to the rules in Figure 4.1. However, it would have executions that we know are not possible (namely, executions where n_1 and n_2 do not satisfy $n = n_1 + n_2 + 1$). These extra paths must be considered by subsequent analyses and, in this case, the absence of the constraint $n = n_1 + n_2 + 1$ would prevent a termination analysis from showing that the instrumented program terminates.

4.2.1 Alternate Size Measures

We just presented a treatment of trees where the notion of size corresponded to the number of nodes in the tree. Trees also admit other notions of size—tree height, for example—

and this is true of most data structures. Even singly-linked lists of integers admit multiple notions of size. One may be interested in tracking the length of the list, the maximal value contained in the list, or the sum of all values contained in the list, to name just a few. The rules presented in Figure 4.1 permit reasoning about any of these notions of size. Any quantity whose update relation can be represented using the expression language can be tracked by inserting instrumentation commands in the manner discussed previously.

As an example, if we want to track the height of a tree, we could use the definition below.

$$\begin{aligned}
\mathit{treeh}(h, r) \equiv{} & (h = 0 \wedge r = \mathsf{nil}) \\
& \vee (h > 0 \wedge \exists h_1, h_2, m.\ (h_1 < h) \wedge (h_2 < h) \wedge (h = h_1 + 1 \vee h = h_2 + 1) \\
& \qquad \exists lc, rc.\ r \mapsto [\mathsf{left} : lc, \mathsf{right} : rc, \mathsf{data} : m] \\
& \qquad\quad * \mathit{treeh}(h_1, lc) * \mathit{treeh}(h_2, rc))
\end{aligned}$$

Here we use the constraint $(h_1 < h) \wedge (h_2 < h) \wedge (h = h_1 + 1 \vee h = h_2 + 1)$ to ensure that if h_1 and h_2 are the heights of the left and right sub-trees, then h is the height of the full tree. If our expression language had a function max of type $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ that returned the greater of its two arguments, then we could represent this constraint more succinctly as $h = max(h_1, h_2) + 1$.

We can also specify more abstract notions of size. For example, below is the same tree definition, but with argument a representing an abstract notion of size, rather than a particular size measure.

$$\begin{aligned}
\mathit{treea}(a, r) \equiv{} & (a = 0 \wedge r = \mathsf{nil}) \\
& \vee (a > 0 \wedge \exists a_1, a_2.\ (a_1 < a) \wedge (a_2 < a) \\
& \qquad \exists lc, rc.\ r \mapsto [\mathsf{left} : lc, \mathsf{right} : rc] \\
& \qquad\quad * \mathit{treea}(a_1, lc) * \mathit{treea}(a_2, rc))
\end{aligned}$$

The specific size measures discussed previously—number of nodes and height—would both satisfy this definition. That is, if treeh is the tree predicate that tracks height and tree is the predicate that specifies the number of nodes and treea is the definition above, then we have

$$\begin{aligned} \mathit{tree}(h, r) &\Rightarrow \mathit{treea}(h, r) \\ \mathit{treeh}(h, r) &\Rightarrow \mathit{treea}(h, r) \end{aligned}$$

This follows from the fact that the update relation for tree is contained in the update relation for treea, and similarly for treeh. More specifically, we can view the pure constraint on sizes as a relation between "size of the entire tree," "size of the left sub-tree," and "size of the right sub-tree." If we then write s, s_l, and s_r for these quantities, thus unifying our variable notation, we get an update relation of $s = s_l + s_r + 1$ for tree and $(s_l < s) \wedge (s_r < s)$ for treea. The fact that for $s_l, s_r \geq 0$ we have $(s = s_l + s_r + 1) \Rightarrow (s_l < s) \wedge (s_r < s)$ is then the main step in justifying the first implication given above.

To consider another example, below is the definition of a predicate for a list of integers where the notion of size is the sum of the integers in the list. Note that termination of a traversal routine could be established for such a notion of size only if the list contains solely positive elements.

$$\begin{aligned} &\mathit{ls}(n, \mathit{first}, \mathit{next}) \equiv \\ &\qquad (\mathbf{emp} \wedge \mathit{first} = \mathit{next} \wedge n = 0) \\ &\quad \vee \big(\exists z.\, \big((\mathit{first} \mapsto [\mathsf{next} : z, \mathsf{data} : d]) * \mathit{ls}(n', z, \mathit{next})\big) \wedge n = n' + d\big) \end{aligned}$$

This is also an example of a situation where there is not a condition on the size that uniquely determines which case of the definition applies. If we have $\mathit{ls}(n, a, b)$ and $n > 0$, then the definition above specifies that the list must be non-empty. However, if $n = 0$, then either case of the definition may hold.

4.3 Soundness

In this section, we prove that instrumented programs meeting our criteria simulate the original program. This takes us half-way to numeric abstractions. In Section 4.4, we complete the formal development by showing how numeric abstractions can be extracted from instrumented programs.

Definition 31. *Let $R^{V,\Gamma}$ be the relation on execution states defined as follows. We use the notation $\widetilde{V}$ to abbreviate the set Vars $- V$.*

$$
\begin{array}{rcll}
\mathbf{goto}(l,(s,h)) & R^{V,\Gamma} & \mathbf{goto}(\widehat{l},(\widehat{s},\widehat{h})) & \textit{iff}\ \big((\widehat{s},\widehat{h}) \models \Gamma(l)\big) \wedge (l = \widehat{l}) \\
 & & & \qquad\qquad \wedge\, (s =_{\widetilde{V}} \widehat{s}) \wedge (h = \widehat{h}) \\
\langle k,(s,h)\rangle & R^{V,\Gamma} & \langle \widehat{k},(\widehat{s},\widehat{h})\rangle & \textit{iff}\ \exists Q.\ \big(\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k\big) \wedge \big((\widehat{s},\widehat{h}) \models Q\big) \\
 & & & \qquad\qquad \wedge\, (s =_{\widetilde{V}} \widehat{s}) \wedge (h = \widehat{h}) \\
\mathbf{final}(s,h) & R^{V,\Gamma} & \mathbf{final}(\widehat{s},\widehat{h}) & \textit{iff}\ (s =_{\widetilde{V}} \widehat{s}) \wedge (h = \widehat{h}) \\
\mathbf{error} & R^{V,\Gamma} & \mathbf{error} &
\end{array}
$$

We can now state the main theorem associated with the proof system in Figure 4.1. This states that, if $\widehat{P}$ is an instrumented version of P according to the proof rules in Figures 4.1 and 4.2, then P with initial states satisfying $\Gamma(\mathit{initloc}(P))$ is simulated by $\widehat{P}$ with the same set of initial states.

Theorem 22. *(Soundness) Let $Q_0 = \Gamma(\mathit{initloc}(P))$. Then $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ implies $(\!(P \mid Q_0)\!) \underset{\sim}{\sqsubset}_{R^{V,\Gamma},=_{\widetilde{V}}} (\!(\widehat{P} \mid Q_0)\!)$.*

Proof. We must show that $R^{V,\Gamma}$ satisfies the conditions in Definition 29. We consider each condition in order.

goal *(Initial States Related)*:

By Definition 14 we have that the initial states I of $(\!(P \mid Q_0)\!)$ are

$$I = \big\{\mathbf{goto}(l_0,(s,h)) \;\big|\; (l_0 = \mathit{initloc}(P)) \wedge (s,h) \models Q_0\big\}$$

and the initial states $\widehat{I}$ of $(\!(\widehat{P} \mid Q_0)\!)$ are

$$\widehat{I} = \big\{\mathbf{goto}(l_0,(s,h)) \;\big|\; (l_0 = \mathit{initloc}(\widehat{P})) \wedge (s,h) \models Q_0\big\}$$

We must show that $\forall\gamma \in I.\ \exists\widehat{\gamma} \in \widehat{I}.\ \gamma\ R^{V,\Gamma}\ \widehat{\gamma}$. Consider $\gamma \in I$. We have that $\gamma = \mathbf{goto}(l_0,(s,h))$ where $l_0 = \mathit{initloc}(P)$ and $(s,h) \models Q_0$. Since $Q_0 = \Gamma(l_0)$ we have $(s,h) \models \Gamma(l_0)$. By our definition of $R^{V,\Gamma}$, we then have the following.

$$\mathbf{goto}(l_0,(s,h))\ R^{V,\Gamma}\ \mathbf{goto}(l_0,(s,h))$$

By Lemma 14 we have $initloc(P) = initloc(\widehat{P})$, thus we have that $\mathbf{goto}(l_0, (s, h)) \in \widehat{I}$, completing the proof of this case.

goal *($=_{\widetilde{V}}$-equivalent)*:

$$\forall \gamma_1, \gamma_2.\ (\gamma_1\ R^{V,\Gamma}\ \gamma_2) \Rightarrow (\gamma_1 =_{\widetilde{V}} \gamma_2)$$

This follows immediately from our definition of $R^{V,\Gamma}$ and the definition of the $=_{\widetilde{V}}$ relation.

goal *(P Transitions Match)*: If $\gamma\ R^{V,\Gamma}\ \widehat{\gamma}$ and $\gamma \xrightarrow[P]{} \gamma'$ then one of the following holds

1. *($\widehat{P}$ Matches)* $\widehat{\gamma} \xrightarrow[P]{} \widehat{\gamma}\,'$ and $\gamma'\ R^{V,\Gamma}\ \widehat{\gamma}\,'$
2. *(P Stutters)* $(\gamma'\ R^{V,\Gamma}\ \widehat{\gamma})$ and $(rankt(\gamma', \widehat{\gamma}) < rankt(\gamma, \widehat{\gamma}))$
3. *($\widehat{P}$ Stutters)* $\widehat{\gamma} \xrightarrow[\widehat{P}]{} \widehat{\gamma}\,'$ and $\gamma\ R^{V,\Gamma}\ \widehat{\gamma}\,'$ and $rankl(\widehat{\gamma}\,', \gamma, \gamma') < rankl(\widehat{\gamma}, \gamma, \gamma')$.

Since $\gamma \xrightarrow[P]{} \gamma'$ we know that γ either has the form $\mathbf{goto}(l, (s, h))$ or $\langle k, (s, h)\rangle$.

Goto State Suppose it has the form $\mathbf{goto}(l, (s, h))$. Then by the definition of $R^{V,\Gamma}$, the state $\widehat{\gamma}$ must have the form $\mathbf{goto}(\widehat{l}, (\widehat{s}, \widehat{h}))$ with $(\widehat{s}, \widehat{h}) \models \Gamma(l)$ and $l = \widehat{l}$ and $s =_{\widetilde{V}} \widehat{s}$ and $h = \widehat{h}$. We have from the definitions of $\xrightarrow[P]{}$ and $\xrightarrow[\widehat{P}]{}$ that

$$\mathbf{goto}(l, (s, h)) \xrightarrow[P]{} \langle P(l), (s, h)\rangle$$

and

$$\mathbf{goto}(\widehat{l}, (\widehat{s}, \widehat{h})) \xrightarrow[\widehat{P}]{} \langle \widehat{P}(\widehat{l}), (\widehat{s}, \widehat{h})\rangle$$

Since $l = \widehat{l}$, the second statement is equivalent to

$$\mathbf{goto}(\widehat{l}, (\widehat{s}, \widehat{h})) \xrightarrow[\widehat{P}]{} \langle \widehat{P}(l), (\widehat{s}, \widehat{h})\rangle$$

We will show that condition 1 holds ($\widehat{P}$ matches). This corresponds to the statement below.

$$\langle P(l), (s, h)\rangle\ R^{V,\Gamma}\ \langle \widehat{P}(l), (\widehat{s}, \widehat{h})\rangle$$

This follows from the conclusions of Lemma 14. We already have that $s =_{\widetilde{V}} \widehat{s}$ and $h = \widehat{h}$ and $(\widehat{s}, \widehat{h}) \models \Gamma(l)$. Lemma 14 gives us that $\Gamma \vdash \{\Gamma(l)\}\ \widehat{P}(l) \blacktriangleright_V P(l)$, which is the last condition needed to establish that the states are $R^{V,\Gamma}$-related.

Intermediate State Now we consider the case where $\widehat{\gamma}$ has the form $\langle \widehat{k}, (\widehat{s}, \widehat{h})\rangle$. From the definition of $R^{V,\Gamma}$ for states of this form, we have that there exists a Q such that the following hold.

(Assumption 1)	$\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k$
(Assumption 2)	$(\widehat{s}, \widehat{h}) \models Q$
(Assumption 3)	$s =_{\widetilde{V}} \widehat{s}$
(Assumption 4)	$h = \widehat{h}$

We will show that for all choices of $k, s, h, \widehat{k}, \widehat{s}, \widehat{h}$ consistent with these assumptions, one of the goal conditions holds (either $\widehat{P}$ matches, P stutters, or $\widehat{P}$ stutters). The proof is by induction on the derivation of $\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k$ with one case for each rule in Figure 4.1. The induction is required to handle the STRENGTHENING rule. Figure 4.11 summarizes the variables used throughout this proof.

In the cases where either P or $\widehat{P}$ stutters, we must also show that a ranking function decreases, in order to rule out the possibility of an infinite sequence of states being matched by a single state (and thus infinite traces being matched by finite traces). The ranking function in this case will simply be the size of the continuation k in a state of the form $\langle k, (s, h)\rangle$ and 0 in the case of **error** or $\mathbf{final}(s, h)$. Formally, we have the following definitions for rankt and rankl, where $\mathit{size}(k)$ represents the number of nodes in the abstract

$$\begin{tikzcd}
\overset{\gamma}{\langle k, (s, h)\rangle} \arrow[r, "P"'] \arrow[d, dash, "R^{V,\Gamma}"'] \arrow[dr, dash, dashed] & \gamma' \arrow[d, dash, dashed, "R^{V,\Gamma}"] \arrow[dl, dash, dashed] \\
\overset{\widehat{\gamma}}{\langle \widehat{k}, (\widehat{s}, \widehat{h})\rangle} \arrow[r, "\widehat{P}"'] & \widehat{\gamma}'
\end{tikzcd}$$

Figure 4.11: Guide to variable names used throughout the proof of Theorem 22. In each case of the proof, our goal is to show that one of the dashed relation lines exists.

syntax tree for k.

$$\begin{aligned}
\mathit{rankt}(\langle k, (s, h)\rangle, \widehat{\gamma}) &= \mathit{size}(k) \\
\mathit{rankt}(\mathbf{error}, \widehat{\gamma}) &= 0 \\
\mathit{rankt}(\mathbf{final}(s, h), \widehat{\gamma}) &= 0
\end{aligned}$$

$$\begin{aligned}
\mathit{rankl}(\langle \widehat{k}, (\widehat{s}, \widehat{h})\rangle, \gamma, \gamma') &= \mathit{size}(\widehat{k}) \\
\mathit{rankl}(\mathbf{error}, \gamma, \gamma') &= 0 \\
\mathit{rankl}(\mathbf{final}(s, h), \gamma, \gamma') &= 0
\end{aligned}$$

CASE $\left(\dfrac{\textsc{Halt}}{\Gamma \vdash \{Q\}\ \mathsf{halt} \blacktriangleright_V \mathsf{halt}}\right)$:

In this case, $k = \mathsf{halt}$ and $\widehat{k} = \mathsf{halt}$ and $\gamma' = \mathbf{final}(s, h)$. Since $\widehat{k} = \mathsf{halt}$, we have that $\langle \widehat{k}, (\widehat{s}, \widehat{h})\rangle \underset{\widehat{P}}{\longrightarrow} \mathbf{final}(\widehat{s}, \widehat{h})$. It remains to show that $\mathbf{final}(s, h)\ R^{V,\Gamma}\ \mathbf{final}(\widehat{s}, \widehat{h})$.

This follows from *(Assumption 3)*, *(Assumption 4)*, and the definition of $R^{V,\Gamma}$. Thus, we have shown that $\widehat{P}$ can match the transition.

CASE $\left(\dfrac{\textsc{Abort}}{\Gamma \vdash \{Q\}\ \mathsf{abort} \blacktriangleright_V \mathsf{abort}}\right)$:

In this case, $k = \mathsf{abort}$ and $\widehat{k} = \mathsf{abort}$. Thus, $\gamma' = \mathbf{error}$. We have immediately from the definition of $\underset{\widehat{P}}{\longrightarrow}$ that $\langle \mathsf{abort}, (\widehat{s}, \widehat{h})\rangle \underset{\widehat{P}}{\longrightarrow} \mathbf{error}$. We have that $\mathbf{error}\ R^{V,\Gamma}\ \mathbf{error}$ by the definition of $R^{V,\Gamma}$ for final states. Thus, we have shown that $\widehat{P}$ can match the transition.

CASE $\left(\begin{array}{c}\text{GOTO}\\ \dfrac{\Gamma(l) = Q}{\Gamma \vdash \{Q\}\ \mathsf{goto}\ l \blacktriangleright_V \mathsf{goto}\ l}\end{array}\right)$:

This is very similar to the halt case. We have that $k = \mathsf{goto}\ l$ and $\widehat{k} = \mathsf{goto}\ l$. By the definition of $\underset{P}{\longrightarrow}$ we have $\langle k, (s, h)\rangle \underset{P}{\longrightarrow} \mathbf{goto}(l, (s, h))$ and $\langle \widehat{k}, (\widehat{s}, \widehat{h})\rangle \underset{\widehat{P}}{\longrightarrow} \mathbf{goto}(l, (\widehat{s}, \widehat{h}))$. We must show that $\mathbf{goto}(l, (s, h))\ R^{V,\Gamma}\ \mathbf{goto}(l, (\widehat{s}, \widehat{h}))$ which requires showing that $s =_{\widetilde{V}} \widehat{s}$, $h = \widehat{h}$, and $(\widehat{s}, \widehat{h}) \models \Gamma(l)$. The first two are exactly *(Assumption 3)* and *(Assumption 4)*. The last follows from *(Assumption 2)* by the premise of this rule, which states that $\Gamma(l) = Q$. Thus, $\widehat{P}$ matches the transition.

CASE $\left(\begin{array}{c}\text{COMMAND}\\ \dfrac{\{Q\}\ c\ \{Q'\} \qquad \Gamma \vdash \{Q'\}\ \widehat{k} \blacktriangleright_V k}{\Gamma \vdash \{Q\}\ (c\,;\widehat{k}) \blacktriangleright_V (c\,;k)}\end{array}\right)$:

We have from *(Assumption 4)* that $h = \widehat{h}$. From the definition of $\underset{P}{\longrightarrow}$, we have the transition $\langle (c\,;k), (s, h)\rangle \underset{P}{\longrightarrow} \gamma$ where either

$$\gamma = \mathbf{error}$$

or

$$\gamma = \langle k, (s', h')\rangle \wedge (s', h') \in [\![c]\!]\,(s, h)$$

For the error case, we apply Corollary 3 to obtain $V \cap fv(c) = \emptyset$ and thus $fv(c) \subseteq \widetilde{V}$. This together with *(Assumption 3)* allows us to apply Lemma 3 and obtain $\mathbf{error} \in [\![c]\!]\,(\widehat{s}, \widehat{h})$ and thus $\langle (c\,;\widehat{k}), (\widehat{s}, \widehat{h})\rangle \underset{\widehat{P}}{\longrightarrow} \mathbf{error}$. This completes this case since $\mathbf{error}\ R^{V,\Gamma}\ \mathbf{error}$.

For the non-error case, we apply Corollary 3 to obtain $fv(c) \subseteq \widetilde{V}$. This and *(Assumption 3)* allows us to apply Lemma 2, which gives us an $\widehat{s}'$ such that $(\widehat{s}', h') \in [\![c]\!]\,(\widehat{s}, h)$ and $s' =_{\widetilde{V}} \widehat{s}'$. The semantics of continuations then gives us that $\langle (c\,;\widehat{k}), (\widehat{s}, h)\rangle \underset{\widehat{P}}{\longrightarrow} \langle \widehat{k}, (\widehat{s}', h')\rangle$.

Applying our equality $h = \widehat{h}$ to this transition we then have $\langle (c\,\boldsymbol{;}\,\widehat{k}), (\widehat{s}, \widehat{h})\rangle \xrightarrow[\widehat{P}]{} \langle \widehat{k}, (\widehat{s}\,', h')\rangle$. Our goal is to show that $\langle k, (s', h')\rangle\ R^{V,\Gamma}\ \langle \widehat{k}, (\widehat{s}\,', h')\rangle$. We have shown one condition of $R^{V,\Gamma}$, namely that $s' =_{\widetilde{V}} \widehat{s}\,'$. The condition on heaps in this case is $h' = h'$, which is immediate. It remains to show that $(\widehat{s}\,', h') \models Q'$ and $\Gamma \vdash \{Q'\}\ \widehat{k} \blacktriangleright_V k$.

From *(Assumption 2)* and $(\widehat{s}\,', h') \in [\![c]\!]\,(\widehat{s}, \widehat{h})$ and $\{Q\}\ c\ \{Q'\}$ we have $(\widehat{s}\,', h') \models Q'$. From the second premise of the rule under consideration we have $\Gamma \vdash \{Q'\}\ \widehat{k} \blacktriangleright_V k$. These were the only remaining conditions, so we have shown that $\widehat{P}$ can match P's transition.

$$\textbf{CASE}\ \left(\begin{array}{l}\textsc{Strengthening}\\ \dfrac{Q \Rightarrow Q' \qquad \Gamma \vdash \{Q'\}\ \widehat{k} \blacktriangleright_V k}{\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k}\end{array}\right):$$

We have $\Gamma \vdash \{Q'\}\ \widehat{k} \blacktriangleright_V k$ by the second premise and $(\widehat{s}, \widehat{h}) \models Q$ by *(Assumption 2)*. Since $Q \Rightarrow Q'$ we have $(\widehat{s}, \widehat{h}) \models Q'$. This, together with *(Assumption 3)* and *(Assumption 4)* allows us to apply the induction hypothesis on $\Gamma \vdash \{Q'\}\ \widehat{k} \blacktriangleright_V k$, thus proving the goal.

$$\textbf{CASE}\ \left(\begin{array}{l}\textsc{Branch}\\ \dfrac{\forall i.\ (\Gamma \vdash \{Q \wedge e_i\}\ \widehat{k_i} \blacktriangleright_V k_i)}{\Gamma \vdash \{Q\}\ \mathsf{branch}\ \ldots, e_i \Rightarrow \widehat{k_i}, \ldots\ \mathsf{end} \blacktriangleright_V \mathsf{branch}\ \ldots, e_i \Rightarrow k_i, \ldots\ \mathsf{end}}\end{array}\right):$$

Since $\gamma \xrightarrow[P]{} \gamma'$ we have that $[\![e_i]\!]\,s = \mathsf{true}$ for some i and $\gamma' = \langle k_i, (s, h)\rangle$. By Corollary 3 we have that $V \cap \mathit{fv}(e) = \emptyset$. Thus, $\mathit{fv}(e) \subseteq \widetilde{V}$. This lets us apply Lemma 1 to conclude that $[\![e_i]\!]\,\widehat{s} = \mathsf{true}$. Thus, $\widehat{\gamma} \xrightarrow[\widehat{P}]{} \widehat{\gamma}\,'$ and $\widehat{\gamma}\,' = \langle \widehat{k_i}, (\widehat{s}, \widehat{h})\rangle$.

Since $[\![e_i]\!]\,\widehat{s} = \mathsf{true}$ and $(\widehat{s}, \widehat{h}) \models Q$ by *(Assumption 2)* we have $(\widehat{s}, \widehat{h}) \models Q \wedge e_i$. We also have $\Gamma \vdash \{Q \wedge e_i\}\ \widehat{k_i} \blacktriangleright_V k_i$ as one of the premises of the rule under consideration. Then $\gamma'\ R^{V,\Gamma}\ \widehat{\gamma}\,'$ follows from these facts and *(Assumption 3)* and *(Assumption 4)*. We have shown that in this case $\widehat{P}$ can match the transition that P takes.

$$\textbf{CASE}\ \left(\begin{array}{l}\textsc{False}\\ \dfrac{}{\Gamma \vdash \{\mathsf{false}\}\ \mathsf{halt} \blacktriangleright_V k}\end{array}\right):$$

This case holds vacuously. One of our assumptions is that $(\widehat{s}, \widehat{h}) \models Q$. But in this case $Q = \mathsf{false}$. Since there are no states satisfying false, our assumptions are contradictory.

$$\textbf{CASE}\left(\begin{array}{l}\textsc{Inst-Assign}\\ \dfrac{\{Q\}\ x := e\ \{Q'\} \qquad \Gamma \vdash \{Q'\}\ \widehat{k} \blacktriangleright_V k}{\Gamma \vdash \{Q\}\ (x := e\mathbf{;}\widehat{k}) \blacktriangleright_V k}\ x \in V\end{array}\right):$$

We will show that $\widehat{P}$ stutters. We have that $\widehat{\gamma} = \langle (x := e\mathbf{;}\widehat{k}), (\widehat{s}, \widehat{h})\rangle$ and, applying the definition of $\xrightarrow[\widehat{P}]{}$ we have $\widehat{\gamma} \xrightarrow[\widehat{P}]{} \widehat{\gamma}\,'$ where $\widehat{\gamma}\,' = \langle \widehat{k}, (\widehat{s}[x \to [\![e]\!]\,\widehat{s}], \widehat{h})\rangle$. Since $x \in V$ we have $\widehat{s}[x \to [\![e]\!]\,\widehat{s}] =_{\widehat{V}} \widehat{s}$ and thus, by *(Assumption 3)* and transitivity of $=_{\widehat{V}}$ we have $\widehat{s}[x \to [\![e]\!]\,\widehat{s}] =_{\widehat{V}} s$. This is one condition required to establish $\gamma\ R^{V,\Gamma}\ \widehat{\gamma}\,'$.

The premise $\{Q\}\ \ x\ := \ e\ \ \{Q'\}$ and *(Assumption 2)* allow us to conclude that $(\widehat{s}[x \to [\![e]\!]\,\widehat{s}], \widehat{h}) \models Q'$. This is another condition for $\gamma\ R^{V,\Gamma}\ \widehat{\gamma}\,'$. The second premise of the rule under consideration and *(Assumption 4)* provide the other two conditions, completing the proof that $\gamma\ R^{V,\Gamma}\ \widehat{\gamma}\,'$.

We must also show that *rankl* decreases. We have $\mathit{rankl}(\widehat{\gamma}, \gamma, \gamma') = \mathit{size}(x := e\mathbf{;}\ \widehat{k})$ and $\mathit{rankl}(\widehat{\gamma}\,', \gamma, \gamma') = \mathit{size}(\widehat{k})$. Since $\mathit{size}(k)$ is the size of the abstract syntax tree for k, we have that $\mathit{size}(\widehat{k}) < \mathit{size}(x := e\mathbf{;}\ \widehat{k})$.

$$\textbf{CASE}\left(\begin{array}{l}\textsc{Inst-Disj}\\ \dfrac{\Gamma \vdash \{Q_1\}\ \widehat{k_1} \blacktriangleright_V k \qquad \Gamma \vdash \{Q_2\}\ \widehat{k_2} \blacktriangleright_V k}{\Gamma \vdash \{Q_1 \vee Q_2\}\ \mathsf{branch\ true} \Rightarrow \widehat{k_1}, \mathsf{true} \Rightarrow \widehat{k_2}\ \mathsf{end} \blacktriangleright_V k}\end{array}\right):$$

We will show that $\widehat{\gamma}$ makes a stuttering transition. That is, $\widehat{\gamma} \xrightarrow[\widehat{P}]{} \widehat{\gamma}\,'$ and $\gamma\ R^{V,\Gamma}\ \widehat{\gamma}\,'$. From *(Assumption 2)* we have that $(\widehat{s}, \widehat{h}) \models Q_1 \vee Q_2$. This implies that either $(\widehat{s}, \widehat{h}) \models Q_1$ or $(\widehat{s}, \widehat{h}) \models Q_2$.

Suppose the first case holds, so $(\widehat{s}, \widehat{h}) \models Q_1$. Then let $\widehat{\gamma}\,'$ be $\langle \widehat{k_1}, (\widehat{s}, \widehat{h})\rangle$. Since $(\widehat{s}, \widehat{h}) \models \mathsf{true}$, we have that $\widehat{\gamma} \xrightarrow[\widehat{P}]{} \widehat{\gamma}\,'$. That $\gamma\ R^{V,\Gamma}\ \widehat{\gamma}\,'$ then follows from the first premise, *(Assumption 3)*, *(Assumption 4)*, and $(\widehat{s}, \widehat{h}) \models Q_1$, which was our assumption for this case.

The $(\widehat{s}, \widehat{h}) \models Q_2$ case is similar, with Q_2 substituted for Q_1 and the second premise used in place of the first premise.

The condition that *rankl* decreases is satisfied since $\widehat{k}_1$ is a smaller term than $\mathsf{branch}\ \mathsf{true} \Rightarrow \widehat{k}_1, \mathsf{true} \Rightarrow \widehat{k}_2\ \mathsf{end}$.

CASE $\left(\begin{array}{l}\textsc{Inst-Exists} \\ \dfrac{\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k}{\Gamma \vdash \{\exists x^\tau .\ Q\}\ (x := ?^\tau ; \widehat{k}) \blacktriangleright_V k}\ x \in V\end{array}\right)$:

This is similar to the previous case, except that the non-determinism is unbounded rather than a choice between two alternatives. We will consider only the case where $\tau = \mathrm{i}$. The case for a is similar. We have that $(\widehat{s}, \widehat{h}) \models \exists x^{\mathrm{i}}.\ Q$ and thus, by the semantics of existential quantifiers there is some $v \in \mathbb{Z}$ such that $(\widehat{s}[x^{\mathrm{i}} \to v], \widehat{h}) \models Q$. From the semantics for non-deterministic assignment, we know there is some execution of $x^{\mathrm{i}} := ?^{\mathrm{i}}$ that assigns v to x^{i}. Formally, we have that $(\widehat{s}[x^{\mathrm{i}} \to v], \widehat{h}) \in [\![x^{\mathrm{i}} := ?^{\mathrm{i}}]\!]\, \widehat{s}$ which implies that $\langle (x^{\mathrm{i}} := ?^{\mathrm{i}} ; \widehat{k}), (\widehat{s}, \widehat{h}) \rangle \underset{\widehat{P}}{\longrightarrow} \widehat{\gamma}\,'$ where $\widehat{\gamma}\,' = \langle \widehat{k}, (\widehat{s}[x^{\mathrm{i}} \to v], \widehat{h}) \rangle$. It remains to show that $\gamma\ R^{V,\Gamma}\ \widehat{\gamma}\,'$.

We have $(\widehat{s}[x^{\mathrm{i}} \to v], \widehat{h}) \models Q$ and $\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k$. Since $x^{\mathrm{i}} \in V$ and $\widetilde{V}$ is the complement of V, we have that $x^{\mathrm{i}} \notin \widetilde{V}$. This allows us to conclude that $\widehat{s}[x^{\mathrm{i}} \to v] =_{\widetilde{V}} \widehat{s}$ and thus, by transitivity of $=_{\widetilde{V}}$ and *(Assumption 3)* we have $\widehat{s}[x^{\mathrm{i}} \to v] =_{\widetilde{V}} s$. This is the third of the four conditions for establishing $\gamma\ R^{V,\Gamma}\ \widehat{\gamma}\,'$. *(Assumption 4)* provides the fourth condition and completes the proof.

As before, the condition on *rankl* reduces to showing that $\mathit{size}(\widehat{k}) < \mathit{size}(x^{\mathrm{i}} := ?^{\mathrm{i}};\ \widehat{k})$ which is immediate.

CASE $\left(\begin{array}{l}\textsc{Inst-Assume} \\ \dfrac{Q \Rightarrow e \qquad \Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k}{\Gamma \vdash \{Q\}\ \mathsf{assume}(e) ; \widehat{k} \blacktriangleright_V k}\end{array}\right)$:

We will show that $\langle (\mathsf{assume}(e) ; \widehat{k}), (\widehat{s}, \widehat{h}) \rangle \underset{P}{\longrightarrow} \widehat{\gamma}\,'$ and $\gamma\ R^{V,\Gamma}\ \widehat{\gamma}\,'$. The transition can occur if $(\widehat{s}, \widehat{h}) \models e$. We have from *(Assumption 2)* that $(\widehat{s}, \widehat{h}) \models Q$. The premise $Q \Rightarrow e$

then gives us that $(\widehat{s}, \widehat{h}) \models e$. It remains to show that $\gamma R \widehat{\gamma}\,'$. This follows from *(Assumption 2)*, *(Assumption 3)*, *(Assumption 4)*, and the second premise.

As before, since $\mathit{size}(\widehat{k}) < \mathit{size}(\mathsf{assume}(e)\texttt{;}\ \widehat{k})$ we have that rankl decreases.

goal *(Final States Related)*:

By Definition 14 we have that the final states F of $\llparenthesis P \,|\, Q_0 \rrparenthesis$ are

$$F = \big\{\mathbf{final}(s,h) \;\big|\; s \in \mathit{Stores} \wedge h \in \mathit{Heaps}\big\} \cup \big\{\mathbf{error}\big\}$$

The final states $\widehat{F}$ of $\llparenthesis \widehat{P} \,|\, Q_0 \rrparenthesis$ are the same.

$$\widehat{F} = \big\{\mathbf{final}(s,h) \;\big|\; s \in \mathit{Stores} \wedge h \in \mathit{Heaps}\big\} \cup \big\{\mathbf{error}\big\}$$

We must show the following.

$$\forall \gamma \in I.\, \forall \widehat{\gamma} \in \widehat{I}.\, (\gamma \; R^{V,\Gamma} \; \widehat{\gamma}) \Rightarrow (\gamma \in F \Leftrightarrow \widehat{\gamma} \in \widehat{F})$$

This follows directly from our definition of $R^{V,\Gamma}$. Examining Definition 31, we can see that $\mathbf{error}$ is only $R^{V,\Gamma}$-related to $\mathbf{error}$ and $\mathbf{final}(s,h)$ is only $R^{V,\Gamma}$-related to $\mathbf{final}(s,h)$. □

Below we make note of an important corollary. This follows from the theorem above (Theorem 22), Theorem 18, and Corollary 2.

Corollary 4. *Let* $Q_0 = \Gamma(\mathit{initloc}(P))$. *Then* $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ *and* $\llparenthesis \widehat{P} \,|\, Q_0 \rrparenthesis \models \phi$ *implies* $\llparenthesis P \,|\, Q_0 \rrparenthesis \models \boxed{\exists}(V, \phi)$

This tells us that if we prove some LTSL formula holds of $\llparenthesis \widehat{P} \,|\, Q_0 \rrparenthesis$, we can obtain an LTSL formula that holds of $\llparenthesis P \,|\, Q_0 \rrparenthesis$ by existentially quantifying the instrumentation variables appearing in the formula. As a special case, formulas that hold of $\widehat{P}$ and do not contain instrumentation variables do not need to be changed. The same formula that held of $\widehat{P}$ will also hold of P.

As an example, consider the program below.

```
$L_0$ : goto $L_1$
$L_1$ : ① branch $x \neq$ nil $\Rightarrow$ ② $x :=$ $x$.next; ③ goto $L_1$,
                 $x =$ nil $\Rightarrow$ ④ halt end
```

The following is an instrumented version of this program.

```
$L_0 : n_2 := n$; $n_1 := 0$; goto $L_1$
$L_1$ : ❶ branch $x \neq$ nil $\Rightarrow$ ❷ $x := x$.next; ❸ $n_1 := n_1 + 1$;
                              $n_2 := n_2 - 1$; goto $L_1$,
              $x =$ nil $\Rightarrow$ ❹ halt end
```

Starting from the precondition $ls(n, x, \mathsf{nil})$ we can show that the following formula holds of the instrumented program.

$$\mathbf{G}\big(atloc(L_1) \Rightarrow (\exists x'.\ ls(n_1, x', x) * ls(n_2, x, \mathsf{nil})) \wedge n_1 + n_2 = n\big)$$

This states that if n is the length of the list before executing the code, then at L_1, during every iteration of the loop, n_1 and n_2 sum to n. Note that n is not an instrumentation variable here, but a program variable containing the initial length of the list. Our corollary above then tells us that the following LTSL formula holds of the original program.

$$\mathbf{G}\big(atloc(L_1) \Rightarrow (\exists n_1, n_2, x'.\ ls(n_1, x', x) * ls(n_2, x, \mathsf{nil})) \wedge n_1 + n_2 = n)\big)$$

This is the same formula as before, but with the instrumentation variables n_1 and n_2 existentially quantified. This loop invariant is strong enough to let us conclude that the length of the list is unchanged by the traversal.

4.4 Numeric Abstractions

In Figure 4.12 we give the rules for generating a *projection* of a continuation onto a set of variables V. This results in a continuation that only involves reads and writes to variables in V and does not include any heap commands. The projection function $\pi_V(k)$ is defined with the help of the predicates $W_V(c)$ and $det_V(c)$.

The predicate $W_V(c)$ holds if the command c writes to a variable in V. For example, if $V = \{x\}$, then $x := \mathsf{alloc}(\ldots)$ satisfies this since it results in the newly allocated address being written to x, which is in V. The other commands that write to x are $x := e$, $x := ?$, and $x := x_2.f$.

The predicate $det_V(c)$ holds if the result of c is *determined* given only the values of the variables in V (and, crucially, given no access to the heap). The only command that satisfies this is $x := e$ in the case where $fv(e) \subseteq V$.

The function $\pi_V(k)$ discards command that do not write to variables in V and it replaces with non-deterministic assignment any commands that write to variables in V but are not determined. The result is that writes into heap cells and free x commands are always discarded. Allocation and heap lookup are replaced with non-deterministic assignment. Non-deterministic assignments present in the original program are carried through to the projected program provided they affect a variable in V. For deterministic assignment commands $x := e$, the command is discarded if $x \notin V$, it is converted to the non-deterministic assignment $x := ?$ if e contains any variables not in V, and otherwise it is carried through unchanged.

Branch conditions are carried over unchanged if the condition only involves variables in V or, if variables outside of V are required, the branch is replaced by true. With such an approach, when we encounter a branch that cannot be evaluated accurately in the projection, we conservatively assume that the branch can be taken, thus erring on the side of exploring more paths (and consequently maintaining soundness for universal properties over paths, such as our LTSL formulae). Note that $fv(\pi_V(P)) \subseteq V$, a fact that can be verified by induction over the structure of P.

The projection operation for programs is defined as follows (where $\pi_V(P(l))$ refers to the projection of the continuation $P(l)$, as defined in Figure 4.12).

Definition 32. *The projection of a program P onto variables V, written $\pi_V(P)$, is the program P' such that $dom(P') = dom(P)$, $initloc(P') = initloc(P)$ and $\forall l \in dom(P)$. $P'(l) = \pi_V(P(l))$.*

Our numeric programs will be the result of projecting an instrumented program onto a subset of the integer-valued variables. These variables can include instrumented variables as well as program variables. Maintaining program variables in the projection is necessary when the LTSL formula being checked contains program variables. It may be necessary in other cases as well—for example, if termination depends on the fact that a program

Commands that write to variables in V

$$\begin{aligned} W_V(c) \quad \text{iff} \quad & \text{for some } x \in V, c \text{ has the form} \\ & \qquad x := e \text{ or } x := {?} \text{ or } x := \mathsf{alloc}(\ldots) \text{ or } x := x_2.f \end{aligned}$$

Commands that are determined given V

$$\mathit{det}_V(c) \quad \text{iff} \quad c \text{ has the form } x := e \text{ and } \mathit{fv}(e) \subseteq V$$

Definition of $\pi_V(k)$

$$\pi_V(c\,\mathbf{;}\,k) \quad = \quad \begin{cases} c\,\mathbf{;}\,(\pi_V(k)) & \text{if } W_V(c) \text{ and } \mathit{det}_V(c) \\ x := {?}\,\mathbf{;}\,(\pi_V(k)) & \text{if } W_V(c) \text{ and } \neg \mathit{det}_V(c) \text{ and} \\ & \quad c \text{ has the form } x := \ldots \\ \pi_V(k) & \text{otherwise} \end{cases}$$

$$\text{let } \pi_V \begin{pmatrix} \mathsf{branch} \\ \qquad e_1 \Rightarrow k_1, \ldots, \\ \qquad e_n \Rightarrow k_n \\ \mathsf{end} \end{pmatrix} \quad = \quad \begin{matrix} \mathsf{branch} \\ \qquad e'_1 \Rightarrow \pi_V(k_1), \ldots, \\ \qquad e'_n \Rightarrow \pi_V(k_n) \\ \mathsf{end} \end{matrix} \qquad \text{where } e'_i = \begin{cases} e_i & \text{if } \mathit{fv}(e_i) \subseteq V \\ \mathsf{true} & \text{if } \mathit{fv}(e_i) \not\subseteq V \end{cases}$$

$$\pi_V(k) \quad = \quad k \quad \text{if } k = \mathsf{abort} \text{ or } k = \mathsf{halt} \text{ or } k = \mathsf{goto}\ l$$

Figure 4.12: Definition of the function $\pi_V(k)$ which projects a continuation onto variables in V.

variable is decreasing and has a lower bound, then that variable must be preserved in the projection.

4.4.1 Projection and Simulation

We now discuss how the concept of program projections fits into the formal framework presented earlier for instrumented programs. Recall the definition of $\overset{\mathrm{s}}{=}_V$ (Definition 24), reproduced below.

Definition 24. $\overset{\mathrm{s}}{=}_V$ is the least relation on execution states satisfying the following.

$$\begin{array}{rcll}
\langle k, (s,h)\rangle & \overset{\mathrm{s}}{=}_V & \langle k', (s',h')\rangle & \text{iff } s =_V s' \\
\mathbf{goto}(l, (s,h)) & \overset{\mathrm{s}}{=}_V & \mathbf{goto}(l, (s',h')) & \text{iff } s =_V s' \\
\mathbf{final}(s,h) & \overset{\mathrm{s}}{=}_V & \mathbf{final}(s',h') & \text{iff } s =_V s' \\
\mathbf{error} & \overset{\mathrm{s}}{=}_V & \mathbf{error} &
\end{array}$$

This will be the relation on states that is preserved by projection. The following theorem captures this fact. The proof is fairly straightforward, as the projection translates each command or branch to a version that is at least as non-deterministic as the original. Thus, the projected command / branch includes the original behavior as well as possibly some additional behavior.

Theorem 23. *If $P' = \pi_V(P)$ then there exists an R such that for all Q_0, the following holds.*

$$(\!(P \mid Q_0)\!) \precsim_{R, \overset{\mathrm{s}}{=}_V} (\!(P' \mid Q_0)\!)$$

Proof. The R in this case is the least relation satisfying the following.

$$\begin{array}{rcll}
\langle k, (s,h)\rangle & R & \langle k', (s',h')\rangle) & \text{iff } k' = \pi_V(k) \text{ and } s =_V s' \\
(\mathbf{goto}(l, (s,h))) & R & (\mathbf{goto}(l, (s',h'))) & \text{iff } s =_V s' \\
\mathbf{final}(s,h) & R & \mathbf{final}(s',h') & \text{iff } s =_V s' \\
\mathbf{error} & R & \mathbf{error} &
\end{array}$$

The ranking functions *rankl* and *rankt* are defined as in the proof of Theorem 22 in Section 4.3 (see page 164).

Initial States Related First we show that initial states are related. Every state $\mathbf{goto}(\mathit{initloc}(P), (s, h))$ is related to the state $\mathbf{goto}(\mathit{initloc}(P'), (s, h))$. This holds because $P' = \pi_V(P)$ ensures that $\mathit{initloc}(P') = \mathit{initloc}(P)$ and reflexivity of $\overset{\mathrm{s}}{=}_V$ gives us $s \overset{\mathrm{s}}{=}_V s$. Together, these establish the necessary conditions for R to hold, giving us

$$(\mathbf{goto}(\mathit{initloc}(P), (s, h)))\ R\ (\mathbf{goto}(\mathit{initloc}(P'), (s, h)))$$

$\overset{\mathrm{s}}{=}_V$**-equivalent** The second condition of stuttering simulation, that R implies $\overset{\mathrm{s}}{=}_V$ is easy to check. We can see that R is strictly contained in $\overset{\mathrm{s}}{=}_V$ since all the conditions are the same except that R additionally requires $k' = \pi_V(k)$ in the case where $\langle k, (s, h)\rangle\ R\ \langle k', (s', h')\rangle$.

Transitions Match The third condition is that any transition of P can be matched. Suppose $\gamma_1\ R\ \gamma_2$ and $\gamma_1 \xrightarrow[P]{} \gamma_1'$. Then γ_1 must either have the form $\mathbf{goto}(l, (s_1, h_1))$ or $\langle k_1, (s_1, h_1)\rangle$.

CASE $\gamma_1 = \mathbf{goto}(l, (s_1, h_1))$: By the definition of R, we have that γ_2 has the form $\mathbf{goto}(l, (s_2, h_2))$ with $s_1 =_V s_2$. By the semantics of program transitions, we have

$$\mathbf{goto}(l, (s_1, h_1)) \xrightarrow[P]{} \langle P(l), (s_1, h_1)\rangle$$

and

$$\mathbf{goto}(l, (s_2, h_2)) \xrightarrow[P']{} \langle P'(l), (s_2, h_2)\rangle$$

We will show

$$\langle P(l), (s_1, h_1)\rangle\ R\ \langle P'(l), (s_2, h_2)\rangle$$

We already have $s_1 =_V s_2$. It remains to show that $P'(l) = \pi_V(P(l))$. This follows directly from the definition of $\pi_V(P)$ and the fact that $P' = \pi_V(P)$. Expanding these definitions, we have that $\pi_V(P)(l) = \pi_V(P(l))$, which gives us our result.

CASE $\gamma_1 = \langle k_1, (s_1, h_1)\rangle$: Since $\gamma_1\ R\ \gamma_2$, we have that γ_2 has the form $\langle k_2, (s_2, h_2)\rangle$ with $s_1 =_V s_2$ and $k_2 = \pi_V(k_1)$. We now consider each possible form for k_1.

CASE $k_1 = (c; k_1')$: In this case, k_2, which is $\pi_V(k_1)$, depends on whether $W_V(c)$ and $\mathit{det}_V(c)$ are true.

SUB-CASE $W_V(c)$ AND $det_V(c)$: In this case, we have that $k_2 = (c\,\mathbf{;}\ k_2')$ where $k_2' = \pi_V(k_1')$. That $det_V(c)$ holds ensures that $c = (x := e)$ and $fv(e) \subseteq V$ which, together with $s_1 =_V s_2$ ensures that $[\![e]\!]\, s_1 = [\![e]\!]\, s_2$ (by Lemma 1). Let v be this value ($[\![e]\!]\, s_1$). The definition of $\xrightarrow[P]{}$ tells us that $\gamma_1 \xrightarrow[P]{} \langle k_1', (s_1[x \to v], h_1)\rangle$. Similarly, we have that $\gamma_2 \xrightarrow[P']{} \langle k_2', (s_2[x \to v], h_2)\rangle$. We must show that $(s_1[x \to v]) =_V (s_2[x \to v])$. This follows from the fact that $s_1 =_V s_2$. We already have that $k_2' = \pi_V(k_1')$. Thus, P' can match the transition.

SUB-CASE $W_V(c)$ AND $\neg det_V(c)$: In this case, c has either the form $x := e$ or $x := ?$ or $x := \mathsf{alloc}(\ldots)$ or $x := x_2.f$ for some $x \in V$. In all these cases, we have a transition $\langle (c\,\mathbf{;}\, k_1'), (s_1, h_1)\rangle \xrightarrow[P]{} \langle k_1', (s_1', h_1')\rangle$. The exact conditions on s_1' and h_1' differ; however, in every case we have that $s_1' = s_1[x \to v]$ for some v in the appropriate domain (either addresses or integers depending on the type of x). We have $k_2 = \pi_V(k_1) = (x := ?\,\mathbf{;}\, \pi_V(k_1'))$, which, given the semantics of $x := ?$ ensures that

$$\langle k_2, (s_2, h_2)\rangle \xrightarrow[P']{} \langle \pi_V(k_1'), (s_2[x \to v], h_2)\rangle$$

That $(s_1[x \to v]) =_V (s_2[x \to v])$ then follows from $s_1 =_V s_2$, which we have from $\gamma_1 \mathrel{R} \gamma_2$. Thus, P' can match the transition of P.

SUB-CASE $\neg(W_V(c))$: In this case, $k_2 = \pi_V(k_1')$.

In this case, either c does not write to some store variable x or it does but x is not in V. If the command in question does not modify the store, then we have $\gamma_1' = \langle k_1', (s_1, h_1')\rangle$. We also have $\gamma_1 \mathrel{R} \gamma_2$ and will show that $\gamma_1' \mathrel{R} \gamma_2$ where we recall that $\gamma_2 = \langle k_2, (s_2, h_2)\rangle$. To do this we must show $s_1 =_V s_2$, which we already have from the definition of R and $\gamma_1 \mathrel{R} \gamma_2$. We also must show that $k_2 = \pi_V(k_1')$, but this we already have from our assumptions. The only remaining condition is to show that the ranking function decreases. This is the case since k_1' is a sub-term of k_1.

We now consider the case where the command c modifies store variable x, but x is not in V. Here we have that $\gamma_1' = \langle k_1', (s_1[x \to v], h_1')\rangle$ for some v. We will show that $\gamma_1' \mathrel{R} \gamma_2$, where $\gamma_2 = \langle k_2, (s_2, h_2)\rangle$. We already have that $k_2 = \pi_V(k_1')$. We must also show that $(s_1[x \to v]) =_V s_2$. This follows from $s_1 =_V s_2$ and $x \notin V$, which we have from our assumptions.

CASE $k_1 = (\textsf{branch } e_1 \Rightarrow k'_1, \ldots, e_n \Rightarrow k'_n \textsf{ end})$: In this case we have

$$k_2 = (\textsf{branch } e'_1 \Rightarrow \pi_V(k'_1), \ldots, e'_n \Rightarrow \pi_V(k'_n) \textsf{ end})$$

where $e'_i = e_i$ if $fv(e_i) \subseteq V$ or $e'_i = \textsf{true}$ otherwise.

We are assuming that $\langle k_1, (s_1, h_1)\rangle \xrightarrow[P]{} \gamma'_1$. If this is the case, then $\gamma'_1 = \langle k'_i, (s_1, h_1)\rangle$ for some i such that $[\![e_i]\!]\, s_1 = \textsf{true}$. We want to show that for $\gamma_2 = \langle k_2, (s_2, h_2)\rangle$ we have $\gamma_2 \xrightarrow[P']{} \gamma'_2$ and $\gamma'_1 \; R \; \gamma'_2$. We first case split on whether $e'_i = \textsf{true}$ or $e'_i = e_i$. In the first case, we are done since branches labeled with $\textsf{true}$ can always be taken. So we have $\gamma_2 \xrightarrow[P']{} \langle \pi_V(k'_i), (s_2, h_2)\rangle$. We already have $s_1 =_V s_2$, which is sufficient to show $\gamma'_1 \; R \; \langle \pi_V(k'_i), (s_2, h_2)\rangle$.

In the case where $e'_i = e_i$, we use our an assumption $[\![e_i]\!]\, s_1 = \textsf{true}$. Since $s_1 =_V s_2$, we have $[\![e_i]\!]\, s_2 = \textsf{true}$ by Lemma 1. Applying the equality $e'_i = e_i$ gives us $[\![e'_i]\!]\, s_2 = \textsf{true}$, which is sufficient to ensure that the transition $\gamma_2 \xrightarrow[P']{} \langle \pi_V(k'_i), (s_2, h_2)\rangle$ exists. That $\gamma'_1 \; R \; \langle \pi_V(k'_i), (s_2, h_2)\rangle$ then follows from our assumption that $s_1 =_V s_2$.

CASE $k_1 = \textsf{abort}$: In this case, $\gamma'_1 = \textbf{error}$. Also, $k_2 = \pi_V(k_1) = \textsf{abort}$, which ensures $\gamma_2 \xrightarrow[P']{} \textbf{error}$. Since $\textbf{error} \; R \; \textbf{error}$ we are done.

CASE $k_1 = \textsf{halt}$: In this case, $k_2 = \pi_V(k_1) = \textsf{halt}$. We have $\gamma_1 \xrightarrow[P]{} \textbf{final}(s_1, h_1)$ and $\gamma_2 \xrightarrow[P']{} \textbf{final}(s_2, h_2)$. From $\gamma_1 \; R \; \gamma_2$ and the definition of R we have $s_1 =_V s_2$, which implies that $\textbf{final}(s_1, h_1) \; R \; \textbf{final}(s_2, h_2)$.

CASE $k_1 = \textsf{goto } l$: In this case, $k_2 = \pi_V(k_1) = \textsf{goto } l$. We have $\gamma_1 \xrightarrow[P]{} \textbf{goto}(l, (s_1, h_1))$ and $\gamma_2 \xrightarrow[P']{} \textbf{goto}(l, (s_2, h_2))$. From $\gamma_1 \; R \; \gamma_2$ and the definition of R we have $s_1 =_V s_2$, which implies that $\textbf{goto}(l, (s_1, h_1)) \; R \; \textbf{goto}(l, (s_2, h_2))$. □

4.4.2 Combining Projection and Instrumentation

We have shown that a program is simulated by any of its instrumentations and that an instrumentation (or any other program) is simulated by any of its projections. As one of our goals is to use numeric programs, which are projections of instrumentations, to reason about the original program, we need to obtain a result relating numeric programs to the original program. Figure 4.13 summarizes the situation.

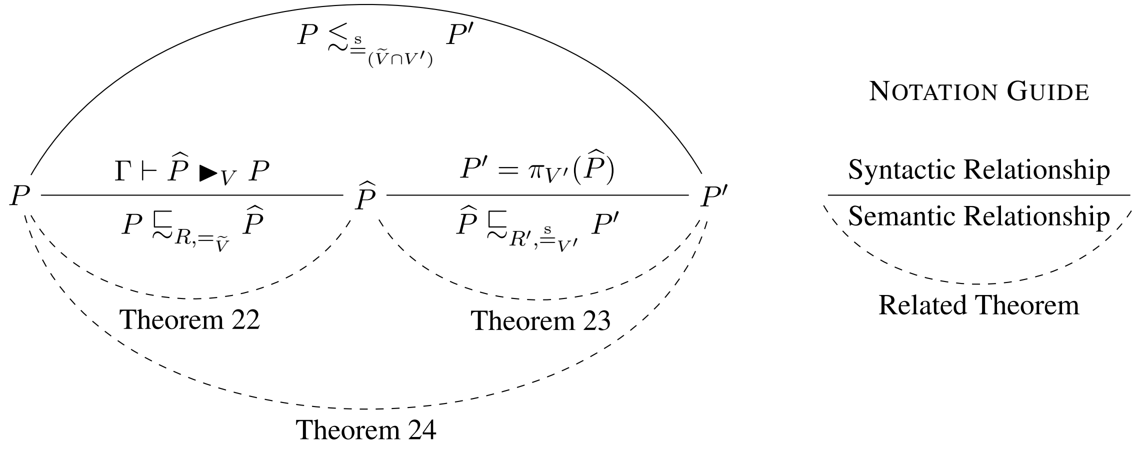

$P \lesssim_{\overset{s}{=}_{(\widetilde{V}\cap V')}} P'$
$\Gamma \vdash \widehat{P} \blacktriangleright_V P$
$P'=\pi_{V'}(\widehat{P})$
P
$\widehat{P}$
P'
$P \mathrel{\underset{\sim}{\sqsubset}}_{R,=_{\widetilde{V}}} \widehat{P}$
$\widehat{P} \mathrel{\underset{\sim}{\sqsubset}}_{R',\overset{s}{=}_{V'}} P'$
Theorem 22
Theorem 23
Theorem 24
NOTATION GUIDE
Syntactic Relationship
Semantic Relationship
Related Theorem

Figure 4.13: A summary of the current state of the technical development.

The following theorem ties the two endpoints in this figure together, describing the simulation result that holds of projections of instrumentations.

Theorem 24. *(Projections of Instrumentations) If* $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ *and* $P' = \pi_{V'}(\widehat{P})$ *and* $Q_0 = \Gamma(\mathit{initloc}(P))$ *then*

$$\llparenthesis P \mid Q_0 \rrparenthesis \lesssim_{\overset{s}{=}_{(\widetilde{V}\cap V')}} \llparenthesis P' \mid Q_0 \rrparenthesis$$

Proof. The result follows from Theorem 22, Theorem 23, Theorem 18, and Theorem 13. By Theorem 22 we have some R such that $\llparenthesis P \mid Q_0 \rrparenthesis \mathrel{\underset{\sim}{\sqsubset}}_{R,=_{\widetilde{V}}} \llparenthesis \widehat{P} \mid Q_0 \rrparenthesis$. By Theorem 23 we have an R' such that $\llparenthesis \widehat{P} \mid Q_0 \rrparenthesis \mathrel{\underset{\sim}{\sqsubset}}_{R',\overset{s}{=}_{V'}} \llparenthesis P' \mid Q_0 \rrparenthesis$. Applying Theorem 18 to each of these yields

$$\llparenthesis P \mid Q_0 \rrparenthesis \lesssim_{=_{\widetilde{V}}} \llparenthesis \widehat{P} \mid Q_0 \rrparenthesis$$

and

$$\llparenthesis \widehat{P} \mid Q_0 \rrparenthesis \lesssim_{\overset{s}{=}_{V'}} \llparenthesis P' \mid Q_0 \rrparenthesis$$

Expanding the definitions of $=_{\widetilde{V}}$ and $\overset{s}{=}_{V'}$ allows us to verify the following.

$$\forall a, b, c.\ (a =_{\widetilde{V}} b) \wedge (b \overset{s}{=}_{V'} c) \Rightarrow (a \overset{s}{=}_{\widetilde{V}\cap V'} c)$$

The proof is by case analysis on a. To take a representative case, suppose $a = \mathbf{final}(s, h)$. Then $b = \mathbf{final}(s', h)$ with $s =_{\widetilde{V}} s'$ and $c = \mathbf{final}(s'', h')$ with $s' =_{V'} s''$. We must show that $\mathbf{final}(s, h) \overset{\mathrm{s}}{=}_{\widetilde{V} \cap V'} \mathbf{final}(s'', h')$. This is the case if we can show $s \overset{\mathrm{s}}{=}_{\widetilde{V} \cap V'} s''$. This requires showing $\forall x.\ (x \in \widetilde{V} \cap V') \Rightarrow s(x) = s''(x)$. If $x \in \widetilde{V} \cap V'$ then $x \in \widetilde{V}$ and $x \in V'$. This allows us to use our assumptions $s =_{\widetilde{V}} s'$ and $s' =_{V'} s''$ to conclude $s(x) = s''(x)$.

Theorem 13 then combines these results, giving us

$$\llparenthesis P \mid Q_0 \rrparenthesis \lesssim_{\overset{\mathrm{s}}{=}_{(\widetilde{V} \cap V')}} \llparenthesis P' \mid Q_0 \rrparenthesis$$

□

The result of this is that numeric programs preserve LTSLP properties over variables in $\widetilde{V} \cap V'$. In practical terms, this means that, provided we include all of the integer-valued variables from the original program in the projection, then any LTSLP property over these original integer variables can be checked by analyzing P'.

4.5 Example

We now consider an example that shows how the translation to numeric programs can be used to check program properties (and also how choosing the wrong numeric program can result in an inability to prove the desired property, an unsurprising result given that numeric programs over-approximate the behavior of the original program).

Figure 4.14 gives a program that traverses a circular linked list rooted at x. The main loop checks whether $x.\mathsf{next} = x$. This is true if and only if the list contains only one element. If the list has more than one element, then $(x.\mathsf{next}).\mathsf{data}$[1] is compared to v. If it is less than or equal to v, then the list cell at $x.\mathsf{next}$ is removed. Otherwise, v is set to $(x.\mathsf{next}).\mathsf{data}$. This will cause the cell at $x.\mathsf{next}.\mathsf{data}$ to be freed during the next iteration.

[1] We use C-style multiple dereference for clarity. The intermediate variables x', y and t are used in Figure 4.14 since our language does not support multiple dereference, nor dereference inside of expressions.

```
$\mathsf{L}_0$ : goto $\mathsf{L}_1$
$\mathsf{L}_1$ : $y = x$.next;
     branch $y = x \Rightarrow$ halt,
            $y \neq x \Rightarrow x' := x$.next;
                         $t := x'$.data;
                         goto $\mathsf{L}_2$
     end
$\mathsf{L}_2$ : branch $t \leq v \Rightarrow x$.next $:= x'$.next;
                        free $x'$;
                        goto $\mathsf{L}_1$,
            $t > v \Rightarrow v := x'$.data;
                        goto $\mathsf{L}_1$
     end
```

Figure 4.14: An example program that traverses a circular linked list, conditionally freeing elements.

In order to show that this program terminates, we will produce an instrumentation that tracks the following two instrumentation variables.

n	the size of the linked list at x
z	the value present at $(x.\mathsf{next}).\mathsf{data}$

We will use the following inductive definition to represent the circular linked list.

$$\begin{aligned} &ls(n, \mathit{first}, \mathit{next}) \equiv \\ &\qquad (\mathbf{emp} \wedge \mathit{first} = \mathit{next} \wedge n = 0) \\ &\quad \vee (\exists z, d.\ (\mathit{first} \mapsto [\mathsf{next} : z, \mathsf{data} : d]) * ls(n-1, z, \mathit{next})) \end{aligned}$$

First, we present an instrumentation tracking only n, the size of the linked list. The left half of Figure 4.15 presents the instrumented program. We consider executions starting from the precondition $\exists n.\ ls(n, x, x) \wedge n \geq 1$ indicating that there is a non-empty circular linked list at x. We underline the instrumentation commands in order to make it more clear which commands were added. The first instrumentation command $\underline{n := ?}$ allows us to remove the quantifier on n from the precondition and reason from $\Gamma(\mathsf{L}_1)$ (displayed at the bottom of Figure 4.15). The removal of an element from the list corresponds to a decrease of n by 1. The command $\underline{\mathsf{assume}(n = 1)}$ records a pure consequence of the branch condition $y = x$. As y is $x.\mathsf{next}$, we have $y = x$ exactly when the list contains a single cell.

The right half of Figure 4.15 gives the numeric program obtained by projecting the instrumented program onto the singleton set $\{n\}$. The branches from the original program become non-deterministic branches and we are left with only the assume commands involving n and the update to n in the first branch of the continuation at L_2. This program is not a sufficiently precise abstraction to enable us to show termination. While we are able to model the fact that n is decreasing, we cannot show that the branch which decreases n is taken infinitely often. It could, for example, be the case that the second branch of the continuation at L_2 is always taken. While it is not sufficient for termination, this numeric program does allow us to prove some non-trivial properties. For example, we can show that n is non-increasing, represented by the following LTSLP formula.

$$\mathbf{G}\big((\mathit{atloc}(\mathsf{L}_1) \wedge n = n_0) \supset \mathbf{G}(\mathit{atloc}(\mathsf{L}_1) \supset n \leq n_0)\big)$$

Note the use of the ghost variable n_0 to capture the current value of n. Since n_0 does not appear in the program, its value is never changed. Since the precondition does not mention n_0, it can have any value in the initial state. This ensures that there are traces for which the antecedent $\mathit{atloc}(\mathsf{L}_1) \wedge n = n_0$ is true. The use of implication then confines our attention to those traces when evaluating the rest of the formula.

We now move on to an instrumented version of the program that also tracks z, the current contents of $x.\mathsf{next}.\mathsf{data}$. The left half of Figure 4.16 gives the instrumented version of the program and the right half of the same figure contains the numeric program obtained by projecting this instrumented program onto the set of variables $\{n, z, v\}$. This program can be shown to terminate since the existence of z enables us to track the contents of $x.\mathsf{next}.\mathsf{data}$ across iterations of the loop at location L_1. Specifically, we can now show that in the numeric program, the second case of the branch at L_2 cannot occur infinitely often.

Instrumented Program

```
L_0 : $\underline{n := ?}$;
      goto L_1
L_1 : $y = x$.next;
      branch $y = x \Rightarrow$ $\underline{\text{assume}(n = 1)}$;
                     halt,
             $y \neq x \Rightarrow$ $\underline{\text{assume}(n > 1)}$;
                     $x'$ := $x$.next;
                     $t$ := $x'$.data;
                     goto L_2
      end
L_2 : branch $t \leq v \Rightarrow$ $x$.next := $x'$.next;
                     free $x'$;
                     $\underline{n := n - 1}$;
                     goto L_1,
             $t > v \Rightarrow$ $v$ := $x'$.data;
                     goto L_1
      end
```

Numeric Program

```
L_0 : $\underline{n := ?}$;
      goto L_1
L_1 : branch true $\Rightarrow$ $\underline{\text{assume}(n = 1)}$;
                     halt,
             true $\Rightarrow$ $\underline{\text{assume}(n > 1)}$;
                     goto L_2
      end
L_2 : branch true $\Rightarrow$ $\underline{n := n - 1}$;
                     goto L_1,
             true $\Rightarrow$ goto L_1
      end
```

$$\begin{aligned}
\Gamma(\mathsf{L}_0) &= \exists n.\ ls(n, x, x) \land n \geq 1 \\
\Gamma(\mathsf{L}_1) &= ls(n, x, x) \land n \geq 1 \\
\Gamma(\mathsf{L}_2) &= \exists a, b.\ \big(x \mapsto [\mathsf{next} : x', \mathsf{data} : a] * x' \mapsto [\mathsf{next} : b, \mathsf{data} : t] \\
&\qquad * ls(n - 2, b, x)\big) \land n > 1
\end{aligned}$$

Figure 4.15: An instrumented version of the program in Figure 4.14 and the corresponding projection onto the set $\{n\}$.

The reason is that executing this branch sets v to z, which then prevents the $\mathsf{assume}(z > v)$ statement from being satisfied the next time L_2 is reached, forcing execution to proceed along the first case of the branch. Thus, at least every other iteration of the loop at L_2 results in n decreasing by 1. If n is initially greater than or equal to 1 (a situation which the assume statements at L_1 force), then eventually n will be equal to 1 and the program will halt.

Finally, we consider a liveness property other than termination. Consider the numeric program in Figure 4.17. This is the same program that was on the right side of Figure 4.16, but with the two cases of the branch at L_2 split into their own continuations. This allows us to write LTSL formulae that specify which branch is taken.

One example of such a formula is the following, which states that it is always the case that after an execution visits label L_4, it eventually visits label L_3.

$$\mathbf{G}\big(\mathit{atloc}(\mathsf{L}_4) \supset \mathbf{F}(\mathit{atloc}(\mathsf{L}_3))\big)$$

If L_4 were associated with a *request* and L_3 with a *response*, then this formula would state that every request is eventually responded to.

Note that all of the properties we have considered are *universal* in that they hold if and only if they hold of all program traces. This is the nature of LTSL formulae. We cannot write statements in LTSL that describe existential path properties. An example of such a property is “there are traces in which $n > 1$ is true at L_1 but L_4 is never visited.” Since numeric programs are *over-approximations* of the original program, such existential properties are not necessarily preserved (it is possible that such a property could hold of the numeric program but not hold of the original program).

4.6 Summary

We now summarize what we have accomplished in this chapter, collecting and combining the various theorems into their most useful forms. We first showed how to associate an instrumented program with an original program. We can reason about the safety and liveness

Instrumented Program

```
$\mathsf{L}_0$ : $\underline{n := ?}$; $\underline{z := ?}$; goto $\mathsf{L}_1$
$\mathsf{L}_1$ : $y = x$.next;
    branch $y = x \Rightarrow$ $\underline{\mathsf{assume}(n = 1)}$;
                halt,
           $y \neq x \Rightarrow$ $\underline{\mathsf{assume}(n > 1)}$;
                $x' := x$.next;
                $t := x'$.data;
                goto $\mathsf{L}_2$ end
$\mathsf{L}_2$ : branch $t \leq v \Rightarrow$ $\underline{\mathsf{assume}(z \leq v)}$;
                $x$.next $:= x'$.next;
                free $x'$;
                $\underline{n := n - 1}$;
                $\underline{z := ?}$;
                goto $\mathsf{L}_1$,
           $t > v \Rightarrow$ $\underline{\mathsf{assume}(z > v)}$;
                $v := x'$.data;
                $\underline{\mathsf{assume}(v = z)}$;
                goto $\mathsf{L}_1$ end
```

Numeric Program

```
$\mathsf{L}_0$ : $\underline{n := ?}$; $\underline{z := ?}$; goto $\mathsf{L}_1$
$\mathsf{L}_1$ : branch true $\Rightarrow$ $\underline{\mathsf{assume}(n = 1)}$;
                halt,
           true $\Rightarrow$ $\underline{\mathsf{assume}(n > 1)}$;
                goto $\mathsf{L}_2$
    end
$\mathsf{L}_2$ : branch true $\Rightarrow$ $\underline{\mathsf{assume}(z \leq v)}$;
                $\underline{n := n - 1}$;
                $\underline{z := ?}$;
                goto $\mathsf{L}_1$,
           true $\Rightarrow$ $\underline{\mathsf{assume}(z > v)}$;
                $v := ?$;
                $\underline{\mathsf{assume}(v = z)}$;
                goto $\mathsf{L}_1$
    end
```

$$
\begin{aligned}
\Gamma(\mathsf{L}_0) &= \exists n.\ ls(n, x, x) \land n \geq 1 \\
\Gamma(\mathsf{L}_1) &= \big(\exists a, b, d.\ x \mapsto [\mathsf{next} : a, \mathsf{data} : d] * a \mapsto [\mathsf{next} : b, \mathsf{data} : z] * ls(n-2, b, x)\big) \\
&\qquad \lor \big(x \mapsto [\mathsf{next} : x, \mathsf{data} : z] \land n = 1\big) \\
\Gamma(\mathsf{L}_2) &= \exists a, b, d.\ (x \mapsto [\mathsf{next} : x', \mathsf{data} : d] * x' \mapsto [\mathsf{next} : b, \mathsf{data} : z] * ls(n-2, b, x)) \\
&\qquad \land z = t
\end{aligned}
$$

Figure 4.16: An instrumentation and projection of the program in Figure 4.14, with instrumentation variables n and z and projection variables n, z, v.

```
$\mathsf{L}_0$ : $\underline{n := ?}$; $\underline{z := ?}$; goto $\mathsf{L}_1$
$\mathsf{L}_1$ : branch  true $\Rightarrow$ $\underline{\mathsf{assume}(n = 1)}$;
                             halt,
                   true $\Rightarrow$ $\underline{\mathsf{assume}(n > 1)}$;
                             goto $\mathsf{L}_2$
       end
$\mathsf{L}_2$ :  branch true $\Rightarrow$ goto $\mathsf{L}_3$
                   true $\Rightarrow$ goto $\mathsf{L}_4$
       end
$\mathsf{L}_3$ : $\underline{\mathsf{assume}(z \leq v)}$;
       $\underline{n := n - 1}$;
       $\underline{z := ?}$;
       goto $\mathsf{L}_1$,
$\mathsf{L}_4$ : $\underline{\mathsf{assume}(z > v)}$;
       $v := ?$;
       $\underline{\mathsf{assume}(v = z)}$;
       goto $\mathsf{L}_1$
```

Figure 4.17: The numeric program from Figure 4.16, but rearranged so that the cases of the second branch are split into separate continuations.

behavior of the instrumented program and the properties satisfied by the instrumentation can be converted into properties that are satisfied by the original program.

Theorem 25. *Let* $Q_0 = \Gamma(initloc(P))$. *If* $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ *and* $\phi \in$ *LTSL then* $\llparenthesis \widehat{P} \mid Q_0 \rrparenthesis \models \phi$ *implies* $\llparenthesis P \mid Q_0 \rrparenthesis \models \boxed{\exists}(V, \phi)$.

Proof. This theorem is the result of combining Theorem 22, Theorem 18, Corollary 2, and Lemma 11. By Theorem 22 we have

$$\llparenthesis P \mid Q_0 \rrparenthesis \underset{\sim}{\sqsubset}_{R^{V,\Gamma},=_{\widetilde{V}}} \llparenthesis \widehat{P} \mid Q_0 \rrparenthesis$$

From Theorem 18 we then have

$$\mathit{traces}(\llparenthesis P \mid Q_0 \rrparenthesis) \lesssim_{=_{\widetilde{V}}} \mathit{traces}(\llparenthesis \widehat{P} \mid Q_0 \rrparenthesis)$$

If we let $V' = \mathit{fv}(\phi) - \widetilde{V}$, then Corollary 2 gives us

$$\llparenthesis P \mid Q_0 \rrparenthesis \models \boxed{\exists}(V', \phi)$$

To complete the proof we need only show that $V' \subseteq V$ and apply Lemma 11. To show this, suppose that $x \in V'$. Then $x \in \mathit{fv}(\phi)$ and $x \notin \widetilde{V}$. This last fact implies $x \in V$ (since $\widetilde{V}$ is the complement of V). This establishes $V' \subseteq V$. □

Instrumented programs let us introduce additional variables and commands and use these to prove properties of the original program. However, we will usually want to decompose the verification problem further, using projection to obtain a program that only involves integer-valued variables and then passing this program to an external verification tool. The following theorem states what we can conclude about the original program if we use such a method.

Theorem 26. *Let* $Q_0 = \Gamma(\mathit{initloc}(P))$*. If the following hold*

1. $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ *and* $\phi \in \mathit{LTSL}$ *and* $\llparenthesis \widehat{P} \mid Q_0 \rrparenthesis \models \phi$
2. $P' = \pi_{V'}(\widehat{P})$ *and* $\phi' \in \mathit{LTSLP}(V')$ *and* $\llparenthesis P' \mid Q_0 \rrparenthesis \models \phi'$

then $\llparenthesis P \mid Q_0 \rrparenthesis \models \boxed{\exists}(V, \phi \curlywedge \phi')$.

Proof. This theorem is primarily a combination of Theorem 23 and Theorem 25. Suppose condition 2 holds. Then by Theorem 23 we have that there is some relation R' such that $\llparenthesis \widehat{P} \mid Q_0 \rrparenthesis \mathrel{\underset{\sim}{\sqsubset}}_{R', \overset{s}{=}_{V'}} \llparenthesis P' \mid Q_0 \rrparenthesis$. By Theorem 18 we have $\llparenthesis \widehat{P} \mid Q_0 \rrparenthesis \lesssim_{\overset{s}{=}_{V'}} \llparenthesis P' \mid Q_0 \rrparenthesis$. By Theorem 16 we have that ϕ' is $\overset{s}{=}_{V'}$-invariant. Then by Corollary 1 we have that $\llparenthesis P' \mid Q_0 \rrparenthesis \models \phi'$ (which we have) implies $\llparenthesis \widehat{P} \mid Q_0 \rrparenthesis \models \phi'$. Since we also have $\llparenthesis \widehat{P} \mid Q_0 \rrparenthesis \models \phi$, we have $\llparenthesis \widehat{P} \mid Q_0 \rrparenthesis \models \phi \curlywedge \phi'$. This holds since for any trace T in $\mathit{traces}\llparenthesis \widehat{P} \mid Q_0 \rrparenthesis$, we have $T \models \phi$ and $T \models \phi'$, which according to the semantics of LTSL implies that $T \models \phi \curlywedge \phi'$.

Finally, we note that $\phi \curlywedge \phi'$ is an LTSL formula and thus Theorem 25 applied to $\llparenthesis \widehat{P} \mid Q_0 \rrparenthesis \models \phi \curlywedge \phi'$ and $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ gives us $\llparenthesis P \mid Q_0 \rrparenthesis \models \boxed{\exists}(V, \phi \curlywedge \phi')$. □

4.7 Conclusion

The instrumentation analysis given in the next section gives a method of automatically generating instrumented programs and thus numeric abstractions. But there are likely to be other approaches to instrumentation analysis that differ in their efficiency, completeness, and generality. Thus, one of the primary technical contributions of this thesis is that the rules given for checking $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ are sufficient to ensure that $\pi_{V'}(\widehat{P})$ simulates P. This gives a well-defined target for analyses that produce numeric abstractions of programs in much the same way that partial correctness proofs in Hoare logic provide a common target for safety analyses. In fact, the process of generating an instrumented program can be viewed as a generalization of the process of proving partial correctness. The invariants Γ that are required are valid partial correctness invariants, but the proving process is relaxed in the sense that, rather than only working with invariants, we are allowed to also insert instrumentation commands.

In this sense, the process is similar to program proving in Hoare logic with auxiliary variables, for example as described in [Owicki and Gries, 1976]. A major difference is due to the handling of non-determinism. Our INST-EXISTS rule lets us insert a command $x := ?$ when we have the precondition $\exists x.\ Q$ in order to reason from Q. And our INST-DISJ rule lets us insert branch true $\Rightarrow \ldots,$ true $\Rightarrow \ldots$ end when we have the precondition $Q_1 \vee Q_2$ in order to reason separately from Q_1 and Q_2. Such operations are not allowed in standard Hoare logic with auxiliary variables. The reason the two methods differ is that we are interested in properties preserved by simulation, which requires the existence of *some* transition with a given property, whereas Hoare logic for partial correctness is interested in properties that hold for *all* transitions. Another reason for the difference is that we are only translating one program to another, whereas Hoare logic is concerned with proving properties of programs. Once we have added the new commands to the program and turn our attention to the problem of proving program properties, we switch to a universal view of transitions, checking that a property holds of all paths.

One contribution of the approach we have taken in this chapter is the careful separation of the addition of auxiliary / instrumentation variables from the process of proving program properties. Once we start down this path, we see that the traditional restrictions on auxiliary variables are overly harsh. By relaxing these, we obtain rules that exhibit a novel correspondence between existential variables and non-deterministic assignment and between disjunction and non-deterministic choice.

Chapter 5

Instrumentation Analysis

In this chapter, we present an automated algorithm for generating instrumented programs of the form given in Chapter 4. We call such an automated procedure an *instrumentation analysis*. The algorithm proceeds by performing a shape analysis on the program, which enables it to discover an appropriate mapping Γ for the proof that $\Gamma \vdash \widehat{P} \blacktriangleright_V P$. During the analysis process, the algorithm also inserts instrumentation commands at certain points in order to record information about numeric properties. The syntax-directed projection operation presented in Section 4.4 can then be used to generate a numeric program from the instrumented program produced by the instrumentation analysis. We have implemented this algorithm in a tool called THOR [Magill et al., 2008], which is able to generate numeric abstractions of C programs using the techniques described in this thesis.

The portion of the analysis that is concerned with the generation of Γ can be described as an abstract interpretation [Cousot and Cousot, 1977] where the abstract domain consists of separation logic formulae of a restricted form. However, familiarity with abstract interpretation will not be required in order to understand the presentation of the algorithm that we provide here. While we will use some terms from the abstract interpretation framework, we will describe the algorithm in terms of our goal of generating instrumented programs according to the rules in Chapter 4. For a description of this style of shape

$$
\begin{array}{rlll}
\textit{Inductive Predicates} & d & \in & \mathcal{P} \\
\textit{Records} & \rho & ::= & \epsilon \mid f^{\tau} : e^{\tau},\ \rho \\
\textit{Spatial Predicates} & \Xi & ::= & \mathbf{emp} \mid e^{\mathrm{a}} \mapsto [\rho] \mid d(\vec{e}) \\
\textit{Spatial Formulae} & \Sigma & ::= & \Xi \mid \Sigma * \Sigma \\
\textit{Pure Formulae} & \Pi & ::= & \mathsf{true} \mid \mathsf{false} \mid e^{\mathrm{a}}_1 = e^{\mathrm{a}}_1 \mid e^{\mathrm{i}}_1 \leq e^{\mathrm{i}}_2 \mid \neg\Pi \mid \Pi_1 \wedge \Pi_2 \\
\textit{Symbolic State Formulae } (\Phi) & \varphi & ::= & \exists\vec{x}.\ \Sigma \wedge \Pi
\end{array}
$$

Figure 5.1: Restricted subset of separation logic formulae. The notation $\vec{x}$ indicates a list of variables $x_1, x_2, \ldots, x_n$ and $\exists\vec{x}.\ Q$ is shorthand for $\exists x_1.\exists x_2.\ldots\exists x_n.\ Q$.

analysis in abstract interpretation terms, see [Distefano et al., 2006] and [Berdine et al., 2007].

We begin our discussion by describing the restricted form of separation logic formulae used by the automated analysis.

5.1 Symbolic State Formulae

Figure 5.1 gives the restricted set of separation logic formulae used in the automated analysis. Working in this subset simplifies the theorem proving problem that we discuss in Section 5.5 and also results in simple predicate transformers for the commands in our language. We write $\vec{x}$ to represent a list of variables $x_1, x_2, \ldots, x_n$. We will implicitly convert these ordered lists into unordered sets as needed when stating certain properties. Such conversions will be obvious due to the set notation used. For example, $\vec{x} \cup \vec{y}$ represents the set consisting of the elements of $\vec{x}$ together with those in $\vec{y}$. The notation $y \in \vec{x}$ indicates that y is a member of the set consisting of the elements of $\vec{x}$.

We would like to identify formulae that are logically equivalent. However, logical equivalence of separation logic formulae cannot always be accurately determined.[1] For

[1] The undecidability of separation logic formulae, as we have defined them, follows from the fact that they contain the integers with addition, multiplication, and existential quantification as a fragment of the

$$\frac{}{\Sigma * \mathit{emp} \equiv \Sigma} \qquad \frac{}{\exists \vec{x_1}, x, \vec{x_2}.\ \Sigma \wedge \Pi \equiv \exists \vec{x_1}, x', \vec{x_2}.\ \Sigma[x'/x] \wedge \Pi[x'/x]}\ (x' \notin \mathit{fv}(\Sigma, \Pi))$$

$$\frac{}{\exists \vec{x_1}, x, x', \vec{x_2}.\ \Sigma \wedge \Pi \equiv \exists \vec{x_1}, x', x, \vec{x_2}.\ \Sigma \wedge \Pi} \qquad \frac{}{\Sigma_1 * \Sigma_2 \equiv \Sigma_2 * \Sigma_1}$$

$$\frac{}{\Sigma_1 * (\Sigma_2 * \Sigma_3) \equiv (\Sigma_1 * \Sigma_2) * \Sigma_3} \qquad \frac{\Sigma_1 \equiv \Sigma_2}{\exists \vec{x}.\ \Sigma_1 \wedge \Pi \equiv \exists \vec{x}.\ \Sigma_2 \wedge \Pi}$$

Figure 5.2: Equivalence relation for symbolic state formulae.

this reason, our implementation may distinguish some formulae that are actually equivalent. This does not affect soundness of the approach, but can affect completeness. We assume that the implemented equivalence check at least identifies formulae that are related by the equivalence relation given in Figure 5.2. This considers formulae equivalent up to commutativity and associativity of $*$, the unit law for **emp**, renaming of quantified variables, and re-ordering of existential quantifiers.

The set Φ is closed with respect to $*$ in the sense that the $*$-conjunction $\varphi * \varphi'$ of elements of Φ is semantically equivalent to an element $\varphi'' \in \Phi$ (according to the semantics given in Figure 2.7). The element φ'' is defined as follows. Let $\varphi = \exists \vec{v}.\ \Sigma \wedge \Pi$ and $\varphi' = \exists \vec{v}\,'.\ \Sigma' \wedge \Pi'$ such that $\mathit{fv}(\Sigma \wedge \Pi) \cap \vec{v}\,' = \emptyset$ and $\mathit{fv}(\Sigma' \wedge \Pi') \cap \vec{v} = \emptyset$ (these constraints can always be satisfied by renaming quantified variables). Then we have the following

$$\varphi * \varphi' \Leftrightarrow \exists \vec{v}, \vec{v}\,'.\ (\Sigma * \Sigma') \wedge (\Pi \wedge \Pi')$$

and this is in Φ.

Similarly, Φ is closed with respect to conjunction of pure formulae (for all $\varphi \in \Phi$ there is a $\varphi' \in \Phi$ such that $(\varphi \wedge \Pi) \Leftrightarrow \varphi'$). These operations will be used freely with the

logic. Decidability of this fragment is Hilbert's 10th problem and was shown to be undecidable by Davis, Matiyasevich, Putnam, and Robinson. Decidability of fragments of the logic not including multiplication has been explored to some extent by [Berdine et al., 2004] and [Bozga et al., 2008], but much is still unknown.

understanding that they refer not to a general separation logic formula that falls outside of Φ, but rather to the element of Φ semantically equivalent to that formula.

5.2 Inductive Predicate Specifications

In order to reason about data structures, our tool incorporates support for *inductive predicate specifications*. We use the term “specification” rather than “definition” deliberately, as these specifications differ from definitions in two key ways.

First, the syntax for specifications adds additional structure beyond that present in definitions. This structure serves to separate the instrumentation variables from the program variables in a way that simplifies automatic reasoning.

Secondly, we allow multiple specifications for the same predicate name, whereas only a single definition for each name was permitted in Section 2.2.2. This allows inductive consequences of definitions to be provided to the tool. Such consequences cannot be inferred by the tool, as the automated analysis does not perform inductive reasoning. Allowing multiple specifications for the same predicate has implications for the semantics of specifications, and we will formally connect this semantics to the semantics of definitions given previously. One consequence of this decision to allow multiple specifications is that it provides opportunity for the user to introduce inconsistency into the system. We address this concern with Theorem 27 on page 198.

Syntax

The syntax for inductive specifications is given in Figure 5.3. A predicate specification has the following form.

$$d(\underline{\vec{x}}; \vec{y}) \ \texttt{<=>} \ C_1(\underline{\vec{x}}; \vec{y}) \mid \ldots \mid C_n(\underline{\vec{x}}; \vec{y})$$

The variable d is the name of the inductive predicate we are specifying. The variables to the left of the semicolon, $\underline{\vec{x}}$, are referred to as *instrumentation parameters*. These

Predicate Names	d	$\in$	$\mathcal{P}$
Inductive Specification	S_d	::=	$d(\underline{\vec{x}};\vec{y})$ <=> $C_1(\underline{\vec{x}};\vec{y})$ '|' ... '|' $C_n(\underline{\vec{x}};\vec{y})$
Case	$C(\underline{\vec{x}};\vec{y})$	::=	Π : let $\underline{\vec{z}}$ satisfy Π' in φ
			where $fv(\Pi) \subseteq \underline{\vec{x}}$ and $fv(\Pi') \subseteq (\underline{\vec{x}} \cup \underline{\vec{z}})$
			and $fv(\varphi) \subseteq (\vec{y} \cup \underline{\vec{z}})$ and $\underline{\vec{x}}, \vec{y}, \vec{z}$ distinct and disjoint

Figure 5.3: Syntax of inductive specifications as implemented in THOR. The notation '|' is used to indicate the literal character |, and distinguish it from the BNF grammar operator consisting of the same symbol.

parameters represent integer-valued quantities that we want our analysis to track with instrumentation variables—for example, the length of a list or the height of a tree. We will underline instrumentation parameters to help the reader identify them. The C_i are *cases* of the definition and have the following form.

$$\Pi : \text{let } \underline{\vec{z}} \text{ satisfy } \Pi' \text{ in } \varphi$$

The pure condition Π is a constraint on the instrumentation parameters $\underline{\vec{x}}$ which gives the condition that differentiates this case from the others. Often the Π_i in the cases of a definition will be non-overlapping in the sense that for any i, j we have $\Pi_i \wedge \Pi_j \Rightarrow \mathsf{false}$. For example, in the definition of a list of length $\underline{n}$, we might have $\underline{n} = 0$ and $\underline{n} > 0$ as our two conditions. However, this disjointness of conditions is not a requirement. For example, a list predicate that does not track list length would simply have true for the condition in both the base case and the inductive case.

Before explaining the rest of the syntax, it is helpful to consider a concrete example. Figure 5.4 shows a graphical depiction of a doubly-linked list segment. The inductive specification for this segment is given below. The syntax [ ] represents an empty list.

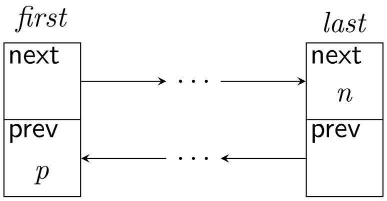


Figure 5.4: Graphical depiction of the doubly-linked list segment predicate.

$$\begin{aligned}
&\mathrm{dll}(\underline{k};\, p, \mathit{first}, \mathit{last}, n)\ \texttt{<=>}\\
&\quad \underline{k} = 0 : \text{let } [\,] \text{ satisfy } \mathsf{true} \text{ in } \mathbf{emp} \land \mathit{first} = n \land \mathit{last} = p\\
&\quad \mid\ \underline{k} > 0 : \text{let } \underline{k}' \text{ satisfy } \underline{k} = \underline{k}' + 1 \text{ in}\\
&\qquad\qquad\qquad \exists z.\ (\mathit{first} \mapsto [\mathsf{prev} : p, \mathsf{next} : z]) * \mathrm{dll}(\underline{k}'; \mathit{first}, z, \mathit{last}, n))
\end{aligned}$$

The parameters first and last are the addresses of the first and last cells in the list segment. The parameter p is the contents of the prev field of the first element and the n parameter is the address value contained in the next field of the last element of the segment. The parameter $\underline{k}$ is the length of the list.

The specification can be read as saying that there are two possible cases for a list segment with length $\underline{k}$. Either $\underline{k} = 0$, in which case the list is empty, or $\underline{k} > 0$, in which case the list is non-empty.

In the non-empty case, the sub-formula

$$\exists z.\ (\mathit{first} \mapsto [\mathsf{prev} : p, \mathsf{next} : z]) * \mathrm{dll}(\underline{k}'; \mathit{first}, z, \mathit{last}, n))$$

indicates that the list can be split into the head element, given by the formula $\mathit{first} \mapsto [\mathsf{prev} : p, \mathsf{next} : z]$ and the tail of the list, given by $\mathrm{dll}(\underline{k}'; \mathit{first}, z, \mathit{last}, n)$. This tail portion of the list has length $\underline{k}'$. The rest of this case of the specification is concerned with relating $\underline{k}$ (the length of the full list segment) and $\underline{k}'$ (the length of the sub-segment).

After the keyword "let," a list of variables can appear. These are the variables that appear as instrumentation parameters in recursive instances of inductive predicates in the body of the case. Returning to our general syntax, reproduced below,

$$C(\underline{\vec{x}}; \vec{y}) ::= \Pi : \text{let } \underline{\vec{z}} \text{ satisfy } \Pi' \text{ in } \varphi$$

the list $\underline{z}$ gives the variables that will be passed as instrumentation parameters to inductive predicates appearing in φ. The formula Π' then relates $\underline{z}$ to the instrumentation parameters for the predicate being specified, which are given by $\underline{\vec{x}}$. In our doubly-linked list example, Π' for the non-empty case is $\underline{k} = \underline{k}' + 1$. Since the empty case contains no instances of inductive predicates, the list of variables in that case in empty. This is the role of the $[\,]$ syntax—it represents an empty list.

To summarize, new variables will be added by our instrumentation analysis and used to track quantities like the length of a list or the size of a tree. The specification of an inductive predicate gives a list of possible expansions. Each expansion may expose substructures which themselves have quantities to be tracked. The list $\underline{\vec{z}}$ contains the variables representing these new quantities and each Π' gives a relation between the variables in $\underline{\vec{x}}$ (the sizes passed into this predicate instance) and those in $\underline{\vec{z}}$ (the sizes passed to recursive instances of the predicate). This relation is represented as an expression over variables in $\underline{\vec{x}} \cup \underline{\vec{z}}$.

Syntactic Connection with Inductive Definitions

Individual specifications are very closely related to individual inductive definitions. In fact, they differ only in syntax. Consider the specification below.

$$d(\underline{\vec{x}}; \vec{y}) \texttt{ <=> } C_1(\underline{\vec{x}}; \vec{y}) \mid \ldots \mid C_n(\underline{\vec{x}}; \vec{y})$$

Let $\langle C_i \rangle$ be defined such that if C_i is Π : let $\underline{\vec{z}}$ satisfy Π' in φ, then $\langle C_i \rangle \overset{\text{def}}{=} \Pi \wedge \exists \underline{\vec{z_n}}.\ (\Pi' \wedge \varphi)$. Then the specification above corresponds to the definition below.

$$d(\underline{\vec{x}}, \vec{y}) \equiv \langle C_1(\underline{\vec{x}}; \vec{y}) \rangle \mid \ldots \mid \langle C_n(\underline{\vec{x}}; \vec{y}) \rangle$$

We will write $\langle S \rangle$ to denote the translation of specification S to the syntax for definitions. We also generalize this to sets of specifications. Let $\mathbf{S} = \{S_1, \ldots, S_n\}$ be a set of inductive specifications. Then $\langle \mathbf{S} \rangle = \langle S_1 \rangle :: \ldots :: \langle S_n \rangle$ (where :: separates the elements in a list of inductive definitions as used in Section 2.2.2). Note that while the translation of a single specification is always a well-formed definition, the translation of a set of specifications will not be a valid list of definitions if there are multiple specifications for the same predicate name.

Multiple Specifications

Note that the specification of a doubly-linked list segment given previously is "front-biased," in that the heap cell exposed in the inductive case is at the front of the list. As we will see when we describe our instrumentation algorithm, this will result in the specification being useless for exposing cells at the back of the list, which is often necessary. Multiple specifications solve this problem by providing multiple ways of viewing a data structure. These various views are then all available for use during the analysis. An example of a specification for accessing a doubly-linked list from the back is given below.

$$\begin{aligned}
&\mathrm{dll}(\underline{k};\, p, \mathit{first}, \mathit{last}, n) \ \texttt{<=>} \\
&\qquad \underline{k} = 0 : \text{let } [\,] \text{ satisfy } \mathsf{true} \text{ in } \mathbf{emp} \wedge \mathit{first} = n \wedge \mathit{last} = p \\
&\quad\ \mid\ \underline{k} > 0 : \text{let } \underline{k}' \text{ satisfy } \underline{k} = \underline{k}' + 1 \text{ in} \\
&\qquad\qquad\qquad \exists z.\ \mathrm{dll}(\underline{k}';\, p, \mathit{first}, z, \mathit{last}) * (\mathit{last} \mapsto [\mathsf{prev} : z, \mathsf{next} : n])
\end{aligned}$$

Unlike the previous specification, here the inductive case involves exposing the points-to predicate at the end of the list segment. These specifications are equivalent in the sense that, if they are taken as definitions, they define the same set of structures. In fact, we can use induction on the length of the list to show that each definition implies the other.

However, it does not have to be the case that all specifications of a given predicate are equivalent. Consider the specification below, which lets us view a list segment as consisting of two sub-segments.

$$\begin{aligned}
&\mathrm{dll}(\underline{k};\, p, \mathit{first}, \mathit{last}, n) \ \texttt{<=>} \\
&\qquad \mathsf{true} : \text{let } \underline{k}_1, \underline{k}_2 \text{ satisfy } \underline{k} = \underline{k}_1 + \underline{k}_2 \text{ in} \\
&\qquad\qquad\qquad \exists x, y.\ \mathrm{dll}(\underline{k}_1;\, p, \mathit{first}, x, y) * \mathrm{dll}(\underline{k}_2;\, x, y, \mathit{last}, n)
\end{aligned}$$

This specification is not equivalent to either of the other two. In fact, taken on its own as a definition, it has multiple fixed-points, the least of which is the empty set of heaps—clearly not the same set defined by the other specifications.

However, the specification above is compatible with the others in the sense that, if we take the forward or backward-oriented specification as our definition of dll, then the specification above can be proved valid. Informally speaking (since we have not yet defined the semantics of specifications), we have that the forward and backward specifications imply the splitting specification above, but neither of the reverse implications hold. In Theorem 27 we formalize this idea of using some subset of the specifications to justify the others.

Semantics

In Definition 6, we gave the semantics of a set of inductive definitions. Inductive definition sets have the restriction that each predicate symbol must appear at most once on the left-hand side of a definition. We have no such restriction for specifications. In fact, a primary reason we introduce specifications is so that we can provide multiple specifications for a single predicate symbol. As such, the method of specifying semantics developed in Theorem 8 is more appropriate here, as it is straightforward to generalize characteristic formulae (Definition 10) in order to reduce the restrictions on where predicate symbols may occur.

When we are provided with multiple specifications for a single predicate symbol, we require that they all hold. The meaning of a single specification S is given by the characteristic formula (Definition 10) associated with the translation of S to a definition. This is given by $\lceil\langle S\rangle\rceil$. The meaning of multiple specifications is then the conjunction of these formulas $\bigwedge_{S\in\mathbf{S}}\lceil\langle S\rangle\rceil$, which we abbreviate as $\lceil\mathbf{S}\rceil$. Formally, we have the following.

Definition 33. *Let* $\mathbf{S}$ *be a set of specifications and let* $dom(\mathbf{S})$ *give the set of predicate names appearing on the left-hand side of "* <=> *" in specifications in* $\mathbf{S}$*. A store, heap pair* s, h *satisfy separation logic formula* Q *given* $\mathbf{S}$*, written* $(s, h) \models^{\mathbf{S}} Q$*, if and only if* $(s, h) \models_X Q$ *for all* $X \in \Delta_{dom(\mathbf{S})}$ *such that* $\models_X \lceil\mathbf{S}\rceil$.

When each predicate name in $dom(\mathbf{S})$ appears to the left of <=> in at most one specification, then each predicate name is defined at most once by $\langle\mathbf{S}\rangle$ and so $\langle\mathbf{S}\rangle$ is a valid list of definitions. In this case, our definition of satisfaction for specifications (Definition 33) coincides with our definition of satisfaction for definitions (Definition 6) and we have

$(s,h) \models^{\mathbf{S}} Q$ if and only if $(s,h) \models^{\langle\mathbf{S}\rangle} Q$. This follows immediately from Definition 33 and Theorem 8.

Even when we have multiple specifications, we can still relate Definition 33 to Definition 6 by taking some subset of the specifications as predicate definitions and showing that these definitions imply the remaining specifications, as demonstrated by the following theorem. Of course, even when the theorem below does not apply, the semantics of specifications are still well-defined by Definition 33.

Theorem 27. *Consider a set of specifications* $\mathbf{S}$ *and a subset* $\mathbf{S}' \subseteq \mathbf{S}$ *such that* $\langle\mathbf{S}'\rangle$ *is a valid set of inductive definitions (no predicate name is defined more than once) and* $\mathit{dom}(\mathbf{S}) = \mathit{dom}(\mathbf{S}')$. *If* $\models^{\langle\mathbf{S}'\rangle} \lceil\mathbf{S}\rceil$ *then for all* Q *we have* $(s,h) \models^{\mathbf{S}} Q$ *implies* $(s,h) \models^{\langle\mathbf{S}'\rangle} Q$.

Proof. Suppose $(s,h) \models^{\mathbf{S}} Q$ holds. Applying the definition of $\models^{\mathbf{S}}$ gives us the following.

$$(s,h) \models_X Q \text{ for all } X \in \Delta_{\mathit{dom}(\mathbf{S})} \text{ such that } \models_X \lceil\mathbf{S}\rceil \tag{5.1}$$

We must show $(s,h) \models^{\langle\mathbf{S}'\rangle} Q$. We have $\models^{\langle\mathbf{S}'\rangle} \lceil\mathbf{S}\rceil$, which by Theorem 8 implies the following.

$$\models_X \lceil\mathbf{S}\rceil \text{ for all } X \in \Delta_{\mathit{dom}(\langle\mathbf{S}'\rangle)} \text{ such that } \models_X \lceil\langle\mathbf{S}'\rangle\rceil \tag{5.2}$$

Note that $\mathit{dom}(\mathbf{S}) = \mathit{dom}(\langle\mathbf{S}'\rangle)$ and thus we can combine (5.1) and (5.2), obtaining the following.

$$(s,h) \models_X Q \text{ for all } X \in \Delta_{\mathit{dom}(\mathbf{S}')} \text{ such that } \models_X \lceil\langle\mathbf{S}'\rangle\rceil$$

Again applying Theorem 8, we have $(s,h) \models^{\langle\mathbf{S}'\rangle} Q$, which was our goal. □

Besides connecting satisfaction involving inductive specifications to satisfaction involving inductive definitions, the theorem above also provides a means to ensure that the use of multiple specifications does not introduce inconsistency into the system. The premise of the theorem requires that a subset of the specifications can be taken as a set of definitions and these definitions imply the validity of the other specifications. If this holds, then the fact that each set of inductive definitions has a least fixed-point (Theorem 4) guarantees that the system remains consistent.

THOR does not check that the premise of the theorem above holds of the inductive specifications provided. Thus, if use of the theorem is desired, the premise must be verified by the user via other means. One option is to employ a system such as that given in [Nguyen and Chin, 2008], which provides support for formally proving separation logic implications involving inductive definitions and in many cases allows for automation of such proofs.

5.3 Basic Types

Figure 5.5 lists the types used by the algorithm and the meta-variables used for terms of these types. The type “τ option” is the type of optional values of type τ. That is, a value of type “τ option” may either be $\mathsf{Some}(a)$ for some a of type τ or it may be None.

Note that we have two types of variable—one that is used for program variables and another that is used for instrumentation variables. In the following presentation we will use underlines to indicate that a variable is of type $\underline{\mathrm{IVar}}$. Non-underlined variables x, y, z and their subscripted forms denote program variables. Either type of variable can appear quantified. The type Gen of instrumentation generators is dependent on a continuation k of type K. This is used in stating the specification that these functions must satisfy. This specification (as well as specifications for the other functions used by the implementation) is given in Figures 5.6 and 5.7.

In the implementation, these different classes of variable are maintained as separate types. However, the syntax and semantics of separation logic formulae and of programs and instrumented programs was given in terms of a single set of variables, *Vars*. Thus, when stating theorems about the implementation presented here, we need some way of encoding these separate types. We will model them as disjoint subsets of the set *Vars*. To support this set-based interpretation, we will sometimes use the name of one of these types to represent the set of variables of that type. So the statement $\underline{x} \in \underline{\mathrm{IVar}}$ should be read as saying that $\underline{x}$ is a variable in the subset of *Vars* corresponding to the type $\underline{\mathrm{IVar}}$ in the implementation.

E = The type of expressions e as defined in Figure 2.1.

E list = The type of lists of expressions.

C = The type of commands c as defined in Figure 2.1.

C list = The type of lists of commands, represented by the meta-variable $\mathbf{c}$.

K = The type of continuations k as defined in Figure 2.1.

$\widehat{\mathrm{K}}$ = The type of instrumented continuations $\widehat{k}$. These are drawn from the same language as values of type K, but are assigned their own type for clarity.

$\mathbb{P}$ = The type of programs P as defined in Figure 2.1.

$\widehat{\mathbb{P}}$ = The type of instrumented programs $\widehat{P}$. These are drawn from the same language as values of type $\mathbb{P}$, but are assigned their own type for clarity.

Φ = The type of symbolic state formulae φ as defined in Figure 5.1.

G = The type of contexts Γ. Equal to *Labels* $\rightarrow (\Phi\ \mathrm{set})$.

$\mathrm{Gen}(k : \mathrm{K})$ = The type of functions f_k, which are *instrumentation generators for continuation* k. These are functions of type $\Phi \rightarrow (\mathrm{G} \times \widehat{\mathrm{K}})$ option that additionally satisfy the specification given in Figure 5.6.

Var = The type of program variables, $x, y, z, x_1, y_1, z_1, \ldots$

$\underline{\mathrm{IVar}}$ = The type of instrumentation variables, $\underline{x}, \underline{y}, \underline{z}, \underline{x}_1, \underline{y}_1, \ldots$.

Figure 5.5: Types used by the instrumentation algorithm.

Values of type G fill the same role as the contexts Γ from Chapter 4. In that chapter, we defined Γ to be a function of type *Labels* $\rightarrow Q$ (a mapping from labels to separation logic formulae). In the implementation, we work with elements of Φ instead of arbitrary separation logic formulae. Since elements of Φ do not contain disjunction, but disjunction is generally necessary to express the invariants in Γ, we let values of type G be functions of type *Labels* $\rightarrow \Phi$ set (mappings from labels to sets of formulae drawn from Φ). The sets in the range are interpreted disjunctively, so the set $\{\varphi_1, \varphi_2, \varphi_3\}$ corresponds to the separation logic formula $\varphi_1 \vee \varphi_2 \vee \varphi_3$.

The implementation also uses lists of commands in certain places. These are represented by the meta-variable $\mathbf{c}$ and the type of such command lists is “C list.” We use standard syntax for lists, writing $[c_1, \ldots, c_n]$ to represent a list of commands, $[\,]$ to represent the empty list, and $c ::\ \mathbf{c}$ to represent the cons operator. We define below an operation that sequences a list of commands with a continuation.

$$\begin{aligned} (c ::\ \mathbf{c}) \fatsemi k &\overset{\text{def}}{=} c\boldsymbol{;}\ (\mathbf{c} \fatsemi k) \\ \epsilon \fatsemi k &\overset{\text{def}}{=} k \end{aligned}$$

5.4 Basic Structure

Figures 5.6 and 5.7 provide a guide to the functions used in the implementation. For each function, we list the type of the function and the formal specification that it must satisfy. The functions all return optional values. The option type is used throughout because each operation in the analysis is partial. The problems we are solving are undecidable in general and so sometimes a solution will not be found. It is also the case that sometimes a solution just does not exist. Our instrumentation system only allows us to derive instrumentations for programs that are memory safe. So if a program is not memory safe, no implementation of the system described in this thesis would be able to produce an instrumented version of that program. This restriction to memory-safe programs arises as a consequence of the COMMAND rule in Figure 4.1, which requires that for every command c in the original program, we can derive the partial correctness triple $\{Q\}\ c\ \{Q'\}$, where Q is the current precondition. Since partial correctness ensures memory safety in separation logic, such a triple is only derivable if c is memory safe.

If the instrumentation process gets stuck and cannot make progress in the analysis, it will return a result of None. All functions called by the main procedure for the analysis (which is called `instrument`) are also allowed to return None and will do so as soon as a command is encountered whose safety cannot be shown. Once this occurs, the value None propagates up the call stack until it is eventually returned by the `instrument` procedure.

Undecidability of the problems involved can also manifest as non-termination. For example, the implementation includes a theorem prover for showing implications between symbolic state formulae. This problem is undecidable and, as a result, it is possible for

Function name and type	Specification
$f_k : \mathrm{Gen}(k)$	If $f_k(\varphi) = \mathsf{Some}\big(\Gamma, \widehat{k}\big)$ then $\Gamma \vdash \{\varphi\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$
`instrument` $: \Phi \times \mathbb{P} \to (\mathrm{G} \times \widehat{\mathbb{P}})$ option	If `instrument`$(\varphi_0, P) = \mathsf{Some}\big(\Gamma, \widehat{P}\big)$ then $\Gamma \vdash \widehat{P} \blacktriangleright_{\underline{\mathrm{IVar}}} P$ and $\varphi_0 \in \Gamma(\mathit{initloc}(P))$
`geninstCont` $: \mathrm{G} \times \Phi \times \mathrm{K} \to (\mathrm{G} \times \widehat{\mathrm{K}})$ option	If `geninstCont`$(\Gamma, \varphi, k) = \mathsf{Some}\big(\Gamma', \widehat{k}\big)$ then $\Gamma' \vdash \{\varphi\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$ and $\forall l.\, \Gamma'(l) \supseteq \Gamma(l)$
`partialPost` $: \Phi \times \mathrm{C} \to \Phi$ option	If `partialPost`$(\varphi, c) = \mathsf{Some}\big(\varphi'\big)$ then $\{\varphi\}\ c\ \{\varphi'\}$
`instPost` $: \Phi \times \mathrm{C} \times \mathrm{Gen}(k) \to$ $(\mathrm{G} \times \widehat{\mathrm{K}})$ option	If `instPost`$(\varphi, c, f_k) = \mathsf{Some}\big(\Gamma, \widehat{k}\big)$ then $\Gamma \vdash \{\varphi\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} (c\mathtt{;}\, k)$

Figure 5.6: A summary of the primary functions involved in the implementation.

an implication to hold but for the theorem prover to fail to show this. If this occurs for an implication that was crucial for construction of the instrumentation proof, the analysis will diverge.

5.4.1 `instrument`

At the highest level of the implementation, we have a function `instrument` of type $\Phi \times \mathbb{P} \to (\mathrm{G} \times \widehat{\mathbb{P}})$ option. A call to `instrument`(φ_0, P) takes the following arguments.

Function name and type	Specification
branchAnnot $: \Phi \times (\mathrm{E\ list}) \to \mathrm{E\ list}$	If branchAnnot$(\varphi, [e_1, \ldots, e_n]) = [e'_1, \ldots, e'_n]$ then $\forall i.\ (\varphi \wedge e_i \Rightarrow e'_i)$
implies $: \Phi \times \Phi \times \widehat{\mathrm{K}} \to \widehat{\mathrm{K}}$ option	If implies$(\varphi, , \varphi', \widehat{k}\,') = \mathsf{Some}\big(\widehat{k}\big)$ then for all Γ, k $\Gamma \vdash \{\varphi'\}\ \widehat{k}\,' \blacktriangleright_{\underline{\mathrm{IVar}}} k$ implies $\Gamma \vdash \{\varphi\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$
exposeCellThenInst $: \Phi \times \mathrm{Var} \times \mathrm{Gen}(k) \to$ $(\mathrm{G} \times \widehat{\mathrm{K}})$ option	If exposeCellThenInst$(\varphi, x, f_k) = \mathsf{Some}\big(\Gamma, \widehat{k}\big)$ then $\Gamma \vdash \{\varphi\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$
abstract $: \Phi \to \Phi \times (\mathrm{C\ list})$	If abstract$(\varphi) = (\varphi', \mathbf{c})$ then for all $\Gamma, k, \widehat{k}\,'$ $\Gamma \vdash \{\varphi'\}\ \widehat{k}\,' \blacktriangleright_{\underline{\mathrm{IVar}}} k$ implies $\Gamma \vdash \{\varphi\}\ (\mathbf{c} \mathbin{;} \widehat{k}\,') \blacktriangleright_{\underline{\mathrm{IVar}}} k$

Figure 5.7: Additional functions used by the implementation. These are primarily concerned with reasoning about implications between symbolic state formulae.

P The program to be analyzed.

φ_0 The precondition under which to analyze P.

It optionally returns a context Γ and an instrumented program $\widehat{P}$ such that the following holds.

$$\Gamma \vdash \widehat{P} \blacktriangleright_{\underline{\mathrm{IVar}}} P$$

If the algorithm cannot find a Γ, $\widehat{P}$ such that this relation holds, then `instrument` returns None.

In the property above, we make use of $\underline{\mathrm{IVar}}$, the set of all instrumentation variables. In practice, any program uses only a finite subset of these. According to Theorem 19, we can reduce the number of variables used in the statement above to $V' = \mathit{fv}(\widehat{P}) - \mathit{fv}(P)$, obtaining the following.

$$\Gamma \vdash \widehat{P} \blacktriangleright_{V'} P$$

Recall that the role of Γ in the instrumentation rules in Figure 4.1 was to give invariants of the program at each label. The instrumentation analysis has to automatically infer such a Γ, which is akin to inferring loop invariants. It also has to determine which instrumentation commands should be added.

The code for the `instrument` function is given on page 205. It consists of two loops, where the first loop is focused on generating Γ and the second loop performs the instrumentation. This separation of concerns aids in the explanation of the algorithm, but does cause us to recompute values that have already been produced. The results of function calls (most crucially `geninstCont`) can easily be cached to avoid such duplicate effort.

The `instrument` function, as well as subsequent functions, make use of a union operation on contexts, defined as follows.

$$(\Gamma_1 \cup \Gamma_2)(l) = \Gamma_1(l) \cup \Gamma_2(l)$$

The `instrument` function processes the program by passing each continuation to the `geninstCont` function. `geninstCont` has type $\mathrm{G} \times \Phi \times \mathrm{K} \to (\mathrm{G} \times \widehat{\mathrm{K}})$ option. It

Function `instrument`(φ_0, P). Main function of the instrumentation analysis.

```
/* Set precondition of initial location to $\varphi_0$ */
$\Gamma_{\text{new}} := \{(l_0, \{\varphi_0\})\} \cup \{(l, \emptyset) \mid l \in dom(P) \wedge l \neq l_0\}$
/* Analyze continuations until a fixed-point on $\Gamma_{\text{new}}$ is
   reached. */
repeat
  $\Gamma_{\text{old}} := \Gamma_{\text{new}}$
  foreach $l \in dom(P)$ do
    foreach $\varphi \in \Gamma_{\text{new}}(l)$ do
      match geninstCont($\Gamma_{\text{new}}, \varphi, P(l)$) with
        case Some$(\Gamma, \widehat{k})$
          $\Gamma_{\text{new}} := \Gamma$
        case None
          return None                /* possible memory fault */
      end
until $\Gamma_{\text{new}} = \Gamma_{\text{old}}$
/* Generate instrumentations of all continuations
   starting from the invariants stored in $\Gamma_{\text{new}}$ */
foreach $l \in dom(P)$ do
  let $\{\varphi_1, \varphi_2, \ldots, \varphi_n\} = \Gamma_{\text{new}}(l)$ in
  let Some$(\Gamma_1, \widehat{k}_1)$ = geninstCont($\Gamma_{\text{new}}, \varphi_1, P(l)$) in
  let Some$(\Gamma_2, \widehat{k}_2)$ = geninstCont($\Gamma_{\text{new}}, \varphi_2, P(l)$) in
    $\vdots$
  let Some$(\Gamma_n, \widehat{k}_n)$ = geninstCont($\Gamma_{\text{new}}, \varphi_n, P(l)$) in
    $\widehat{P}(l) :=$ (branch true $\Rightarrow \widehat{k}_1$, true $\Rightarrow \widehat{k}_2, \ldots,$ true $\Rightarrow \widehat{k}_n$ end)
end
return $(\Gamma_{\text{new}}, \widehat{P})$
```

takes a context Γ, a symbolic state formula representing a precondition φ_0 and a continuation k and optionally returns an instrumented continuation $\widehat{k}$ together with a new context Γ' mapping labels to symbolic state formulae. The context Γ describes the invariants at locations that the analysis has discovered thus far. The returned context Γ' is Γ extended with information about the states reachable through k. Formally, if `geninstCont`(Γ, φ_0, k) returns $\mathsf{Some}\big(\Gamma', \widehat{k}\big)$ then these should satisfy

$$\Gamma' \vdash \{\varphi_0\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

It will also be the case that $\forall l.\ \Gamma'(l) \supseteq \Gamma(l)$. That is, Γ' is an *extension* of Γ obtained by adding more disjuncts. If None is returned, it indicates that no such $\Gamma', \widehat{k}$ could be found. After calling `geninstCont`, passing in Γ_{new} as the context, the `instrument` function then sets Γ_{new} to be the context that was returned, thus ensuring the current context reflects the information about reachable states discovered by `geninstCont`.

At a high level, we can describe the instrumentation analysis as a fixed-point computation on Γ. Suppose we are analyzing the program P. First, we assume that $\mathit{fv}(P) \subseteq \mathrm{Var}$ (we can always establish this by renaming variables). This ensures that the new variables we will be adding (which are in $\underline{\mathrm{IVar}}$) are disjoint from the program variables. Initially we set $\Gamma = \{(l_0, \{\varphi_0\})\} \cup \{(l, \emptyset) \mid l \in \mathit{dom}(P) \wedge l \neq l_0\}$. That is, Γ maps the initial location to φ_0 and all other locations to the empty set. We then repeatedly infer the post-conditions of the continuations in the domain of P, adding these post-conditions to Γ. The function Γ maps each label to the set of reachable states that have been discovered at that label. If this process converges, such that Γ is no longer growing, this indicates that we have fully characterized all the reachable states of the program. We then generate the instrumentation of the program by instrumenting each continuation under each possible precondition. The version of `instrument` given here discards the instrumentations that it generates in the first loop, which computes Γ. In practice, these results are retained to avoid duplicating work. A simple memoization scheme is sufficient to allow reuse of these previously computed instrumentations.

Proof of Correctness We now show that if `geninstCont` satisfies its specification as given in Figure 5.6, then `instrument` also satisfies its specification. That is, we show the following.

$$\begin{aligned}&\text{if } \texttt{instrument}(\varphi_0, P) = \mathsf{Some}\big(\Gamma, \widehat{P}\big)\\ &\text{then } \Gamma \vdash \widehat{P} \blacktriangleright_{\underline{\mathrm{IVar}}} P \text{ and } \varphi_0 \in \Gamma(\mathit{initloc}(P))\end{aligned}$$

Suppose $\texttt{instrument}(\varphi_0, P) = \mathsf{Some}\big(\Gamma, \widehat{P}\big)$. This implies that the first loop has terminated and each `geninstCont` call in the second loop returns $\mathsf{Some}\big(\Gamma_j, \widehat{k}_j\big)$.

That the first loop terminates implies that $\Gamma_{\mathrm{new}} = \Gamma_{\mathrm{old}}$. This implies that every assignment $\Gamma_{\mathrm{new}} := \Gamma$ in the body of the loop left Γ_{new} unchanged. That is, for each φ_l^i such that $\varphi_l^i \in \Gamma_{\mathrm{new}}(l)$ we have that $\texttt{geninstCont}(\Gamma_{\mathrm{new}}, \varphi_l^i, P(l)) = \mathsf{Some}\big(\Gamma_l^i, \widehat{k}_l^i\big)$ implies $\Gamma_l^i = \Gamma_{\mathrm{new}}$. Given the specification of `geninstCont` from Figure 5.6, these Γ_l^i and $\widehat{k}_l^i$ also each satisfy $\Gamma_l^i \vdash \{\varphi_l^i\}\ \widehat{k}_l^i \blacktriangleright_{\underline{\mathrm{IVar}}} P(l)$ which, applying the equalities $\Gamma_l^i = \Gamma_{\mathrm{new}}$, implies $\Gamma_{\mathrm{new}} \vdash \{\varphi_l^i\}\ \widehat{k}_l^i \blacktriangleright_{\underline{\mathrm{IVar}}} P(l)$ for each φ_l^i and $\widehat{k}_l^i$.

Since `geninstCont` is deterministic (in fact, all functions involved in our implementation are deterministic), the calls to `geninstCont` in the second loop will also satisfy these properties. In particular, $\Gamma_{\mathrm{new}} \vdash \{\varphi_l^i\}\ \widehat{k}_l^i \blacktriangleright_{\underline{\mathrm{IVar}}} P(l)$ for all $\varphi_l^i \in \Gamma_{\mathrm{new}}(l)$ implies

$$\Gamma_{\mathrm{new}} \vdash \{\bigvee_i \varphi_l^i\}\ \mathsf{branch}\ \ldots, \mathsf{true} \Rightarrow \widehat{k}_l^i, \ldots\ \mathsf{end} \blacktriangleright_{\underline{\mathrm{IVar}}} P(l) \tag{5.3}$$

by repeated application of the INST-DISJ rule from Figure 4.1.

We will now show that the program $\widehat{P}$ constructed by the second loop satisfies

$$\Gamma \vdash \widehat{P} \blacktriangleright_{\underline{\mathrm{IVar}}} P \text{ and } \varphi_0 \in \Gamma(\mathit{initloc}(P))$$

There is only one rule for showing this, namely the INST-PROG rule in Figure 4.2. Since $\underline{\mathrm{IVar}}$ was defined to be disjoint from the program variables, we have $\underline{\mathrm{IVar}} \cap \mathit{fv}(P) = \emptyset$, which is the first premise of that rule. We have $\mathit{dom}(\widehat{P}) = \mathit{dom}(P)$ from the fact that the second loop defines $\widehat{P}(l)$ for each $l \in \mathit{dom}(P)$. The initial locations are the same in each program, so we have $\mathit{initloc}(\widehat{P}) = \mathit{initloc}(P)$. Finally we must show the following.

$$\forall l \in \mathit{dom}(P).\ (\Gamma_{\mathrm{new}} \vdash \{\Gamma_{\mathrm{new}}(l)\}\ \widehat{P}(l) \blacktriangleright_{\underline{\mathrm{IVar}}} P(l))$$

This follows from (5.3) and the fact that Γ_{new} is interpreted disjunctively, so if $\Gamma_{\mathrm{new}}(l) = \{\varphi_l^1, \ldots, \varphi_l^n\}$ then this corresponds to the formula $\varphi_l^1 \vee \ldots \vee \varphi_l^n$.

The second conjunct of the specification for `instrument` follows from the second conjunct of the specification of `geninstCont`. We have $\varphi_0 \in \Gamma_{\text{new}}(l_0)$ initially. We also have that all calls $\texttt{geninstCont}(\Gamma_{\text{new}}, \varphi, k) = \mathsf{Some}\big(\Gamma', \widehat{k}\big)$ satisfy $\forall l.\ \Gamma'(l) \supseteq \Gamma_{\text{new}}(l)$, which implies that $\varphi_0 \in \Gamma'(l_0)$. From this it follows that $\varphi_0 \in \Gamma_{\text{new}}(l_0)$ for the final value of Γ_{new} computed by `instrument`.

Organization We will now proceed to discuss `geninstCont` and the other functions that the implementation makes use of. These are all mutually recursive and thus difficult to discuss separately. However the guide in Figures 5.6 and 5.7 should be of use in understanding at a high level the role of functions that have yet to be discussed. We will also attempt to informally give the intuition behind functions that are being used, but whose full description is yet to come. As we discuss each function, we prove that it satisfies its specification as given in Figures 5.6 and 5.7.

5.4.2 `geninstCont`

The function call $\texttt{geninstCont}(\Gamma, \varphi, k)$ takes the following arguments.

- Γ A mapping from labels to sets of abstract state formulae that describes the invariants that have already been discovered.
- φ A symbolic state formula that gives the current precondition.
- k The continuation to be instrumented.

`geninstCont` has an optional return value. If it returns $\mathsf{Some}\big(\Gamma', \widehat{k}\big)$, then these must satisfy the following.

$$\big(\Gamma' \vdash \{\varphi\}\ \widehat{k} \blacktriangleright_{\underline{\text{IVar}}} k\big) \wedge \big(\forall l.\ \Gamma'(l) \supseteq \Gamma(l)\big)$$

Recall that $\widehat{k}$ consists of the commands and control structure of k, plus possibly some additional commands over variables in $\underline{\text{IVar}}$.

The code for `geninstCont` is given on page 210. We first check if the precondition is unsatisfiable by calling $\texttt{implies}(\varphi, \mathsf{false}, \ldots)$, which returns $\mathsf{Some}\big(\widehat{k}\big)$ only if false

can be established from the precondition φ (modulo the instrumentation commands, this corresponds to showing $\varphi \Rightarrow \mathsf{false}$). Such inconsistency can occur due to the accumulation of constraints from branch conditions. `implies` also ensures $\widehat{k}$ is an instrumentation command that establishes the precondition false. A formal summary of `implies` is given in Figure 5.7. Since $\Gamma \vdash \{\mathsf{false}\}\ (\mathsf{assert}(\mathsf{false})\,\texttt{;}\ \mathsf{halt}) \blacktriangleright_{\underline{\mathrm{IVar}}} k$ holds for any k by rule FALSE from Figure 4.1, our specification of `implies` ensures that the following holds.

$$\Gamma \vdash \{\varphi\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

This result satisfies the specification for `geninstCont` from Figure 5.6.

If φ is consistent, then the instrumentation depends on the form of the continuation k. We now consider each case in turn, describing the operations performed by `geninstCont` and presenting the soundness argument at the same time (that is, we show in each case that `geninstCont` satisfies its specification as given in Figure 5.6).

CASE $k = (c\,\texttt{;}\,k')$: In the case of a command, where $k = (c\,\texttt{;}\,k')$, we construct the following function, which we will refer to here as $f_{k'}$.

$$f_{k'} \stackrel{\text{def}}{=} \lambda x.\ \texttt{geninstCont}(\Gamma, x, k')$$

Given the specification of `geninstCont` from Figure 5.6, this function has the type $\mathrm{Gen}(k')$. It can thus be passed to `instPost`, which expects such a function as its third argument.

The function call $\texttt{instPost}(\varphi, c, f_{k'})$ computes the post-condition of c with respect to the state φ. It then calls $f_{k'}$ with that post-condition. The reason `instPost` operates this way, instead of simply returning the post-condition, is that it is sometimes necessary to perform case splits before the post-condition of c can be determined. In such situations, the post-condition can be different under each branch of the case split. Passing $f_{k'}$ to `instPost` yields a simple method of obtaining instrumentations of k for each of these cases.

By examining the specifications given in Figure 5.6, we can verify that the code in the $k = (c\,\texttt{;}\,k')$ case is correct. To satisfy the specification for `geninstCont`, this case must

Function `geninstcont`(Γ, φ, k). Generates an instrumented continuation for k starting from precondition φ.

```
if implies($\varphi$, false, (assume(false); halt)) = Some($\widehat{k}$) then
  /* If $\varphi$ is unsatisfiable, return $\widehat{k}$.                    */
  return Some($\Gamma$, $\widehat{k}$)
else
  /* Otherwise, continue instrumenting $k$.                                  */
  match $k$ with
    case ($c$; $k'$)
      return instPost($\varphi, c, \lambda x.$ geninstCont($\Gamma, x, k'$))
    case branch $e_1 \Rightarrow k_1, \ldots, e_n \Rightarrow k_n$ end
      let $[e'_1, \ldots, e'_n]$ = branchAnnot($\varphi, [e_1, \ldots, e_n]$) in
      let Some($\Gamma_1, \widehat{k_1}$) = geninstCont($\Gamma, \varphi \wedge e_1, k_1$) in
          ⋮
      let Some($\Gamma_n, \widehat{k_n}$) = geninstCont($\Gamma, \varphi \wedge e_n, k_n$) in
        return Some($\bigcup_i(\Gamma_i)$, branch $e_1 \Rightarrow$ assume($e'_1$); $\widehat{k_1}, \ldots$
                                                  $e_n \Rightarrow$ assume($e'_n$); $\widehat{k_n}$ end)
      match failed $\Rightarrow$ return None
    case goto $l$
      if $\exists \varphi' \in \Gamma(l)$. implies($\varphi, \varphi'$, goto $l$) = Some($\widehat{k}$) then
        return Some($\Gamma, \widehat{k}$)
      else
        let ($\varphi'$, c) = abstract($\varphi$) in
          return Some($\Gamma[l \to (\Gamma(l) \cup \varphi')]$, (c ⨾ goto $l$))
    case halt
      return Some($\Gamma$, halt)
    case abort
      return Some($\Gamma$, abort)
  end
```

return $\mathsf{Some}\big(\Gamma, \widehat{k}\big)$ such that

$$\Gamma \vdash \{\varphi\}\ \widehat{k} \blacktriangleright_{\underline{\text{IVar}}} (c\,\mathbf{;}\,k')$$

(or return None). Checking the specification for `instPost`, we see that the return value of `instPost`$(\varphi, c, f_{k'})$ satisfies this exactly.

CASE $k = \mathsf{branch}\ \ldots, e_i \Rightarrow k_i, \ldots\ \mathsf{end}$:

For each case i of the branch, we conjoin e_i to the current symbolic state φ and then pass this updated state to a recursive call of `geninstCont`. By the specification of `geninstCont`, this will return either None or $\mathsf{Some}\big(\Gamma_i, \widehat{k_i}\big)$ such that the following holds.

$$\Gamma_i \vdash \{\varphi \wedge e_i\}\ \widehat{k_i} \blacktriangleright_{\underline{\text{IVar}}} k_i$$

We also call `branchAnnot`$(\varphi, [e_1, \ldots, e_n])$. This returns $[e'_1, \ldots, e'_n]$ such that each e'_i is an over-approximation of e_i in the state φ. That is, $\varphi \wedge e_i \Rightarrow e'_i$ for all e_i, e'_i. The idea is that, whereas the e_i are statements over program variables, which may involve variables of address type, the e'_i will be statements over instrumentation variables.

For example, under the symbolic state $ls(\underline{n}; x, \mathsf{nil})$, the branch condition $x = \mathsf{nil}$ might be translated to $\underline{n} = 0$. In this case, the call

$$\texttt{branchAnnot}(ls(\underline{n}; x, \mathsf{nil}), [x = \mathsf{nil}, x \neq \mathsf{nil}])$$

would return

$$[\underline{n} = 0, \underline{n} > 0]$$

The specifications of the recursive `geninstCont` calls and the `branchAnnot` function are sufficient to allow us to show that this case satisfies the specification of `geninstCont`. The implications $\varphi \wedge e_i \Rightarrow e'_i$ allow us to apply the INST-ASSUME rule to conclude

$$\Gamma_i \vdash \{\varphi \wedge e_i\}\ (\mathsf{assume}(e'_i)\,\mathbf{;}\,\widehat{k_i}) \blacktriangleright_{\underline{\text{IVar}}} k_i$$

Let $\Gamma' = \bigcup_i(\Gamma_i)$. Since the sets given by $\Gamma'(l)$ are interpreted disjunctively—that is, $\bigcup_i(\Gamma_i)(l)$ corresponds to the separation logic formula $\bigvee_i(\Gamma_i(l))$—we have that for all l, i

the implication $\Gamma_i(l) \Rightarrow \Gamma'(l)$ holds. Thus we can apply Lemma 12 to obtain

$$\Gamma' \vdash \{\varphi \wedge e_i\}\ (\mathsf{assume}(e_i');\widehat{k_i}) \blacktriangleright_{\underline{\text{IVar}}} k_i$$

for all e_i, k_i. This then allows us to apply the BRANCH rule to obtain

$$\Gamma' \vdash \{\varphi\}\ \mathsf{branch}\ \ldots, e_i \Rightarrow \mathsf{assume}(e_i');\widehat{k_i}, \ldots\ \mathsf{end} \blacktriangleright_{\underline{\text{IVar}}} k$$

Thus the value returned satisfies the specification for `geninstCont`.

CASE $k = \mathsf{goto}\ l$:

In the goto case, there are two approaches, depending on what can be shown of the current state φ.

"then" branch If there is some φ' in Γ associated with the same label we are jumping to such that $\varphi \Rightarrow \varphi'$, then we can apply the GOTO rule followed by the STRENGTHENING rule as follows.

We first note that if $\varphi' \in \Gamma$ then we have the following by the GOTO rule from Figure 4.1.

$$\Gamma \vdash \{\varphi'\}\ \mathsf{goto}\ l \blacktriangleright_{\underline{\text{IVar}}} \mathsf{goto}\ l$$

Examining the specification for the call to `implies`$(\varphi, \varphi', \mathsf{goto}\ l)$, we see that if the result is $\mathsf{Some}(\widehat{k})$ then this ensures that the following holds.

$$\Gamma \vdash \{\varphi\}\ \widehat{k} \blacktriangleright_{\underline{\text{IVar}}} \mathsf{goto}\ l$$

Thus returning $\mathsf{Some}(\widehat{k})$ allows this case to satisfy the specification for `geninstCont`.

In essence, the goal of `implies`$(\varphi, \varphi', \widehat{k}')$ is to generate an instrumentation that connects φ to φ'. This instrumentation may involve applications of INST-ASSIGN, which will prepend commands to $\widehat{k}'$. It may also make use of STRENGTHENING and case-splitting rules such as our INST-BRANCH derived rule from Section 4.1.3.

As a simple example, consider the call `implies`$(\mathit{ls}(\underline{n}-1; x, \mathsf{nil}), \mathit{ls}(\underline{n}; x, \mathsf{nil}), \mathsf{goto}\ l)$, where Γ maps l to $\{\mathit{ls}(\underline{n}; x, \mathsf{nil})\}$. This would return the instrumented continuation

($\underline{n}$:= $\underline{n} - 1$; goto l), where the addition of the command $\underline{n}$:= $\underline{n} - 1$ ensures that if $ls(\underline{n} - 1; x, \mathsf{nil})$ is the precondition, then $ls(\underline{n}; x, \mathsf{nil})$ will hold just prior to the goto l statement.

"else" branch If we instead end up executing the "else" branch in the goto l case, then we call `abstract`(φ). The goal of `abstract` is to weaken symbolic state formulae so that they cover more states. These more abstract states are then more likely to be loop invariants.

For example, during execution of a program that creates a linked list, we might encounter a symbolic state such as the one below.

$$\varphi_1 \overset{\text{def}}{=} \exists z.\ (x \mapsto [\mathsf{next} : z]) * (z \mapsto [\mathsf{next} : \mathsf{nil}])$$

This formula implies the formula below, which would be a valid loop invariant for a list creation routine.

$$ls(\underline{n}; x, \mathsf{nil})$$

In order to establish this formula, we need to initialize $\underline{n}$. This is the role of the second component of the return value of `abstract`. The initialization command for this example is $\underline{n} = 2$ and so `abstract`(φ_1) would return $(ls(\underline{n}; x, \mathsf{nil}), [\underline{n} = 2])$.

The formal specification of `abstract` given in Figure 5.7 ensures that if `abstract`(φ) returns $(\varphi', \mathbf{c})$ then for all $\Gamma, k, \widehat{k}'$ we have that $\Gamma \vdash \{\varphi'\}\ \widehat{k}' \blacktriangleright_{\underline{\text{IVar}}} k$ implies $\Gamma \vdash \{\varphi\}\ (\mathbf{c} \mathbin{⨾} \widehat{k}') \blacktriangleright_{\underline{\text{IVar}}} k$. Let $\Gamma' = \Gamma[l \to (\Gamma(l) \cup \{\varphi'\})]$. Clearly $\forall l.\ \Gamma'(l) \supseteq \Gamma(l)$. We have that $\Gamma' \vdash \{\varphi'\}$ goto $l \blacktriangleright_{\underline{\text{IVar}}}$ goto l. The specification of `abstract` then tells us that $\Gamma' \vdash \{\varphi\}\ \mathbf{c} \mathbin{⨾}$ goto $l \blacktriangleright_{\underline{\text{IVar}}}$ goto l holds. Since we return $\mathsf{Some}\big(\Gamma', (\mathbf{c} \mathbin{⨾} \text{goto } l)\big)$, this establishes the specification of `geninstCont` in this case of the match.

CASE halt, abort: In the case of halt or abort, no instrumentation commands are added. The fact that the return values in these cases satisfy the specification for `geninstCont` follows directly from the rules HALT and ABORT in Figure 4.1.

Second Conjunct

We now show that the second conjunct in the specification of `geninstCont` holds. We must show that if $\texttt{geninstCont}(\Gamma, \varphi, k) = \mathsf{Some}\big(\Gamma', \widehat{k}\big)$ then

$$\forall l.\ \Gamma'(l) \supseteq \Gamma(l)$$

In the branch case, we have $\forall l.\ \Gamma_i(l) \supseteq \Gamma(l)$ by the inductive hypothesis. We then have $\bigcup_i(\Gamma_i)$ by the definition of $\cup$ on contexts. The halt, and abort cases are immediate, as $\forall l.\ \Gamma(l) \supseteq \Gamma(l)$ trivially holds. This leaves the $(c\texttt{;}\,k)$ case and the $\mathsf{goto}\ l$ case.

For $(c\texttt{;}\,k)$ we need to examine the definition of `instPost`. This is defined in the next section and we will discuss it in more detail there. For now, it suffices to note that the context `instPost` returns is the same context produced by the function passed as the third argument—in this case, a recursive call to `geninstCont`. This lets us apply the inductive hypothesis, from which this case then immediately follows.

For $\mathsf{goto}\ l$, the "then" branch is immediate as the input context is returned unchanged. The "else" branch returns $\Gamma[l \to (\Gamma(l) \cup \varphi')]$. Since $\Gamma(l) \cup \varphi' \supseteq \Gamma(l)$ we have our result.

5.4.3 `instPost`

The function `instPost`, which is responsible for instrumenting commands, is given on page 215. A call $\texttt{instPost}(\varphi, c, f_k)$ takes the following arguments.

φ	A symbolic state formula that gives the precondition.
c	The command whose post-condition should be taken.
f_k	The instrumentation generator to apply to the post-condition when it is obtained.

`instPost` has an optional return value. If it returns $\mathsf{Some}\big(\Gamma, \widehat{k}\big)$, then these must satisfy the following.

$$\Gamma \vdash \{\varphi\}\ \widehat{k} \blacktriangleright_{\underline{\text{IVar}}} (c\texttt{;}\,k)$$

We write $A[x]$ to denote the commands that access the cell at x.

$$A[x] \quad ::= \quad y := x.f \mid \mathsf{free}\ x \mid x.f = e$$

These commands require a heap cell to exist at x in order to ensure that execution does not result in a memory fault.

Function `instpost` (φ, c, f_k). Takes the post-condition of φ with respect to the command c and applies f_k to the result, returning an instrumentation of c**;** k.

```
fun doPost($\varphi, c, f_k$) =
  match partialPost($\varphi, c$) with
    case Some($\varphi'$)
      if $f_k(\varphi') = \mathsf{Some}\big(\Gamma, \widehat{k}\big)$ then
        return $\mathsf{Some}\big(\Gamma, (c;\widehat{k})\big)$
      else
        return None
    case None
      return None
  end
in
  match $c$ with
    case $A[x]$
      return exposeCellThenInst($\varphi, x, \lambda\varphi.$ doPost($\varphi, c, f_k$))
    otherwise
      return doPost($\varphi, c, f_k$)
  end
```

The function `instPost` makes use of two helper functions: `partialPost` and `exposeCellThenInst`. The `partialPost` function returns the post-condition of a command with respect to some precondition, but is not able to perform the theorem proving that is sometimes necessary to show that the heap contains a cell at a given address. The `exposeCellThenInst` fills in this shortcoming by making calls into a theorem prover for symbolic state formulae.

Helper Function: `partialPost`

The code for `partialPost` is given on page 217. This function implements a partial post-condition operator. It takes the following arguments.

- φ A symbolic state formula that gives the current precondition.
- c The command for which the postcondition should be computed.

It returns either None or $\mathsf{Some}\big(\varphi'\big)$. If $\mathsf{Some}\big(\varphi'\big)$ is returned, then this formula satisfies the following.

$$\{\varphi\}\ c\ \{\varphi'\}$$

For assignment, the standard strongest post-condition rule is used. For allocation, we use the standard post-condition rule from separation logic Reynolds [2002]. For non-deterministic assignment we existentially quantify what is now the previous value of x. For skip we leave the precondition unchanged.

The rules for the heap-manipulating commands first check that the precondition syntactically contains a points-to predicate specifying the contents of the heap cell being accessed. For example, in the case for $x_1 := x_2.f$, the expression

$$\textbf{let}\ (\exists \vec{z}.\ (\Sigma * (x_2 \mapsto [f : e, \rho])) \wedge \Pi) = \varphi\ \textbf{with}\ x_1, x_2 \notin \vec{z}\ \textbf{in}$$

matches φ against the pattern $\exists \vec{z}.\ (\Sigma * (x_2 \mapsto [f : e, \rho])) \wedge \Pi$. The match succeeds if φ can be shown to have the given form using only the equivalence defined in Figure 5.2. If the match succeeds, then $\vec{z}, \Sigma, e, \rho$, and Π are bound to the sub-formulae at these positions in φ. Additionally, the condition $x_1, x_2 \notin \vec{z}$ is enforced, which may require alpha-varying φ prior to performing the matching.

Once this syntactic match has been performed, the precondition is updated to reflect the effect of executing the command. Heap-manipulating commands such as $x_1 := x_2.f$ are only safe in states containing a heap cell at a given address (in this case a heap cell at x_2 with field f). If the required heap cell does not appear in the formula explicitly as a points-to predicate (that is, if the syntactic match fails), then the function returns None. Otherwise it returns $\mathsf{Some}\big(\varphi'\big)$ where φ' is the post-condition.

Function `partialPost`(φ, c). Returns the post-condition for command c given precondition φ. All primed variables are chosen to be fresh. Side conditions are satisfied by alpha-varying φ (the match fails if this is not possible).

```
match $c$ with
  case $x := e$
    return $\mathsf{Some}\big(\exists x'.\ (\varphi[x'/x] \wedge x = e[x'/x])\big)$
  case $x := \mathsf{alloc}(f_1, \ldots, f_n)$
    return $\mathsf{Some}\big(\exists x', y'_1, \ldots, y'_n.\ (\varphi[x'/x] * (x \mapsto [f_1 : y'_1, \ldots, f_n : y'_n]))\big)$
  case $x := ?$
    return $\mathsf{Some}\big(\exists x.\ \varphi\big)$
  case $\mathsf{skip}$
    return $\mathsf{Some}\big(\varphi\big)$
  case $x_1 := x_2.f$
    let $(\exists \vec{z}.\ (\Sigma * (x_2 \mapsto [f : e, \rho])) \wedge \Pi) = \varphi$ with $x_1, x_2 \notin \vec{z}$ in
    let $e' = e[x'_1/x_1]$ in
    let $\rho' = \rho[x'_1/x_1]$ in
    let $\Sigma' = \Sigma[x'_1/x_1]$ in
    let $\Pi' = \Pi[x'_1/x_1]$ in
      return $\mathsf{Some}\big(\exists x'_1, \vec{z}.\ (\Sigma' * (x_2[x'_1/x_1] \mapsto [f : e', \rho'])) \wedge (\Pi' \wedge x_1 = e')\big)$
    match failed $\Rightarrow$ return $\mathsf{None}$
  case $x.f := e$
    let $(\exists \vec{z}.\ (\Sigma * (x \mapsto [f : e_1, \rho])) \wedge \Pi) = \varphi$ with $fv(x, e) \cap \vec{z} = \emptyset$ in
      return $\mathsf{Some}\big(\exists \vec{z}.\ (\Sigma * (x \mapsto [f : e, \rho])) \wedge \Pi\big)$
    match failed $\Rightarrow$ return $\mathsf{None}$
  case $\mathsf{free}\ x$
    let $(\exists \vec{z}.\ (\Sigma * (x \mapsto [\rho])) \wedge \Pi) = \varphi$ with $x \notin \vec{z}$ in
      return $\mathsf{Some}\big(\exists \vec{z}.\ \Sigma \wedge \Pi\big)$
    match failed $\Rightarrow$ return $\mathsf{None}$
end
```

Helper Function: `exposeCellThenInst`

In order to produce a result for a command that accesses a heap cell at x, the code discussed above for `partialPost` requires the precondition to contain a term that syntactically matches $(x \mapsto [\rho]) * \varphi$ for some ρ and φ. This causes the code to return None in some cases where a post-condition does exist. An example of such a case is the formula $\mathit{ls}(\underline{n}; x, \mathsf{nil}) \wedge \underline{n} > 0$, which implies that the list at x is non-empty and thus x is a valid pointer into the heap. However, discovering this fact requires reasoning about separation logic implications.

We will talk about separation logic reasoning in Section 5.5. In the meantime, we will give a high-level description of `exposeCellThenInst`, which is the function that makes the appropriate call into our theorem proving system to show that a heap cell at some address x exists. The call `exposeCellThenInst`(φ, x, f_k) takes the following arguments.

φ	A symbolic state formula that gives the current precondition.
x	The address of the heap cell to be revealed.
f_k	The instrumentation generator to apply to the formula that results from showing that x is in the heap.

If `exposeCellThenInst`(φ, x, f_k) returns $\mathsf{Some}\big(\Gamma, \widehat{k}\big)$ then these must satisfy

$$\Gamma \vdash \{\varphi\}\ \widehat{k} \blacktriangleright_{\underline{\text{IVar}}} k$$

As with the `implies` function, informally described on page 212, the instrumentation commands added to the result of f_k in order to obtain $\widehat{k}$ may consist of assignments or branches. To take a branching example, consider the following symbolic state formula.

$$\varphi_0 \overset{\text{def}}{=} (\mathit{ls}(\underline{a}; x, y) * \mathit{ls}(\underline{b}, y, x)) \wedge \underline{a} + \underline{b} > 0$$

This states that there is a non-empty cyclic singly-linked list with x and y pointing into it. The pointers x and y divide the cycle into two segments: one starting at x and ending at y and the other running from y back to x. The condition $\underline{a} + \underline{b} > 0$ implies that there is at least one heap cell in the cyclic list. This implies that at least one of the segments is non-empty, but it does not specify which. If we want to expose the heap cell at x, we must first case split on whether the list segment starting at x is empty. We obtain the following if the segment starting at x is non-empty (and thus $\underline{a} > 0$)

$$\varphi_1 \overset{\text{def}}{=} (\exists z.\ x \mapsto [\mathsf{next} : z] * \mathit{ls}(\underline{a} - 1; z, y) * \mathit{ls}(\underline{b}, y, x)) \wedge \underline{a} > 0$$

and the following if that segment is empty (and thus $\mathsf{a} = 0$)

$$\varphi_2 \overset{\text{def}}{=} (\exists z.\ x \mapsto [\mathsf{next} : z] * \mathit{ls}(\underline{b} - 1; z, x)) \wedge x = y \wedge \underline{a} = 0 \wedge \underline{b} > 0$$

If $f_k(\varphi_1) = \mathsf{Some}\big(\Gamma_1, \widehat{k}_1\big)$ and $f_k(\varphi_2) = \mathsf{Some}\big(\Gamma_2, \widehat{k}_2\big)$ then the call

$$\texttt{exposeCellThenInst}(\varphi_0, x, f_k)$$

would return

$$\mathsf{Some}\Big(\Gamma_1 \cup \Gamma_2, \mathsf{branch}\ \underline{a} > 0 \Rightarrow \widehat{k}_1, \underline{a} = 0 \Rightarrow \widehat{k}_2\ \mathsf{end}\Big)$$

Correctness

We now show that `instPost` satisfies its specification. We first consider the case where c does not match $A[x]$. In this case, `instPost` calls $\texttt{doPost}(\varphi, c, f_k)$ which calls $\texttt{partialPost}(\varphi, c)$. Suppose $\texttt{partialPost}(\varphi, c)$ returns $\mathsf{Some}\big(\varphi'\big)$. Then by its specification in Figure 5.6 we have

$$\{\varphi\}\ c\ \{\varphi'\} \tag{5.4}$$

Since f_k has type $\mathrm{Gen}(k)$ we have that if $f(\varphi')$ returns $\mathsf{Some}\big(\Gamma, \widehat{k}\big)$ then the following holds.

$$\Gamma \vdash \{\varphi'\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k \tag{5.5}$$

We can then apply the COMMAND rule from Figure 4.1 to (5.4) and (5.5) to obtain

$$\Gamma \vdash \{\varphi\}\ (c\,\texttt{;}\,\widehat{k}) \blacktriangleright_{\underline{\mathrm{IVar}}} (c\,\texttt{;}\,k)$$

which establishes that our return value satisfies the specification for `instPost`.

For the $c = A[x]$ case, we first note that one consequence of the argument above about `doPost` is that the function

$$\lambda\varphi.\ \texttt{doPost}(\varphi, c, f_k)$$

has type $\mathrm{Gen}(c\,\boldsymbol{;}\,k)$. This allows it to be passed to `exposeCellThenInst`. The specification of `exposeCellThenInst` then tells us that if this call returns $\mathsf{Some}\big(\Gamma, \widehat{k}\big)$ then we have

$$\Gamma \vdash \{\varphi\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} (c\,\boldsymbol{;}\,k)$$

which satisfies the specification for `exposeCellThenInst`.

5.5 Theorem Proving

We now describe our proof system for symbolic state formulae.[2] This forms the basis of many of the remaining functions. Specifically, the functions `exposeCellThenInst`, `implies`, and `branchAnnot` all make use of the theorem prover. Each of these functions answers slightly different problems, and so we will actually describe three different proof systems. However, the vast majority of the proof rules are shared by all three systems. We will thus start with the simplest problem, *entailment*, which is used by the `implies` function, and then describe our solution for the more complex problems of *frame inference* and *pure abduction*, by focusing on the differences between the proof systems for these problems and the proof system for entailment. The discussion of pure abduction will be delayed until Section 5.10, as this constitutes an optional portion of the algorithm. Instrumentations for programs can be produced without having a proof system for pure abduction, but including this system enables us to generate more precise instrumentations.

[2] As symbolic state formulae correspond to separation logic formulae of a restricted form, this can also be viewed as a proof system for separation logic formulae of this form.

5.5.1 Entailment

Our system for entailment targets the same problem as Berdine et al. [2004] and Nguyen and Chin [2008], although our system is unique in that it generates instrumentation commands during proof search. This addition is necessary if the prover is to be used in a system for producing instrumented programs, such as the one we are considering in this chapter.

We start with an example showing when entailment is useful. Suppose we have reached symbolic state

$$\varphi \overset{\mathrm{def}}{=} \mathit{ls}(\underline{n} + 1; x, \mathsf{nil})$$

and have previously discovered that the symbolic state

$$\varphi' \overset{\mathrm{def}}{=} \mathit{ls}(\underline{n}; x, \mathsf{nil})$$

is reachable at the same location. In this case, we would like to notice that we can reach φ' from φ by executing the instrumentation command $\underline{n} := \underline{n} + 1$. If we can show this, then we may stop exploring this branch. If we fail to notice such situations, this can lead to non-termination of the algorithm. This is the sort of query performed by the `implies` function and supported by our proof system for entailment.

Formally, we will define the following judgment.

$$\varphi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k}'} \varphi' \mathbin{/\!\!/} \widehat{k}$$

In the above, φ, φ', S, and $\widehat{k}'$ are considered inputs and $\widehat{k}$ is the output. Recall that S is a set of inductive predicate specifications as described in Section 5.2.

The proof system will be designed such that if the judgment $\varphi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k}'} \varphi' \mathbin{/\!\!/} \widehat{k}$ holds and $\Gamma \vdash \{\varphi'\}\ \widehat{k}' \blacktriangleright_{\underline{\mathrm{IVar}}} k$ for some Γ, k, then

$$\Gamma \vdash \{\varphi\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

To establish this, the entailment system can be viewed as transforming a proof of $\Gamma \vdash \{\varphi'\}\ \widehat{k}' \blacktriangleright_{\underline{\mathrm{IVar}}} k$ into a proof of $\Gamma \vdash \{\varphi\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$ by using the instrumentation rules in Figure 4.1 to fill in the gaps between φ and φ'. And in fact, we will establish soundness of the proof system by showing that each rule presented can be justified in terms of rules from Figure 4.1.

As an example, if φ is

$$\mathit{ls}(\underline{n}_1 + 1; y, x) * \mathit{ls}(\underline{n}_2; x, \mathsf{nil}) \wedge x \neq \mathsf{nil}$$

and φ' is

$$\exists z, v.\ \mathit{ls}(\underline{n}_1; y, x) * x \mapsto [\mathsf{next} : z, \mathsf{data} : v] * \mathit{ls}(\underline{n}_2; z, \mathsf{nil})$$

then the system may reason that φ' can be reached from φ by inserting the instrumentation command $\underline{n}_1 := \underline{n}_1 + 1$. The post-condition of this command is

$$\mathit{ls}(\underline{n}_1; y, x) * \mathit{ls}(\underline{n}_2; x, \mathsf{nil}) \wedge x \neq \mathsf{nil}$$

from which φ' follows by pure separation logic reasoning.

Bookkeeping

At a high level, proving proceeds by matching spatial predicates to the left of $\underset{\mathbf{s}}{\Longrightarrow}$ with spatial predicates on the right. This matching procedure is essentially an application of the following inference rule (the*frame rule*), which is admissible in separation logic.

$$\frac{Q_1 \Rightarrow Q_2}{Q_1 * R \Rightarrow Q_2 * R}$$

To give an analogous example in our syntax, if the following holds

$$\varphi \underset{\mathbf{s}}{\Longrightarrow}_{\widehat{k'}} \varphi' \mathbin{/\!\!/} \widehat{k}$$

then the statement below does as well (provided x and y are program variables and not instrumentation variables).

$$\varphi * x \mapsto [\mathsf{data} : y] \underset{\mathbf{s}}{\Longrightarrow}_{\widehat{k'}} \varphi' * x \mapsto [\mathsf{data} : y] \mathbin{/\!\!/} \widehat{k}$$

We then view proof search as proceeding from the bottom up. If we are ever faced with a goal matching that given above, we can note that $x \mapsto [\mathtt{data} : y]$ occurs on both sides, discard it, and proceed to search for a proof of $\varphi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k}'} \varphi' \mathbin{/\!\!/} \widehat{k}$.

This relatively simple matching process becomes somewhat complicated in the presence of instrumentation commands, pure formulae, and quantifiers, so the actual proof search is performed over an expanded form of the judgment, which includes some bookkeeping information.

The rules for the proof system are given in Figures 5.8 and 5.9 and involve judgments of the following form.

$$\Sigma_a \mathbin{[\!]} \varphi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k}'} \varphi' \mathbin{/\!\!/} \widehat{k}$$

The $\Gamma, \varphi, \varphi', \mathbf{S}$, and $\widehat{k}'$ components are the same as before. The Σ_a component exists to aid in the matching process. As spatial predicates in φ are matched with predicates in φ', the matched predicate is moved to Σ_a.

Formally, if the sequent

$$\Sigma_a \mathbin{[\!]} \varphi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k}'} \varphi' \mathbin{/\!\!/} \widehat{k}$$

is derivable, then the following holds

$$\Gamma \vdash \{\Sigma_a * \varphi'\}\ \widehat{k}' \blacktriangleright_{\underline{\mathrm{IVar}}} k \text{ implies } \Gamma \vdash \{\Sigma_a * \varphi\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

The following components are *inputs* in a bottom-up proof search using these rules.

$$\mathbf{S}, \widehat{k}', \Sigma_a, \varphi, \varphi'$$

The only *output* is $\widehat{k}$.

Our earlier notation $\varphi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k}'} \varphi' \mathbin{/\!\!/} \widehat{k}$ should be viewed as an abbreviation for the following.

$$\mathbf{emp} \mathbin{[\!]} \varphi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k}'} \varphi' \mathbin{/\!\!/} \widehat{k}$$

Notation

One common operation in the rules in Figures 5.8 and 5.9 is to check whether a spatial formula is present in a symbolic state formula. We define the following notation to indicate this check (where $\equiv$ denotes the equality relation given in Figure 5.2).

$$\Sigma' \in \varphi \;\overset{\text{def}}{=}\; \big(\varphi \equiv \exists\vec{x}.\, \Sigma \wedge \Pi\big) \text{ and } \Sigma = \Sigma' * \Sigma_1 \text{ for some } \Sigma_1 \text{ and } \mathit{fv}(\Sigma') \cap \vec{x} = \emptyset$$

This implies that φ is logically equivalent to $\varphi' * \Sigma'$, where $\varphi' = \exists\vec{x}.\, \Sigma_1 \wedge \Pi$ (using the variable names in the definition above).

An example usage of this notation occurs in rule NotNull in Figure 5.8, where we have $(e \mapsto \rho) \in (\Sigma_a * \varphi)$ as one of the premises. Recall that $\Sigma_a * \varphi$ denotes the symbolic state formula φ' that is semantically equivalent to $\Sigma_a * \varphi$ (for more details, see Section 5.1). The result is that the statement $(e \mapsto \rho) \in (\Sigma_a * \varphi)$ is true when $e \mapsto \rho$ is present in either Σ_a or φ, with quantified variables in Σ_a and φ handled appropriately (though, as can be seen by examining the other rules, Σ_a will never contain quantifiers).

As another example, consider the statement $((e_1 \mapsto \rho_1) * (e_2 \mapsto \rho_2)) \in (\Sigma_a * \varphi)$, as present in the Disjoint rule. This is true if $e_1 \mapsto \rho_1$ and $e_2 \mapsto \rho_2$ both occur in Σ_a, or both occur in φ, or if one occurs in Σ_a and one occurs in φ. Thus, this notation gives us a concise way of writing statements regarding the presence of spatial formulae which would otherwise involve a great deal of disjunction.

Rule Explanation and Soundness

We now go through each rule in turn, explaining its effect and presenting its soundness proof. Soundness is shown via induction on the structure of the derivation. Intuitively, we want a derivation of $\Sigma_a \,[\!]\, \varphi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k}'} \varphi' \,/\!/\, \widehat{k}$ to ensure that we can reach φ' from φ. That is, via repeated application of the instrumentation rules from Figure 4.1, we can construct some continuation prefix that reaches the state φ' along all of its branches. Formally, we have the statement below.

Theorem 28. *If $\Sigma_a \,[\!]\, \varphi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k}'} \varphi' \,/\!/\, \widehat{k}$ is derivable then for all Γ, k*

$$\Gamma \vdash \{\Sigma_a * \varphi'\}\; \widehat{k}' \blacktriangleright_{\underline{\mathrm{IVar}}} k \;\; \textit{implies} \;\; \Gamma \vdash \{\Sigma_a * \varphi\}\; \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

Stated in terms of our abbreviated form of judgment, this becomes the following.

$$\frac{\Sigma_a \;[\!]\; \varphi[e/x] \wedge x = e \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}} \varphi' \;/\!\!/\; \widehat{k}}{\Sigma_a \;[\!]\; \varphi \wedge x = e \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}} \varphi' \;/\!\!/\; \widehat{k}}\;\textsc{PropEqL}$$

$$\frac{(e \mapsto \rho) \in (\Sigma_a * \varphi) \qquad \Sigma_a \;[\!]\; \varphi \wedge (e \neq \mathsf{nil}) \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}} \varphi' \;/\!\!/\; \widehat{k}}{\Sigma_a \;[\!]\; \varphi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}} \varphi' \;/\!\!/\; \widehat{k}}\;\textsc{NotNull}$$

$$\frac{((e_1 \mapsto \rho_1) * (e_2 \mapsto \rho_2)) \in (\Sigma_a * \varphi) \qquad \Sigma_a \;[\!]\; \varphi \wedge (e_1 \neq e_2) \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}} \varphi' \;/\!\!/\; \widehat{k}}{\Sigma_a \;[\!]\; \varphi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}} \varphi' \;/\!\!/\; \widehat{k}}\;\textsc{Disjoint}$$

$$\frac{\Pi \Rightarrow \exists\vec{x}.\, \Pi' \text{ is valid}}{\Sigma_a \;[\!]\; \mathbf{emp} \wedge \Pi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}} \exists\vec{x}.\, \mathbf{emp} \wedge \Pi' \;/\!\!/\; \widehat{k'}}\;\textsc{RightPure} \qquad \frac{\Pi \Rightarrow \mathsf{false} \text{ is valid}}{\Sigma_a \;[\!]\; \Sigma \wedge \Pi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}} \varphi' \;/\!\!/\; \mathsf{assume(false);\ halt}}\;\textsc{LeftPureFalse}$$

$$\frac{\Sigma_a * (e \mapsto \rho) \;[\!]\; \varphi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}} \varphi' \;/\!\!/\; \widehat{k}}{\Sigma_a \;[\!]\; (e \mapsto \rho) * \varphi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}} \varphi' * (e \mapsto \rho) \;/\!\!/\; \widehat{k}}\;\textsc{PtoMatches} \qquad \frac{\Sigma_a * d(\vec{e}) \;[\!]\; \varphi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}} \varphi' \;/\!\!/\; \widehat{k}}{\Sigma_a \;[\!]\; d(\vec{e}) * \varphi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}} \varphi' * d(\vec{e}) \;/\!\!/\; \widehat{k}}\;\textsc{PredMatches}$$

Figure 5.8: Proof system for entailment. Basic rules.

Corollary 5. *If* $\varphi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}} \varphi' \;/\!\!/\; \widehat{k}$ *then for all* Γ, k

$$\Gamma \vdash \{\varphi'\}\; \widehat{k'} \blacktriangleright_{\underline{\mathrm{IVar}}} k \textit{ implies } \Gamma \vdash \{\varphi\}\; \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

Proof. The proof is by induction on the structure of the derivation of $\Sigma_a \;[\!]\; \varphi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}} \varphi' \;/\!\!/\; \widehat{k}$. We consider each case below.

PropEqL This rule propagates equalities throughout the formula on the left. Applying our inductive hypothesis yields

$$\Gamma \vdash \{\Sigma_a * \varphi'\}\; \widehat{k'} \blacktriangleright_{\underline{\mathrm{IVar}}} k \;\text{ implies }\; \Gamma \vdash \{\Sigma_a * (\varphi[e/x] \wedge x = e)\}\; \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k \tag{5.6}$$

DEFL

$$\frac{\begin{array}{c}\big(d(\vec{v})\ \texttt{<=>}\ \ldots \mid C_i(\vec{v}) \mid \ldots\big) \in \mathbf{S}\\ C_i(\vec{e}) = \big(\Pi_i : \text{let } \underline{\vec{z}_i} \text{ satisfy } \Pi_i' \text{ in } \varphi_i\big)\\ \forall i.\ \big(\Sigma_a \mathbin{[\!]} (\varphi * \varphi_i) \wedge \Pi_i \wedge \Pi_i' \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}} \varphi' \mathbin{/\!/} \widehat{k}_i\big)\end{array}}{\begin{array}{l}\Sigma_a \mathbin{[\!]} \varphi * d(\vec{e}) \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}} \varphi' \mathbin{/\!/}\\ \mathsf{branch}\ \ldots, \Pi_i \Rightarrow \underline{\vec{z}_i} := \texttt{?;}\,\mathsf{assume}(\Pi_i')\texttt{;}\,\widehat{k}_i, \ldots\ \mathsf{end}\end{array}}\ \forall i.\ \vec{z}_i \notin fv(\varphi, \Sigma_a, \Pi_i)$$

INSTL

$$\frac{\Sigma_a \mathbin{[\!]} \varphi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}} \varphi' \mathbin{/\!/} \widehat{k}}{\Sigma_a \mathbin{[\!]} \varphi[e/\underline{x}] \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}} \varphi' \mathbin{/\!/} (\underline{x} := e\texttt{;}\,\widehat{k})}\ \underline{x} \notin fv(\Sigma_a), fv(x, e) \cap V = \emptyset$$

EXISTSR

$$\frac{\Sigma_a \mathbin{[\!]} \varphi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}} \varphi'[e/x] \mathbin{/\!/} \widehat{k}}{\Sigma_a \mathbin{[\!]} \varphi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}} \exists x.\ \varphi' \mathbin{/\!/} \widehat{k}}$$

EXISTSL

$$\frac{\Sigma_a \mathbin{[\!]} \varphi[\underline{c}/x] \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}} \varphi' \mathbin{/\!/} \widehat{k}}{\Sigma_a \mathbin{[\!]} \exists x.\ \varphi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}} \varphi' \mathbin{/\!/} \underline{c} := \texttt{?;}\,\widehat{k}}\ \underline{c}\text{ fresh}$$

Figure 5.9: Proof system for entailment. Rules for inductively specified predicates and variables. We write $\underline{\vec{z}} := ?$ to indicate the sequence of commands $\underline{z}_1 := \texttt{?;}\ \ldots\texttt{;}\,\underline{z}_n := ?$.

We must show

$$\Gamma \vdash \{\Sigma_a * \varphi'\}\ \widehat{k'} \blacktriangleright_{\underline{\text{IVar}}} k \ \text{ implies }\ \Gamma \vdash \{\Sigma_a * (\varphi \wedge x = e)\}\ \widehat{k} \blacktriangleright_{\underline{\text{IVar}}} k$$

We first assume $\Gamma \vdash \{\Sigma_a * \varphi'\}\ \widehat{k'} \blacktriangleright_{\underline{\text{IVar}}} k$ and apply (5.6) to derive

$$\Gamma \vdash \{\Sigma_a * (\varphi[e/x] \wedge x = e)\}\ \widehat{k} \blacktriangleright_{\underline{\text{IVar}}} k$$

We can then apply the STRENGTHENING rule from Figure 4.1 to the formula above using the following implication.

$$\Big(\Sigma_a * (\varphi \wedge x = e)\Big) \Rightarrow \Big(\Sigma_a * (\varphi[e/x] \wedge x = e)\Big)$$

This yields

$$\Gamma \vdash \{\Sigma_a * (\varphi \wedge x = e)\}\ \widehat{k} \blacktriangleright_{\underline{\text{IVar}}} k$$

which completes the proof.

Note that the antecedent of the goal matched the antecedent of the implication we got from the inductive hypothesis (5.6). This will be the case for all rules, so we will henceforth focus on showing that the conclusion of the implication from the inductive hypothesis implies the conclusion of our goal.

NOTNULL This rule adds $e \neq \mathsf{nil}$ to our assumptions in cases where a cell at location e has been shown to be present in the heap. For soundness, we have

$$\Gamma \vdash \{\Sigma_a * (\varphi \wedge e \neq \mathsf{nil})\}\ \widehat{k} \blacktriangleright_{\underline{\text{IVar}}} k$$

and

$$(e \mapsto \rho) \in (\Sigma_a * \varphi)$$

which, by our definition of this notation (see page 223) gives us

$$(\Sigma_a * \varphi) = (e \mapsto \rho) * \varphi_1$$

for some φ_1. Note that this implies

$$\Sigma_a * \varphi \Rightarrow (e \neq \mathsf{nil})$$

We must show

$$\Gamma \vdash \{\Sigma_a * \varphi\}\ \widehat{k} \blacktriangleright_{\underline{\text{IVar}}} k$$

This follows from STRENGTHENING and the implication above.

DISJOINT This rule is similar to the one above, except that it uses the fact that both $e_1 \mapsto \rho_1$ and $e_2 \mapsto \rho_2$ are present on the left to infer $e_1 \neq e_2$. We have

$$\Gamma \vdash \{\Sigma_a * (\varphi \wedge e_1 \neq e_2)\}\ \widehat{k} \blacktriangleright_{\underline{\text{IVar}}} k$$

and

$$((e_1 \mapsto \rho_1) * (e_2 \mapsto \rho_2)) \in (\Sigma_a * \varphi)$$

This second fact implies

$$(\Sigma_a * \varphi) = ((e_1 \mapsto \rho_1) * (e_2 \mapsto \rho_2)) * \varphi_1$$

for some φ_1, which implies

$$(\Sigma_a * \varphi) \Rightarrow e_1 \neq e_2$$

We need to show

$$\Gamma \vdash \{\Sigma_a * \varphi\}\ \widehat{k} \blacktriangleright_{\underline{\text{IVar}}} k$$

which follows from STRENGTHENING and the implication above.

RIGHTPURE This is one of the axioms of the proof system. It is triggered when the right-hand side becomes empty—that is, the component to the right of the $\underset{\mathbf{s}}{\Longrightarrow}$ no longer contains any spatial predicates. In such a case, we check that the left also contains no spatial predicates and that the pure entailment $\Pi \Rightarrow \exists\vec{x}.\ \Pi'$ holds. Since this entailment does not involve spatial predicates, it can be sent to a standard theorem prover for first-order logic plus arithmetic. We then set the output to $\widehat{k}'$ (viewing the proof system as specifying a bottom-up search algorithm). This output gets passed down the proof tree and added to by various rules such as DEFL, INSTL, and EXISTSL.

For the soundness proof, we have

$$\Pi \Rightarrow \exists\vec{x}.\ \Pi'$$

and

$$\Gamma \vdash \{\Sigma_a * (\exists\vec{x}.\ \mathbf{emp} \wedge \Pi')\}\ \widehat{k}' \blacktriangleright_{\underline{\text{IVar}}} k$$

We must show that the following holds.

$$\Gamma \vdash \{\Sigma_a * (\mathbf{emp} \wedge \Pi)\}\ \widehat{k} \blacktriangleright_{\underline{\text{IVar}}} k$$

This is a simple application of STRENGTHENING with the following implication.

$$\Big(\Sigma_a * (\mathbf{emp} \wedge \Pi)\Big) \Rightarrow \Big(\Sigma_a * (\exists\vec{x}.\ \mathbf{emp} \wedge \Pi')\Big)$$

The implication above follows directly from our assumption that $\Pi \Rightarrow \exists\vec{x}.\ \Pi'$.

LEFTPUREFALSE This is the axiom that applies when the left-hand side has been discovered to be unsatisfiable. As with RIGHTPURE, the pure entailment $\Pi \Rightarrow \mathsf{false}$ can be checked with a standard theorem prover for classical logic with arithmetic.

For the soundness proof in this case, we have $\Pi \Rightarrow \mathsf{false}$ and must show

$$\Gamma \vdash \{\Sigma_a * (\Sigma \wedge \Pi)\}\ (\mathsf{assume}(\mathsf{false})\,\mathsf{;}\,\mathsf{halt}) \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

This is an application of FALSE from Figure 4.1 to obtain

$$\Gamma \vdash \{\mathsf{false}\}\ \mathsf{halt} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

followed by INST-ASSUME to obtain

$$\Gamma \vdash \{\mathsf{false}\}\ (\mathsf{assume}(\mathsf{false})\,\mathsf{;}\,\mathsf{halt}) \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

followed by STRENGTHENING with $\Sigma_a * (\Sigma \wedge \Pi) \Rightarrow \mathsf{false}$ to obtain our goal.

PTOMATCHES In this case, we match a points-to predicate on the left and the right. For the soundness proof, we have

$$\Gamma \vdash \{(\Sigma_a * (e \mapsto \rho)) * \varphi\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

and must show

$$\Gamma \vdash \{\Sigma_a * ((e \mapsto \rho) * \varphi)\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

which follows immediately from STRENGTHENING and associativity of $*$.

PREDMATCHES This is the same as PTOMATCHES except that we are matching an inductive predicate instance instead of a points-to predicate.

DEFL In this case, we expand an inductive predicate on the left, case splitting on the possible expansions. We insert a branch into the instrumented program, with one case for each condition Π_i. In each case, we first non-deterministically assign the $\underline{\vec{z}}_i$, then assume Π'_i, which establishes the connection between $\underline{\vec{v}}$ and $\underline{\vec{z}}_i$. Finally we insert $\widehat{k}_i$, the instrumented continuation for case i of the inductive predicate.

As an example, suppose φ is as given below

$$ls(\underline{n}_1; x, y) * ls(\underline{n}_2; y, \mathsf{nil}) \wedge \underline{n}_1 + \underline{n}_2 > 0$$

and φ' is

$$\exists z, v.\ x \mapsto [\mathsf{next} : z, \mathsf{data} : v] * ls(\underline{n}_3; z, \mathsf{nil})$$

If we then search bottom-up for a proof of

$$\Sigma_a \mathrel{[]} \varphi \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k}'} \varphi' \mathbin{/\!\!/} \widehat{k}$$

then the first step of entailment will be to case split on whether the first list segment in φ is empty. This results in the following two sub-goals

$$\Sigma_a \mathrel{[]} ls(\underline{n}_2; y, x) \wedge x = y \wedge \underline{n}_1 = 0 \wedge \underline{n}_1 + \underline{n_2} > 0 \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \Gamma_1 \vdash \widehat{k}_1$$

and

$$\begin{aligned}\Sigma_a \mathrel{[]} \exists z.\ x \mapsto [\mathsf{next} : z] * ls(\underline{n}'_1; z, y)& \\ * ls(\underline{n}_2; y, x) \wedge \underline{n}_1 + \underline{n}_2 > 0 \wedge \underline{n}_1 > 0 \wedge \underline{n}_1 = \underline{n}'_1 + 1 &\underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \Gamma_2 \vdash \widehat{k}_2\end{aligned}$$

Assuming proofs of these subgoals are found (which in this case they are), then they are combined such that the $\widehat{k}$ returned is

$$\begin{aligned}&\mathsf{branch}\ \underline{n}_1 = 0 \Rightarrow \mathsf{assume}(\mathsf{true})\,\mathtt{;}\, \widehat{k}_1,\\ &\qquad \underline{n}_1 > 0 \Rightarrow \underline{n}'_1 := ?\mathtt{;}\ \mathsf{assume}(\underline{n}_1 = \underline{n}'_1 + 1)\mathtt{;}\ \widehat{k}_2\ \mathsf{end}\end{aligned}$$

For the proof of soundness, we have the following for each i from our inductive hypotheses.

$$\Gamma \vdash \{((\varphi * \varphi_i) \wedge \Pi_i \wedge \Pi'_i) * \Sigma_a\}\ \widehat{k}_i \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

From each of these assumptions, we can construct the following proof. We write Str for Strengthening, I-E for Inst-Exists, and I-A for Inst-Assume.

$$\text{Str}\ \dfrac{\text{I-E}\ \dfrac{\text{I-A}\ \dfrac{\text{Str}\ \dfrac{\text{Ind. Hyp.}\ \dfrac{}{\Gamma_i \vdash \{((\varphi * \varphi_i) \wedge \Pi_i \wedge \Pi'_i) * \Sigma_a\}\ \widehat{k}_i \blacktriangleright_{\underline{\text{IVar}}} k}}{\Gamma_i \vdash \{(((\varphi * \varphi_i) \wedge \Pi_i) * \Sigma_a) \wedge \Pi'_i\}\ \widehat{k}_i \blacktriangleright_{\underline{\text{IVar}}} k}}{\begin{array}{l}\Gamma_i \vdash \{(((\varphi * \varphi_i) \wedge \Pi_i) * \Sigma_a) \wedge \Pi'_i\} \\ \qquad \mathsf{assume}(\Pi'_i)\,\mathsf{;}\,\widehat{k}_i \blacktriangleright_{\underline{\text{IVar}}} k\end{array}}}{\begin{array}{l}\Gamma_i \vdash \{\exists\vec{z}_i.\ (((\varphi * \varphi_i) \wedge \Pi_i) * \Sigma_a) \wedge \Pi'_i\} \\ \qquad \vec{z}_i := ?\,\mathsf{;}\,\mathsf{assume}(\Pi'_i)\,\mathsf{;}\,\widehat{k}_i \blacktriangleright_{\underline{\text{IVar}}} k\end{array}}}{\begin{array}{l}\Gamma_i \vdash \{(\varphi * (\exists\vec{z}_i.\ \varphi_i{\wedge}\Pi'_i) * \Sigma_a) \wedge \Pi_i\} \\ \qquad \vec{z}_i := ?\,\mathsf{;}\,\mathsf{assume}(\Pi'_i)\,\mathsf{;}\,\widehat{k}_i \blacktriangleright_{\underline{\text{IVar}}} k\end{array}}\ \vec{z}_i \notin \mathit{fv}(\varphi, \Sigma_a, \Pi_i)$$

Note that each assumption now has a precondition of the form below

$$(\varphi * (\exists\vec{z}_i.\ \varphi_i \wedge \Pi'_i) * \Sigma_a) \wedge \Pi_i \tag{5.7}$$

Our goal is to show that the following holds, where $\widehat{k}_b$ is the branch in the conclusion of the rule.

$$\Gamma \vdash \{(\varphi * d(\vec{e})) * \Sigma_a\}\ \widehat{k}_b \blacktriangleright_{\underline{\text{IVar}}} k$$

By expanding d according to the same specification used in the premise of the rule we are considering, we can see that the precondition in this formula is equivalent to the following.

$$\varphi * \left(\bigvee_i \Big(\lceil C_i(\vec{e})\rceil\Big)\right) * \Sigma_a$$

Recall that $\lceil C_i(\vec{e})\rceil$ gives the interpretation of $C_i(\vec{e})$ as a separation logic formula. Applying the definition of $\lceil C_i(\vec{e})\rceil$ we obtain the following

$$\varphi * \left(\bigvee_i \Big(\Pi_i \wedge (\exists z_i.\ \Pi'_i \wedge \varphi_i)\Big)\right) * \Sigma_a$$

By commuting and re-associating terms, we can rewrite this such that it is equal to equation (5.7) for each i. The soundness of the branch that we add will then follow from an n-ary version of the derived rule given below.

$$\frac{Q \Rightarrow (Q_1 \wedge e_1) \vee (Q_2 \wedge e_2) \qquad \Gamma \vdash \{Q_1 \wedge e_1\}\ \widehat{k_1} \blacktriangleright_V k \qquad \Gamma \vdash \{Q_2 \wedge e_2\}\ \widehat{k_2} \blacktriangleright_V k}{\Gamma \vdash \{Q\}\ \mathsf{branch}\ e_1 \Rightarrow \widehat{k_1}, e_2 \Rightarrow \widehat{k_2}\ \mathsf{end} \blacktriangleright_V k}$$

This rule is simply Inst-Branch from Section 4.1.3 but with the premise $Q \Rightarrow (Q_1 \wedge e_1) \vee (Q_2 \wedge e_2)$ instead of $Q \Rightarrow e_1 \vee e_2$ and preconditions $Q_i \wedge e_i$ instead of $Q \wedge e_i$. The reasoning used to justify it is the same.

InstL This rule is responsible for unifying the names of instrumentation variables. For example, if the left-hand side of the sequent contains $\mathit{ls}(\underline{n}{+}1; x, \mathsf{nil})$ and the right-hand side contains $\mathit{ls}(\underline{n}; x, \mathsf{nil})$ then we cannot apply PredMatches to remove these nearly matching spatial formulae until we have made the instrumentation variables match. Since we are allowed to insert new commands that affect the instrumentation variables, we can add the command $\underline{n} := \underline{n} + 1$ in order to connect the two formulae. The post-condition of the left-hand side after executing this command is then $\mathit{ls}(\underline{n}; x, \mathsf{nil})$ and the PredMatches rule can be applied.

In order to show soundness, we assume $x \notin \mathit{fv}(\Sigma_a)$ and

$$\Gamma \vdash \{\Sigma_a * \varphi\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

By the Inst-Assign rule and the backward Hoare logic rule for assignment, we have

$$\Gamma \vdash \{(\Sigma_a * \varphi)[e/\underline{x}]\}\ (\underline{x} := e\,;\widehat{k}) \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

We will then apply Strengthening to show that our goal, given below, follows.

$$\Gamma \vdash \{\Sigma_a * \varphi[e/\underline{x}]\}\ (\underline{x} := e\,;\widehat{k}) \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

To do so, we must prove the implication

$$\Big(\Sigma_a * \varphi[e/\underline{x}]\Big) \Rightarrow \Big((\Sigma_a * \varphi)[e/\underline{x}]\Big)$$

We assume $\Sigma_a * \varphi[e/\underline{x}]$. Then since $x \notin \mathit{fv}(\Sigma_a)$ we can extend the scope of the substitution, obtaining the needed result.

$$(\Sigma_a * \varphi)[e/\underline{x}]$$

EXISTSR This is the rule used to instantiate existentially quantified variables on the right of $\underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k'}}$. Reading it from top to bottom, if $\varphi'[e/x]$ follows from φ, then $\exists x.\ \varphi'$ follows from φ.

For soundness, we assume that for some Γ, k we have $\Gamma \vdash \{\Sigma_a * (\exists x.\ \varphi')\}\ \widehat{k'} \blacktriangleright_{\underline{\text{IVar}}} k$. We can then use strengthening and the implication $\varphi'[e/x] \Rightarrow \exists x.\ \varphi'$ to obtain

$$\Gamma \vdash \{\Sigma_a * \varphi'[e/x]\}\ \widehat{k'} \blacktriangleright_{\underline{\text{IVar}}} k$$

From our inductive hypothesis we have

$$\Gamma \vdash \{\Sigma_a * \varphi'[e/x]\}\ \widehat{k'} \blacktriangleright_{\underline{\text{IVar}}} k \ \text{ implies } \ \Gamma \vdash \{\Sigma_a * \varphi\}\ \widehat{k} \blacktriangleright_{\underline{\text{IVar}}} k$$

As we have established the antecedent of this implication, we can conclude

$$\Gamma \vdash \{\Sigma_a * \varphi\}\ \widehat{k} \blacktriangleright_{\underline{\text{IVar}}} k$$

which is our goal.

EXISTSL This rule governs the elimination of existentially quantified variables on the left and is justified using the INST-EXISTS rule from Figure 4.1. We introduce a fresh variable c for the quantified variable, as this renaming is performed by our implementation. It is not strictly necessary for soundness.

We must show the following

$$\Gamma \vdash \{\Sigma_a * (\exists x.\ \varphi)\}\ \underline{c} := ?\,;\widehat{k} \blacktriangleright_{\underline{\text{IVar}}} k$$

and we have the following as an assumption.

$$\Gamma \vdash \{\Sigma_a * \varphi[\underline{c}/x]\}\ \widehat{k} \blacktriangleright_{\underline{\text{IVar}}} k$$

We first apply INST-EXISTS to obtain the statement below.

$$\Gamma \vdash \{\exists \underline{c}.\ \Sigma_a * \varphi[\underline{c}/x]\}\ \underline{c} := ?\,;\widehat{k} \blacktriangleright_{\underline{\text{IVar}}} k$$

That $\underline{c}$ is fresh implies $\underline{c} \notin fv(\Sigma_a)$ and thus we have that $\exists \underline{c}.\ \Sigma_a * \varphi[\underline{c}/x]$ implies $\Sigma_a * (\exists \underline{c}.\ \varphi[\underline{c}/x])$. Applying Strengthening with this implication yields the following.

$$\Gamma \vdash \{\Sigma_a * (\exists \underline{c}.\ \varphi[\underline{c}/x])\}\ \underline{c} := \mathtt{?;}\, \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

We then note that since $\underline{c}$ is fresh and thus $\underline{c} \notin fv(\varphi)$, the formula $\exists \underline{c}.\ \varphi[\underline{c}/x]$ is an alpha-varying of $\exists x.\ \varphi$. We thus have that $\exists x.\ \varphi$ implies $\exists \underline{c}.\ \varphi[\underline{c}/x]$ and can apply Strengthening again to obtain the following, which is our goal.

$$\Gamma \vdash \{\Sigma_a * (\exists x.\ \varphi)\}\ \underline{c} := \mathtt{?;}\, \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

Proof Search Structure

There are many potential search techniques involving the rules presented in Figures 5.8 and 5.9. Here we discuss the choices we made in our implementation of this proof system.

Our proof search procedure starts by eliminating all existentials on the left with the ExistsL rule. Any new existentials that appear on the left during the search (e.g. by the expansion of definitions) are also eliminated as soon as they arise. The procedure then proceeds by inferring pure consequences of the heap assumptions (rules NotNull and Disjoint), propagating equalities (rule PropEqL), introducing constants for existentials on the left (ExistsL), expanding definitions (rule DefL) and matching spatial predicates (rules PtoMatches and PredMatches). As spatial predicates are matched, they are moved to the portion of the sequent to the left of the $[]$ symbol. Once all spatial predicates in φ' have been matched, then the proof search can terminate with the RightPure rule, closing off the current branch. The search can also succeed via the LeftPureFalse rule if the antecedent ever becomes inconsistent. The pure entailment checks present in the premises of these rules (for example, $\Pi \Rightarrow \exists \vec{x}.\ \Pi'$) can be implemented as a call to an automated theorem prover for classical logic. We use the SMT solver Yices [Dutertre and Moura, 2006], but any prover with support for existential quantifiers and unbounded integer variables would work.

There are a few rules that would seem to interfere with an efficient implementation of the proof system. The ExistsR and InstL rules both require us to guess a substitution to apply when moving from the inputs in the conclusion to the inputs in the premise. However, this substitution can be delayed until the term to be substituted is clear. In our implementation, we only apply these rules when attempting to match spatial predicates via the PTOMATCHES or PREDMATCHES rules. In such cases, we may have, for example

$$x \mapsto [\mathsf{next} : a, \mathsf{data} : b] * \varphi$$

on the left and

$$\exists z, q.\ x \mapsto [\mathsf{next} : z, \mathsf{data} : q] * \varphi'$$

on the right. In this case, we can apply the EXISTSR rule to instantiate z with a and q with b, which results in the two point-to predicates matching according to the PTOMATCHES rule.

Inductive Specifications

The DEFL rule first looks up a specification for the inductive predicate d in the set of specifications $\mathbf{S}$. If there are multiple specifications, any one may be chosen. The side conditions on this rule can always be satisfied by applying alpha conversion, since $\vec{z_i}$ is considered bound in “let $\vec{z_i}$ satisfy Π'_i in φ_i.”

This expansion of inductive predicates is a potential source of non-termination for our proof search. If we are not careful, we can end up repeatedly expanding definitions on the left. The DEFL rule is also the only source of branching in the proof system and the number of inductive predicate expansions applied has a large effect on the running time of our proof search. To combat both these problems, we restrict the number of times a predicate can be expanded. In our implementation, we associate an integer with each inductive predicate instance and increment this counter each time the instance is expanded. This integer starts at zero and, when it reaches some bound, we do not allow further expansion of that predicate instance. The bound can be set via a command line argument. We have found that a bound of one (allowing each predicate instance to be expanded once) is usually sufficient, however in some cases two expansions are required. With a bound of two, we have not yet had an example fail verification where the reason for failure was too few predicate expansions (any failures have always been related to failure of the abstraction heuristics described in Section 5.7 or failure to make the appropriate inductive predicate specification available to the system).

Since predicate expansions are so costly in terms of execution time, we try to perform them only when necessary. Our proof search will only apply DEFL when no other rules are applicable. When we do apply the expansion rules, we try to intelligently choose the appropriate specification from $\mathbf{S}$ to use. Suppose we are applying DEFL to our current goal formula. We will look at the formula on the right of the $\underset{\mathbf{S}}{\Longrightarrow}_f$ arrow and see what spatial predicates have not yet been matched. We then select a definition that can expose a predicate matching one of the predicates we have on the right.

To compute what predicates a definition may generate, we start from an instance of the definition with distinct variables in each argument position, say $d(\vec{x})$. We then recursively expand d. As we perform the expansions, we replace any fresh variables that would be generated with a wildcard variable. We also replace non-address variables with wildcards and only record which non-$\mathbf{emp}$ spatial predicates are generated. Thus, we only track what happens to the pointer-valued arguments of d during expansion. For example, suppose we have the doubly-linked list specification below.

$$\begin{aligned}
&\mathrm{dll}(\underline{k};\, p, \mathit{first}, \mathit{last}, n) \ \texttt{<=>} \\
&\qquad \underline{k} = 0 : \text{let } [\,] \text{ satisfy } \mathsf{true} \text{ in } \mathbf{emp} \wedge \mathit{first} = n \wedge \mathit{last} = p \\
&\quad\ \mid\ \underline{k} > 0 : \text{let } \underline{k}' \text{ satisfy } \underline{k} = \underline{k}' + 1 \text{ in} \\
&\qquad\qquad\qquad\qquad \exists z.\ (\mathit{first} \mapsto [\mathsf{prev} : p, \mathsf{next} : z]) * \mathrm{dll}(\underline{k}'; \mathit{first}, z, \mathit{last}, n))
\end{aligned}$$

Using _ to represent a wildcard variable, and expanding $\mathrm{dll}(\_; a, b, c, d)$ once (and discarding non-spatial predicates), we obtain the following.

$$b \mapsto [\mathsf{prev} : a, \mathsf{next} : \_] \qquad \mathrm{dll}(\_; b, \_, c, d)$$

The first pattern cannot be expanded further, but the second pattern can. If we expand $\mathrm{dll}(\_; b, \_, c, d)$ we obtain the following.

$$\_ \mapsto [\mathsf{prev} : b, \mathsf{next} : \_] \qquad \mathrm{dll}(\_; \_, \_, c, d)$$

At this point, expanding any of these patterns results only in patterns that have already been generated. Thus, we have generated all the patterns that will result from expanding $\mathrm{dll}(\_; a, b, c, d)$.

We then store these patterns in a data structure that supports efficient querying. This is essentially a multimap from patterns to specifications that is aware of unification. Suppose we look up $\exists z.\ x \mapsto [\mathsf{prev} : y, \mathsf{next} : z]$. The map will see that this matches $b \mapsto [\mathsf{prev} : a, \mathsf{next} : \_]$. It will bind b to x and a to y and return as one of its results the pattern $\mathrm{dll}(\_; y, x, \_, \_)$ along with the specification that was used to obtain it. This indicates that expanding a predicate instance matching $\mathrm{dll}(\_; y, x, \_, \_)$ will produce a points-to predicate that matches $\exists z.\ x \mapsto [\mathsf{prev} : y, \mathsf{next} : z]$. We then search the left formula of our current goal for such a spatial formula matching $\mathrm{dll}(\_; y, x, \_, \_)$, expand it, and proceed.

We can generate this pattern map on program start-up as soon as we read in the list of inductive predicate specifications provided by the user, after which it benefits every proof search performed by the analysis (and there are typically hundreds of frame inference queries even for small examples). Applying this optimization significantly speeds up our proof search. Furthermore, proof search is by far the major contributor to running time, thus any proof search optimizations have a large effect on total running time of the analysis.

Note that we do not have a corresponding "DEFR" rule for expanding definitions on the right. Such a rule could be added, but has proved unnecessary in our experiments. We comment further on this in Section 5.7, which discusses abstraction, as this is the operation that renders DEFR unnecessary.

5.5.2 `implies`

We now show how the proof system just presented is used to implement the `implies` function. On page on the following page we give the implementation of `implies`. The function call `implies`$(\varphi, \varphi', \widehat{k'})$ takes the following arguments.

φ An antecedent formula.

φ' The consequent formula.

$\widehat{k}'$ An instrumentation of some continuation under precondition φ'.

Given an instrumentation $\widehat{k}'$ of some continuation k starting from the precondition φ', a call to $\texttt{implies}(\varphi, \varphi', \widehat{k}')$ returns $\mathsf{Some}\big(\widehat{k}\big)$ if it can establish that $\widehat{k}$ is an instrumentation of k with precondition φ. That is, if $\texttt{implies}(\varphi, \varphi', \widehat{k}') = \mathsf{Some}\big(\widehat{k}\big)$ then for all k

$$\Gamma \vdash \{\varphi'\}\ \widehat{k}' \blacktriangleright_{\underline{\text{IVar}}} k$$

implies

$$\Gamma \vdash \{\varphi\}\ \widehat{k} \blacktriangleright_{\underline{\text{IVar}}} k$$

Function $\texttt{implies}(\varphi, \varphi', \widehat{k}')$. Assumes that $\Gamma \vdash \{\varphi'\}\ \widehat{k}' \blacktriangleright_{\underline{\text{IVar}}} k$ for some Γ and k. If so, and $\texttt{implies}$ returns $\mathsf{Some}\big(\widehat{k}\big)$ then $\Gamma \vdash \{\varphi\}\ \widehat{k} \blacktriangleright_{\underline{\text{IVar}}} k$ holds for the same Γ and k.

```
let $(\varphi_a, \mathbf{c}_a) = \texttt{abstract}(\varphi)$ in
  if $\varphi_a \underset{\mathbf{S}}{\Longrightarrow}_{\widehat{k}'} \varphi' \mathbin{/\!\!/} \widehat{k}$ then
    return $\mathsf{Some}\big(\Gamma, (\mathbf{c}_a ; \widehat{k})\big)$
else return $\mathsf{None}$
```

The function first calls $\texttt{abstract}(\varphi)$ in order to simplify the state formula. In particular, $\texttt{abstract}$ will fold inductive predicate definitions, which is something that our entailment system does not do—entailment will only expand predicates on the left. For example, $\texttt{abstract}(\exists k.\ x \mapsto [\mathsf{next} : k] * k \mapsto [\mathsf{next} : \mathsf{nil}])$ will return $ls(\underline{n}; x, \mathsf{nil})$ and the instrumentation command $\underline{n} := 2$. Entailment is not able to create instances of data structures, nor for example to take

$$\exists z.\ x \mapsto [\mathsf{next} : z] * ls(\underline{n}; z, \mathsf{nil})$$

and discover this implies $ls(\underline{n} + 1; z, \mathsf{nil})$.

This is a deliberate choice, as restricting entailment only to expansionary rules significantly decreases the search space and helps prevent cycles in the proof search. By combining the expansionary behavior of entailment with the collapsing or summarizing behavior of abstraction, we are able to perform all the inference steps necessary for our instrumentation procedure while increasing efficiency of the component operations.

Following the call to `abstract`, the `implies` function then calls into entailment, passing in the continuation k. It then returns the instrumentation $\widehat{k}$ that is discovered by entailment.

That `implies` satisfies its specification from Figure 5.7 follows directly from Corollary 5 and the specification of `abstract`.

5.5.3 Frame Inference

We now consider a slight modification of the proof system presented in Section 5.5.1. Whereas the original proof system was able to answer queries of the form $\varphi \Rightarrow \varphi'$, the new system permits the case where φ' specifies a sub-heap of φ (implication, in contrast, requires both formulae to describe heaps with the same domain). The problem is very similar to the *frame inference problem* described in Berdine et al. [2005], but differs in that we will need to produce instrumentation commands during the proof search. The *frame* refers to that portion of the heap described by the hypothesis which is not in the conclusion. Inferring frames is useful when a particular command requires a piece of heap to exist but does not care whether the heap contains additional elements.

As an example of such a situation, consider the symbolic state

$$\varphi \overset{\text{def}}{=} \mathit{ls}(\underline{n}; x, \mathsf{nil}) \land x \neq \mathsf{nil}$$

Suppose we are trying to take the post-condition of this state with respect to the command $x := x.\mathsf{next}$. Doing so requires us to show that a heap cell at x exists. In this case, such a cell does exist since φ implies the following formula.

$$\varphi' \overset{\text{def}}{=} \exists z, v.\ x \mapsto [\mathsf{next} : z, \mathsf{data} : v] * \mathit{ls}(\underline{n} - 1; z, \mathsf{nil})$$

However, we don't generally know this expanded version of the state formula. We would like to be able to ask our proof system to show that x is in the heap and obtain φ' while providing only φ and x. This is the sort of query facilitated by our system for frame inference.

Frame inference is also useful for answering *pure entailments*. Suppose we have the symbolic state

$$\varphi \stackrel{\text{def}}{=} ls(\underline{n}; x, \mathsf{nil}) \wedge \underline{n} = 0$$

and we want to know whether this implies $x = \mathsf{nil}$. In this case, we can ask whether the implication below holds.

$$\varphi \Rightarrow x = \mathsf{nil}$$

But note that this is different from the implications considered in Section 5.5.1. In the previously-presented proof system for entailment, there was a spatial aspect to the proving—we wanted all of the heap described by the antecedent to be accounted for by the consequent. In this example, since the consequent is pure, we do not have this requirement. The antecedent is allowed to describe any amount of heap. Such a situation is captured by asking whether there is a frame that allows us to show $x = \mathsf{nil}$ follows from φ (the particular frame does not matter, we only check that a valid frame exists).

Pure entailment could also be handled by our system for entailment from Section 5.5.1 if we allowed true to appear as a spatial formula. The example query above would then correspond to the implication $\varphi \Rightarrow (x = \mathsf{nil}) * \mathsf{true}$. However, since we do not have "$*\mathsf{true}$" in our language of symbolic state formulae, pure entailment is more naturally built on top of frame inference.

Formulae with holes In order to account for queries such as "does the heap contain a cell at address x?" which arise frequently when checking memory safety, we allow the consequent of a frame inference query to contain the special points-to predicate $x \mapsto \square$. The $\square$ will match any record expression and is only allowed to occur once in any symbolic state formula. Thus, the predicate $x \mapsto \square$ states that the heap contains a cell at address x, but provides no information about the contents of the heap cell. This predicate is satisfied by any heap consisting of a single cell at x. In particular, the set of fields present at x do not matter, so the following are both valid implications.

$$x \mapsto [\mathsf{next} : \mathsf{nil}] \Rightarrow x \mapsto \square$$
$$x \mapsto [\mathsf{next} : y, \mathsf{data} : 0] \Rightarrow x \mapsto \square$$

Formally, we can give a semantics for $x \mapsto \square$ by extending the satisfaction relation in Figure 2.7 with the following case.

$$(s, h) \models_X e^{\mathrm{a}} \mapsto \square \quad \Leftrightarrow \quad h = \{((\llbracket e^{\mathrm{a}} \rrbracket\, s), r)\} \text{ for some } r \in \textit{Records}$$

The predicate $x \mapsto \square$ essentially acts as a pattern, ensuring that frame inference exposes a points-to at the appropriate address. This operates somewhat like the common separation logic abbreviation $x \mapsto -$, which is frequently used as shorthand for $\exists y.\ x \mapsto y$. If we had variables of record type and permitted existential quantification over these, such that y in $\exists y.\ x \mapsto y$ could represent some set of field bindings, then we could use a similar abbreviation. Since we make limited use of these patterns (in particular, since we only require at most one in any formula), we found it simpler to work with the weaker $x \mapsto \square$ form and avoid the complexities of introducing more types of variable.

Judgment Form and Soundness

As just mentioned, our primary use of frame inference is to expose heap cells needed to compute post-conditions for heap-manipulating commands. The structure of the judgment we define must change slightly to accommodate this usage. The interface we will adopt is the following.

Input: φ A symbolic state formula describing the current state.

φ' A symbolic state formula describing the heap that is required to be present.

f_k A function that takes a formula φ'' and produces an optional pair $(\Gamma', \widehat{k}')$, where Γ' is a context and $\widehat{k}'$ is an instrumented continuation.

S A set of inductive predicate specifications describing the data structures used.

We also require that the input satisfy the following invariant:

$$\text{If } f_k = \mathsf{Some}\big(\Gamma', \widehat{k}'\big) \text{ then } \Gamma' \vdash \{\varphi'\}\ \widehat{k}' \blacktriangleright_{\underline{\text{IVar}}} k$$

Note that f_k is parameterized by the continuation k that it produces an instrumentation of. This parameter is included to help make it clear which k is being considered during examples and proofs.

Output: $\widehat{k}$ An instrumentation of k.

Γ A context.

These outputs must satisfy $\Gamma \vdash \{\varphi\}\ \widehat{k} \blacktriangleright_{\underline{\text{IVar}}} k$.

The form of our judgment for frame inference will be the following.

$$\varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \Gamma \vdash \widehat{k}$$

where φ, φ', $\mathbf{S}$, and f_k are considered inputs and Γ and $\widehat{k}$ are the outputs.

Relation to Entailment The function f_k in frame inference corresponds to the input $\widehat{k}'$ from entailment. One might wonder why frame inference requires this input to be a function while a single-valued input sufficed for entailment. The reason is that, when searching for a frame that shows φ contains φ', we may find different frames along different branches of the proof.

For example, let φ be the following formula

$$(\mathit{ls}(\underline{n}_1; x, y) * \mathit{ls}(\underline{n}_2; y, x)) \wedge (\underline{n}_1 + \underline{n}_2 > 0)$$

and suppose we want to show the following.

$$\varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} x \mapsto \square \mathbin{/\!\!/} \Gamma \vdash \widehat{k}$$

We know from $\underline{n}_1 + \underline{n}_2 > 0$ that at least one of the two lists is non-empty and thus x is in the heap. However, the portion of the heap that remains when we separate out x is different depending on whether $\underline{n}_1 > 0$. If $\underline{n}_1 > 0$ then we have that φ implies the following.

$$\exists z, v.\ x \mapsto \big[\mathsf{next} : z, \mathsf{data} : v\big] * \mathit{ls}(\underline{n}_1 - 1; z, y) * \mathit{ls}(\underline{n}_2; y, x) \tag{5.8}$$

If $\underline{n}_1 = 0$ then we have that φ implies the formula below.

$$\exists z, v.\ (y \mapsto [\mathsf{next} : z, \mathsf{data} : v] * ls(\underline{n}_2 - 1; z, x)) \land x = y \tag{5.9}$$

We use the function f_k to account for this. In the above example, f_k would be expected to produce an instrumentation for each of these possible preconditions. Let φ_1 be formula (5.8) and φ_2 be formula (5.9). If $f_k(\varphi_1) = \mathsf{Some}\big(\Gamma_1, \widehat{k}_1\big)$ and $f_k(\varphi_2) = \mathsf{Some}\big(\Gamma_2, \widehat{k}_2\big)$ then a valid instrumentation from the precondition φ is

$$\begin{aligned} \mathsf{branch}\ \underline{n}_1 > 0 &\Rightarrow \widehat{k}_1, \\ \underline{n}_1 = 0 &\Rightarrow \widehat{k}_2\ \mathsf{end} \end{aligned}$$

Let this continuation be $\widehat{k}$. We then have the following.

$$\Gamma_1 \cup \Gamma_2 \vdash \{\varphi\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

This fact—that the output of frame inference results in a valid instrumentation of k—is the main soundness theorem for frame inference and is discussed further below.

As with entailment, we track some extra bookkeeping information during the search for a proof in the form of a list of matched spatial formulae Σ_a. This plays the same role it did in entailment and is described on page 223. The statement $\varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \Gamma \vdash \widehat{k}$ is an abbreviation for the following judgment, which tracks this extra information.

$$\Sigma_a \mathbin{[]} \varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \Gamma \vdash \widehat{k}$$

Soundness As with entailment, the soundness result we will seek states that the output of frame inference is a valid instrumentation.

Theorem 29. *If* $\Sigma_a \mathbin{[]} \varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \Gamma \vdash \widehat{k}$ *is derivable then so is*

$$\Gamma \vdash \{\Sigma_a * \varphi\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

Stated in terms of our abbreviated form of judgment, this becomes the following.

Corollary 6. *If $\varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!/} \Gamma \vdash \widehat{k}$ is derivable then so is*

$$\Gamma \vdash \{\varphi\}\, \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

Since a major use of frame inference in our system is to rewrite symbolic state formulae into a given form, it is also worth showing that the function f_k is called with arguments of the appropriate form. This is captured by the following theorem, which states that the instrumentation function f_k is only called with symbolic states φ which have been shown to describe a heap containing some sub-heap satisfying φ', the symbolic state formula to the right of the $\underset{\mathbf{S}}{\Longrightarrow}$.

Theorem 30. *In a derivation of*

$$\Sigma_a \mathbin{[\!]} \varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!/} \Gamma \vdash \widehat{k}$$

The function f_k is only called with inputs of the form $(\varphi'' * \Sigma_a)$ for some φ'' such that $\varphi'' \Rightarrow \varphi' * \mathsf{true}$.

Stated in terms of our abbreviated form of judgment, this becomes the following.

Corollary 7. *In a derivation of $\varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!/} \Gamma \vdash \widehat{k}$, the function f_k is only called with inputs φ'' such that $\varphi'' \Rightarrow \varphi' * \mathsf{true}$.*

Rules and Proof of Soundness

We now present the rules for frame inference along with a proof of Theorems 29 and 30 (which are shown by structural induction on the frame inference derivation). Most of the rules are the same as for entailment, with the only difference being the replacement of input $\widehat{k}'$ with the input function f_k and the inclusion of the output context Γ. For example, the rule PROPEQL becomes the following.

$$\textsc{PropEqL}\quad \frac{\Sigma_a \mathbin{[\!]} \varphi[e/x] \wedge x = e \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!/} \Gamma \vdash \widehat{k}}{\Sigma_a \mathbin{[\!]} \varphi \wedge x = e \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!/} \Gamma \vdash \widehat{k}}$$

$$\textsc{PropEqL}\quad \frac{\Sigma_a \mathrel{[\!]} \varphi[e/x] \wedge x = e \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \Gamma \vdash \widehat{k}}{\Sigma_a \mathrel{[\!]} \varphi \wedge x = e \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \Gamma \vdash \widehat{k}}$$

$$\textsc{NotNull}\quad \frac{(e \mapsto \rho) \in (\Sigma_a * \varphi) \qquad \Sigma_a \mathrel{[\!]} \varphi \wedge (e \neq \mathsf{nil}) \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \Gamma \vdash \widehat{k}}{\Sigma_a \mathrel{[\!]} \varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \Gamma \vdash \widehat{k}}$$

$$\textsc{Disjoint}\quad \frac{((e_1 \mapsto \rho_1) * (e_2 \mapsto \rho_2)) \in (\Sigma_a * \varphi) \qquad \Sigma_a \mathrel{[\!]} \varphi \wedge (e_1 \neq e_2) \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \Gamma \vdash \widehat{k}}{\Sigma_a \mathrel{[\!]} \varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \Gamma \vdash \widehat{k}}$$

$$\textsc{LeftPureFalse}\quad \frac{\Pi \Rightarrow \mathsf{false\ is\ valid}}{\Sigma_a \mathrel{[\!]} \Sigma \wedge \Pi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \Gamma \vdash \mathsf{assume(false)}\texttt{;}\ \mathsf{halt}}$$

$$\textsc{PtoMatches}\quad \frac{\Sigma_a * (e \mapsto \rho) \mathrel{[\!]} \varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \Gamma \vdash \widehat{k}}{\Sigma_a \mathrel{[\!]} (e \mapsto \rho) * \varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' * (e \mapsto \rho) \mathbin{/\!\!/} \Gamma \vdash \widehat{k}}$$

$$\textsc{PredMatches}\quad \frac{\Sigma_a * d(\vec{e}) \mathrel{[\!]} \varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \Gamma \vdash \widehat{k}}{\Sigma_a \mathrel{[\!]} d(\vec{e}) * \varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' * d(\vec{e}) \mathbin{/\!\!/} \Gamma \vdash \widehat{k}}$$

$$\textsc{InstL}\quad \frac{\Sigma_a \mathrel{[\!]} \varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \Gamma \vdash \widehat{k}}{\Sigma_a \mathrel{[\!]} \varphi[e/\underline{x}] \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \Gamma \vdash (\underline{x} := e\texttt{;}\,\widehat{k})}\ \underline{x} \notin fv(\Sigma_a), fv(x, e)$$

$$\textsc{ExistsR}\quad \frac{\Sigma_a \mathrel{[\!]} \varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi'[e/x] \mathbin{/\!\!/} \Gamma \vdash \widehat{k}}{\Sigma_a \mathrel{[\!]} \varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \exists x.\, \varphi' \mathbin{/\!\!/} \Gamma \vdash \widehat{k}}$$

$$\textsc{ExistsL}\quad \frac{\Sigma_a \mathrel{[\!]} \varphi[c/x] \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \Gamma \vdash \widehat{k}}{\Sigma_a \mathrel{[\!]} \exists x.\, \varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \Gamma \vdash c := ?\texttt{;}\,\widehat{k}}\ c \text{ fresh}$$

Figure 5.10: Rules for frame inference that are the same as for entailment.

The full list of rules that are essentially unchanged is given in Figure 5.10.

The first rule that is different is RIGHTPURE. In the system for frame inference, rather than returning the $\widehat{k}'$ that was passed in as the output instrumentation, we instead call f_k to obtain the output instrumentation. We also no longer require that the spatial portion of the left-hand formula be empty. The new rule is given in Figure 5.11.

We also must change the DEFL rule to account for the fact that each branch of the proof may return a different context (the other rules do not branch and thus just pass the context from the premise through to the conclusion). The new rule merges the contexts from the premises using the union operation defined for contexts on page 204. The updated version is given in Figure 5.11.

Finally, we must add a rule to handle our new $x \mapsto \square$ construct. This is given as rule PTOMATCHESANY in Figure 5.11 and captures the fact that $x \mapsto \square$ on the right matches any points-to predicate of the form $x \mapsto \rho$ on the left.

Proof of Soundness The proof of Theorem 29 for the rules in Figure 5.10 is the same as for Theorem 28, which was described on page 243. The only difference is the presence of Γ and the fact that f_k is a function.

We take the rule PROPEQL as a representative example. In the proof for PROPEQL for entailment we showed that given

$$\Gamma \vdash \{\Sigma_a * (\varphi[e/x] \wedge x = e)\} \; \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k \tag{5.10}$$

we can derive the following by application of the STRENGTHENING rule from Figure 4.1.

$$\Gamma \vdash \{\Sigma_a * (\varphi \wedge x = e)\} \; \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

For entailment, the inductive hypothesis and our goal were both implications and (5.10) was the conclusion of the inductive hypothesis. In the soundness theorem for frame inference, we get (5.10) directly from the inductive hypothesis. Once (5.10) is obtained, further reasoning is the same. We apply STRENGTHENING with the implication below.

$$\Big(\Sigma_a * (\varphi \wedge x = e)\Big) \Rightarrow \Big(\Sigma_a * (\varphi[e/x] \wedge x = e)\Big)$$

We now consider the rules in Figure 5.11.

RightPure

$$\frac{\Pi \Rightarrow \exists \vec{x}.\, \Pi' \qquad f_k(\exists \vec{x}.\, (\Sigma_a * \Sigma) \wedge \Pi') = \mathsf{Some}\big(\Gamma, \widehat{k}\big)}{\Sigma_a \,[\!]\, \Sigma \wedge \Pi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \exists \vec{x}.\, \mathbf{emp} \wedge \Pi' \mathbin{/\!\!/} \Gamma \vdash \widehat{k}}$$

DefL

$$\frac{\begin{array}{c}\big(d(\vec{v}) \;\texttt{<=>}\; \ldots \mid C_i(\vec{v}) \mid \ldots\big) \in \mathbf{S} \\ C_i(\vec{e}) = \big(\Pi_i : \text{let } \underline{\vec{z_i}} \text{ satisfy } \Pi_i' \text{ in } \varphi_i\big) \\ \forall i.\, \big(\Sigma_a \,[\!]\, (\varphi * \varphi_i) \wedge \Pi_i \wedge \Pi_i' \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \Gamma_i \vdash \widehat{k_i}\big)\end{array}}{\begin{array}{l}\Sigma_a \,[\!]\, \varphi * d(\vec{e}) \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \\ \bigcup_i (\Gamma_i) \vdash \mathsf{branch}\ \ldots, \Pi_i \Rightarrow \underline{\vec{z_i}} := ?\,\texttt{;}\, \mathsf{assume}(\Pi_i')\,\texttt{;}\, \widehat{k_i}, \ldots\ \mathsf{end}\end{array}} \;\forall i.\, \vec{z_i} \notin \mathit{fv}(\varphi, \Sigma_a, \Pi_i)$$

PtoMatchesAny

$$\frac{\Sigma_a * (e \mapsto \rho) \,[\!]\, \varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \Gamma \vdash \widehat{k}}{\Sigma_a \,[\!]\, (e \mapsto \rho) * \varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' * (e \mapsto \square) \mathbin{/\!\!/} \Gamma \vdash \widehat{k}}$$

Figure 5.11: Rules for frame inference that differ from those for entailment.

RightPure We are given $\Pi \Rightarrow \exists \vec{x}.\, \Pi'$ from the first premise and

$$\Gamma \vdash \{(\Sigma_a * \Sigma) \wedge \Pi'\}\; \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

from our requirement that f_k produce valid instrumentations of k. We then must show the following.

$$\Gamma \vdash \{(\Sigma_a * \Sigma) \wedge \Pi\}\; \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

This follows from our assumption on $\widehat{k}$ by the Strengthening rule from Figure 4.1 together with the implication below.

$$(\Sigma_a * \Sigma) \wedge \Pi \Rightarrow \exists \vec{x}.\, (\Sigma_a * \Sigma) \wedge \Pi'$$

The implication holds since $\Pi \Rightarrow \exists \vec{x}.\, \Pi'$ implies the following.

$$(\Sigma_a * \Sigma) \wedge \Pi \Rightarrow (\Sigma_a * \Sigma) \wedge (\exists \vec{x}.\, \Pi')$$

The scope of the existential on $\vec{x}$ can then be extended, as $\exists \vec{x}.\, \Pi'$ can always be alpha-varied such that $\vec{x} \cap \mathit{fv}(\Sigma_a, \Sigma) = \emptyset$.

PTOMATCHESANY This case follows the same reasoning as for PTOMATCHES, as the only difference in the rules involves the formula on the right-hand side of the sequent arrow, which does not participate in the statement of this theorem.

DEFL We have the following from our inductive hypothesis applied to each premise $\left(\Sigma_a \mathbin{\Box} (\varphi * \varphi_i) \wedge \Pi_i \wedge \Pi'_i \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!/} \Gamma_i \vdash \widehat{k_i}\right)$.

$$\Gamma \vdash \{\Sigma_a * ((\varphi * \varphi_i) \wedge \Pi_i \wedge \Pi'_i)\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

We then follow the same reasoning as in the proof for our entailment system (Theorem 28), generating the following result for each premise.

$$\begin{aligned}\Gamma_i \vdash \{(\varphi * (\exists \vec{z_i}.\ \varphi_i {\wedge} \Pi'_i) * \Sigma_a) \wedge \Pi_i\}& \\ \vec{z_i} := ?\,\texttt{;}\ \mathsf{assume}(\Pi'_i)\,\texttt{;}\ \widehat{k_i} \blacktriangleright_{\underline{\mathrm{IVar}}} k&\end{aligned}$$

Note that each assumption now has a precondition of the form below

$$(\varphi * (\exists \vec{z_i}.\ \varphi_i \wedge \Pi'_i) * \Sigma_a) \wedge \Pi_i \tag{5.11}$$

Our goal is to show that the following holds, where $\widehat{k_b}$ is the branch in the instrumented continuation in the conclusion of the DEFL rule (which has the form branch ... end).

$$\Gamma \vdash \{(\varphi * d(\vec{e})) * \Sigma_a\}\ \widehat{k_b} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

As with entailment, we note that the precondition in the formula above is equivalent to the following.

$$\varphi * \left(\bigvee_i \Big(\Pi_i \wedge (\exists z_i.\ \Pi'_i \wedge \varphi_i)\Big)\right) * \Sigma_a$$

By commuting and re-associating terms, we can rewrite this such that it is equal to equation (5.11) for each i. In entailment, we then had that the soundness of the branch that we add follows from an n-ary version of the derived rule below.

$$\frac{Q \Rightarrow (Q_1 \wedge e_1) \vee (Q_2 \wedge e_2) \qquad \Gamma \vdash \{Q_1 \wedge e_1\}\ \widehat{k_1} \blacktriangleright_V k \qquad \Gamma \vdash \{Q_2 \wedge e_2\}\ \widehat{k_2} \blacktriangleright_V k}{\Gamma \vdash \{Q\}\ \mathsf{branch}\ e_1 \Rightarrow \widehat{k_1}, e_2 \Rightarrow \widehat{k_2}\ \mathsf{end} \blacktriangleright_V k}$$

This was the extent of the proof for this case in Theorem 28. For frame inference, one more step is necessary. We have to address the fact that the statements of valid instrumentation for our premises do not involve the same context. For this reason, we need the rule below.

$$\dfrac{\begin{array}{c} Q \Rightarrow (Q_1 \wedge e_1) \vee (Q_2 \wedge e_2) \\ \Gamma_1 \vdash \{Q_1 \wedge e_1\}\ \widehat{k_1} \blacktriangleright_V k \qquad \Gamma_2 \vdash \{Q_2 \wedge e_2\}\ \widehat{k_2} \blacktriangleright_V k \end{array}}{\Gamma_1 \cup \Gamma_2 \vdash \{Q\}\ \mathsf{branch}\ e_1 \Rightarrow \widehat{k_1}, e_2 \Rightarrow \widehat{k_2}\ \mathsf{end} \blacktriangleright_V k}\ \textsc{Inst-Branch}'$$

This can be derived from the previous rule (where the contexts were required to be the same) by making use of Lemma 12. Recall that $(\Gamma \cup \Gamma')(l) = \Gamma(l) \vee \Gamma'(l)$[3]. Since $\Gamma(l) \Rightarrow \Gamma(l) \vee \Gamma'(l)$ and $\Gamma'(l) \Rightarrow \Gamma(l) \vee \Gamma'(l)$ we can unify the contexts present in the premises of our desired inference rule above, obtaining the following derivation, which establishes this as a valid derived rule and completes the proof of soundness for this case.

$$\begin{tikzcd}
\text{Lem. 12}\ \dfrac{\Gamma_1 \vdash \{Q_1 \wedge e_1\}\ \widehat{k_1} \blacktriangleright_V k}{\Gamma_1 \cup \Gamma_2 \vdash \{Q_1 \wedge e_1\}\ \widehat{k_1} \blacktriangleright_V k} & \text{Lem. 12}\ \dfrac{\Gamma_1 \vdash \{Q_1 \wedge e_1\}\ \widehat{k_1} \blacktriangleright_V k}{\Gamma_1 \cup \Gamma_2 \vdash \{Q_1 \wedge e_1\}\ \widehat{k_1} \blacktriangleright_V k} \\
\dfrac{Q \Rightarrow (Q_1 \wedge e_1) \vee (Q_2 \wedge e_2) \qquad\qquad\qquad}{\Gamma_1 \cup \Gamma_2 \vdash \{Q\}\ \mathsf{branch}\ e_1 \Rightarrow \widehat{k_1}, e_2 \Rightarrow \widehat{k_2}\ \mathsf{end} \blacktriangleright_V k}\ \textsc{Inst-Branch}' \arrow[u] \arrow[ur] &
\end{tikzcd}$$

Proper Form We now show the proof for Theorem 30, which states that f_k is only called with inputs of the appropriate form. The proof is by induction on the derivation of

$$\Sigma_a \mathbin{[]} \varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \mathbin{/\!\!/} \Gamma \vdash \widehat{k}$$

For rules where φ' and Σ_a are identical in the premise and conclusion of the rule, our result follows immediately from the inductive hypothesis. This includes rules PROPEQL, NOTNULL, DISJOINT, INSTL, EXISTSL, and DEFL. For LEFTPUREFALSE there is nothing to prove, as f_k is not called in the derivation (this rule is an axiom that does not call f_k).

[3] Technically, contexts in this chapter map locations to sets of symbolic state formulas, whereas the contexts in Chapter 4 mapped locations to separation logic formulas. However, since we are interpreting sets of symbolic state formulas disjunctively, the equality given here in terms of formulas holds.

We now consider each of the other rules.

PtoMatches We have from our inductive hypothesis that f_k is only called with inputs of the form

$$(\varphi'' * (\Sigma_a * (e \mapsto \rho)))$$

for some φ'' such that $\varphi'' \Rightarrow \varphi' * \mathsf{true}$. We must show that f_k is only called with inputs of the form $(\varphi''' * \Sigma_a)$ such that $\varphi''' \Rightarrow \big(\varphi' * (e \mapsto \rho) * \mathsf{true}\big)$. We let $\varphi''' = \varphi'' * (e \mapsto \rho)$. To complete the proof, we must show $\big(\varphi'' * (e \mapsto \rho)\big) \Rightarrow \big(\varphi' * (e \mapsto \rho) * \mathsf{true}\big)$. This follows directly from our assumption $\varphi'' \Rightarrow \varphi' * \mathsf{true}$ and the fact that, in separation logic, if $p \Rightarrow q$ is valid, then so is $p * r \Rightarrow q * r$.

PredMatches The proof for this case is the same as for PtoMatches, but with $d(\vec{e})$ substituted for $e \mapsto \rho$.

PtoMatchesAny We have from our inductive hypothesis that f_k is only called with inputs of the form

$$(\varphi'' * (\Sigma_a * (e \mapsto \rho)))$$

for some φ'' such that $\varphi'' \Rightarrow \varphi' * \mathsf{true}$. We must show that f_k is only called with inputs of the form $(\varphi''' * \Sigma_a)$ such that $\varphi''' \Rightarrow \big(\varphi' * (e \mapsto \square) * \mathsf{true}\big)$. We let $\varphi''' = \varphi'' * (e \mapsto \rho)$. To complete the proof, we must then show $\big(\varphi'' * (e \mapsto \rho)\big) \Rightarrow \big(\varphi' * (e \mapsto \square) * \mathsf{true}\big)$. This follows directly from our assumption $\varphi'' \Rightarrow \varphi' * \mathsf{true}$ and the fact that $e \mapsto \rho$ implies $e \mapsto \square$.

ExistsR We have from our inductive hypothesis that f_k is only called with inputs of the form

$$(\varphi'' * \Sigma_a)$$

for some φ'' such that $\varphi'' \Rightarrow \varphi'[e/x] * \mathsf{true}$. We must show that f_k is only called with inputs of the form $\varphi''' * \Sigma$ such that $\varphi''' \Rightarrow \big(\exists x.\ \varphi' * \mathsf{true}\big)$. We let $\varphi''' = \varphi''$. Because $\varphi'[e/x] \Rightarrow \exists x.\ \varphi'$ we then have $\varphi''' \Rightarrow (\exists x.\ \varphi') * \mathsf{true}$ which is our goal.

RIGHTPURE This is the only axiom that calls f_k and thus is the base case for this proof. The argument passed to f_k is the following

$$\exists\vec{x}.\ (\Sigma_a * \Sigma) \wedge \Pi'$$

We must show that this has the form $\varphi'' * \Sigma_a$ where $\varphi'' \Rightarrow (\exists\vec{x}.\ \mathbf{emp} \wedge \Pi') * \mathsf{true}$. We let φ'' be $\exists\vec{x}.\ \Sigma \wedge \Pi'$. We then must show

$$(\exists\vec{x}.\ \Sigma \wedge \Pi') \Rightarrow (\exists\vec{x}.\ \mathbf{emp} \wedge \Pi') * \mathsf{true}$$

We first assume $(\exists\vec{x}.\ \Sigma \wedge \Pi')$. From this and the tautology $\Sigma \Rightarrow \mathsf{true}$, we have that $\exists\vec{x}.\ \mathsf{true} \wedge \Pi'$ holds. Since $\mathsf{true} \Leftrightarrow \mathsf{true} * \mathbf{emp}$ we have $\exists\vec{x}.\ (\mathsf{true} * \mathbf{emp}) \wedge \Pi'$. Since Π' is pure this implies $\exists\vec{x}.\ \mathsf{true} * (\mathbf{emp} \wedge \Pi')$. Applying commutativity of $*$ and moving true outside the scope of the existential quantifier then gives us our result.

Usage Example

We now provide an example designed to give some intuition into the use of frame inference in the construction of an instrumentation.

One main problem that we are introducing frame inference to address is the failure of post-conditions to match up with preconditions in general. Our `partialPost` function on page 217 requires the preconditions of commands that access a heap cell at x to explicitly contain a points-to predicate at x. Often, the precondition does not have this form, but can be shown to imply one which does. In such cases, having a method of proving this implication allows us to proceed with our program analysis.

Suppose we are instrumenting continuation k which is equal to $(x := x.\mathsf{next})\texttt{;}\,k'$. Further assume that we have a precondition of $ls(\underline{n}; x, \mathsf{nil}) \wedge x \neq \mathsf{nil}$. In order to apply `partialPost`, we need a precondition of the form $\exists\vec{y}.\ ((x \mapsto [\rho]) * \Sigma) \wedge \Pi$. We can then construct a frame inference query that produces an instrumentation starting from $ls(\underline{n}; x, \mathsf{nil}) \wedge x \neq \mathsf{nil}$ as follows.

Let f_k be the function below.

$$f_k \stackrel{\text{def}}{=} \lambda s_1.\ \texttt{instPost}(s_1, x := x.\mathsf{next}, \lambda s_2.\ \texttt{geninstCont}(\emptyset, s_2, k')))$$

Then the frame inference query that we want is the one below.

$$ls(\underline{n}; x, \mathsf{nil}) \wedge x \neq \mathsf{nil} \underset{\mathbf{S}}{\Longrightarrow}_{f_k} (x \mapsto \Box) \mathbin{/\!\!/} \Gamma \vdash \widehat{k}$$

This is an abbreviation for the query below, which initiates a proof search using the rules in Figures 5.10 and 5.11.

$$\mathbf{emp} \,[]\, ls(\underline{n}; x, \mathsf{nil}) \wedge x \neq \mathsf{nil} \underset{\mathbf{S}}{\Longrightarrow}_{f_k} (x \mapsto \Box) \mathbin{/\!\!/} \Gamma \vdash \widehat{k}$$

5.5.4 `exposeCellThenInst`

The function `exposeCellThenInst` provides the interface to frame inference in our implementation. The code for this function is given on the next page. The call `exposeCellThenInst`(φ, x, f_k) takes the following arguments.

- φ A symbolic state formula that gives the current precondition.
- x The address of the heap cell to be revealed.
- f_k The instrumentation generator to apply to the formula that results from showing that x is in the heap.

If `exposeCellThenInst` returns $\mathsf{Some}\big(\Gamma, \widehat{k}\big)$ then these must satisfy

$$\Gamma \vdash \{\varphi\}\, \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

This function issues a frame inference query with the pattern $x \mapsto \Box$ on the right in order to expose the heap cell at x. The sequent $\varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} x \mapsto \Box \mathbin{/\!\!/} \Gamma \vdash \widehat{k}$ will be derivable only if x can be shown to be in the heap. If the cell at x is indeed exposed, then f_k will be called with the resulting heap. This gives us a method of converting symbolic state formulae to the form expected by the `partialPost` function presented on page 217.

The soundness result for frame inference tells us that the following holds.

$$\Gamma \vdash \{\varphi\}\, \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

which is exactly what is required for `exposeCellThenInst` to satisfy its specification from Figure 5.7.

Function `exposeCellThenInst`(φ, x, f_k). Exposes the heap cell at x by attempting to prove an implication of the form $\varphi \Rightarrow (x \mapsto \Box) * \varphi'$ where the box represents any record expression. If this proof succeeds, then the instrumentation generator f_k is applied to the formula that results.

```
let $f'_k = \lambda(b, \varphi).\ f_k(\varphi)$ in
  Search for proof of $\varphi \underset{\mathbf{S}}{\Longrightarrow}_{f'_k} x \mapsto \Box \mathbin{/\!/} \Gamma \vdash \widehat{k}$. (The elements $\Gamma$ and $\widehat{k}$ are returned by the proof procedure if a proof is found. The others are provided as inputs.)
  if proof is found and proof procedure returns $\Gamma, \widehat{k}$ then
    return $\mathsf{Some}\big(\Gamma, \widehat{k}\big)$
  else
    return $\mathsf{None}$
```

5.6 Example

We now pause to present an example of the automated analysis we have developed thus far. We will consider the following inductive specification of a singly linked list.

$$\begin{aligned} &\mathit{ls}(\underline{n}; x, y) \ \texttt{<=>} \\ &\quad \underline{n} = 0 : \mathsf{let}\ [\,]\ \mathsf{satisfy}\ \mathsf{true}\ \mathsf{in}\ \mathbf{emp} \wedge x = y \\ &\mid\ \underline{n} > 0 : \mathsf{let}\ \underline{n}'\ \mathsf{satisfy}\ \underline{n} = \underline{n}' + 1\ \mathsf{in} \\ &\qquad\qquad\quad \exists z.\ (x \mapsto [\mathsf{next} : z]) * \mathit{ls}(\underline{n}'; z, y) \end{aligned}$$

And analyze the following program, which traverses a list of this form.

$$\begin{aligned} L_1 : \quad &\textcircled{1}\ \mathsf{branch}\ x \neq \mathsf{nil} \Rightarrow \textcircled{2}\ x := x.\mathsf{next}\texttt{;}\ \textcircled{3}\ \mathsf{goto}\ L_1, \\ &\qquad\qquad x = \mathsf{nil} \Rightarrow \textcircled{4}\ \mathsf{halt\ end} \end{aligned}$$

We will let $\varphi_0 = \mathit{ls}(\underline{n}; x, \mathsf{nil})$ and $\Gamma = \{(L_1, \varphi_0)\}$ and we will execute

$$\texttt{geninstCont}(\Gamma, \varphi_0, \textcircled{1})$$

Since we have not yet presented definitions of `abstract` and `branchAnnot`, we will adopt the following definitions for now, which trivially satisfy the specifications given in

Figure 5.7, but are not as useful as those we present later.

$$\begin{array}{rcl} \texttt{abstract}(\varphi) & = & (\varphi, \epsilon) \\ \texttt{branchAnnot}(\varphi, [e_1, \ldots, e_n]) & = & [\mathsf{true}, \ldots, \mathsf{true}] \end{array}$$

The first construct in our continuation is a branch, so the code for `geninstCont` on page 210 calls

$$\texttt{branchAnnot}(\varphi, [x \neq nil, x = \mathsf{nil}])$$

This returns $[\mathsf{true}, \mathsf{true}]$. Next, the function calls `geninstCont` recursively on ② and ④. The call to $\texttt{geninstCont}(\Gamma, \varphi_0 \wedge x = \mathsf{nil}, ④)$ returns $\mathsf{Some}\big(\Gamma, \mathsf{halt}\big)$. The call to $\texttt{geninstCont}(\Gamma, \varphi_0 \wedge x \neq \mathsf{nil}, ②)$ calls `instPost` in order to process $x := x.\mathsf{next}$. So we now have the partial instrumentation given below, where we elide portions that have not been generated yet and write the precondition at that point in braces. We also write dark circle numbers to indicate those control points that have already been considered by our algorithm.

$$\begin{array}{ll} L_1: & ❶\ \mathsf{branch}\ \ x \neq \mathsf{nil} \Rightarrow \mathsf{assume}(\mathsf{true})\,\texttt{;}\,\{ls(\underline{n}; x, \mathsf{nil}) \wedge x \neq \mathsf{nil}\}②\ldots, \mathsf{end} \\ & \qquad\qquad\quad x = \mathsf{nil} \Rightarrow \mathsf{assume}(\mathsf{true})\,\texttt{;}\,❹\ \mathsf{halt} \end{array}$$

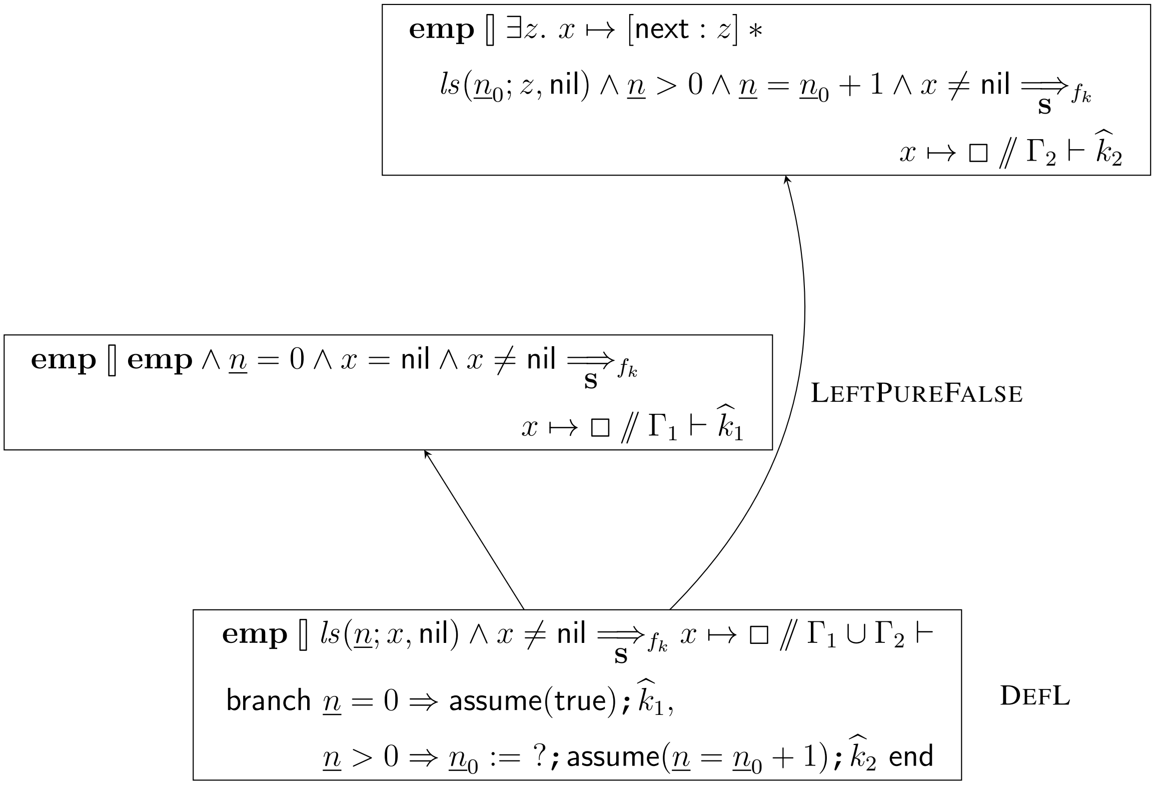
The `instPost` function notices that $x := x.\mathsf{next}$ is in $A[x]$—that is, it is a command that requires a memory cell at x to be present in the heap. Because of this, it calls frame inference to derive a proof of

$$ls(\underline{n}; x, \mathsf{nil}) \wedge x \neq \mathsf{nil} \underset{\mathbf{S}}{\Longrightarrow}_{f_k} x \mapsto \Box \mathbin{/\!\!/} \Gamma \vdash \widehat{k}$$

where the function f_k is the function that calls `partialPost` and then `geninstCont` on the post-condition to continue processing. Recall that the above is an abbreviation for the following sequent.

$$\mathbf{emp}\ [\!]\ ls(\underline{n}; x, \mathsf{nil}) \wedge x \neq \mathsf{nil} \underset{\mathbf{S}}{\Longrightarrow}_{f_k} x \mapsto \Box \mathbin{/\!\!/} \Gamma \vdash \widehat{k}$$

The first step of frame inference applies DEFL, obtaining the following start for the proof tree.


$\mathbf{emp} \mathbin{[]} \exists z.\ x \mapsto [\mathsf{next} : z] * ls(\underline{n}_0; z, \mathsf{nil}) \wedge \underline{n} > 0 \wedge \underline{n} = \underline{n}_0 + 1 \wedge x \neq \mathsf{nil} \underset{\mathbf{S}}{\Longrightarrow}_{f_k} x \mapsto \square \mathbin{/\!\!/} \Gamma_2 \vdash \widehat{k}_2$
$\mathbf{emp} \mathbin{[]} \mathbf{emp} \wedge \underline{n} = 0 \wedge x = \mathsf{nil} \wedge x \neq \mathsf{nil} \underset{\mathbf{S}}{\Longrightarrow}_{f_k} x \mapsto \square \mathbin{/\!\!/} \Gamma_1 \vdash \widehat{k}_1$
LEFTPUREFALSE
$\mathbf{emp} \mathbin{[]} ls(\underline{n}; x, \mathsf{nil}) \wedge x \neq \mathsf{nil} \underset{\mathbf{S}}{\Longrightarrow}_{f_k} x \mapsto \square \mathbin{/\!\!/} \Gamma_1 \cup \Gamma_2 \vdash$
branch $\underline{n} = 0 \Rightarrow$ assume(true); $\widehat{k}_1$,
$\underline{n} > 0 \Rightarrow \underline{n}_0 := ?$; assume($\underline{n} = \underline{n}_0 + 1$); $\widehat{k}_2$ end
DEFL

Clearly the sequents involved are far too long to display a full traditional proof tree here. Instead, we will present an abbreviated tree that labels each node with the inference rule applied at that point and also records the arguments used in any calls to f. We will write the information needed to reconstruct the full rule instance to the side of the rule name. For the matching rules, this will be the formula that is matched. For rules that instantiate variables, this will be the substitution. For DEFL, this will be the predicate instance expanded. The context and instrumented continuation that are returned by each rule are listed below it. We write Γ_1 and $\widehat{k}_1$ to refer to the context and continuation returned by the first (leftmost) child in the tree, $\Gamma_2, \widehat{k}_2$ to refer to the second, etc. Figure 5.12 gives the derivation tree.

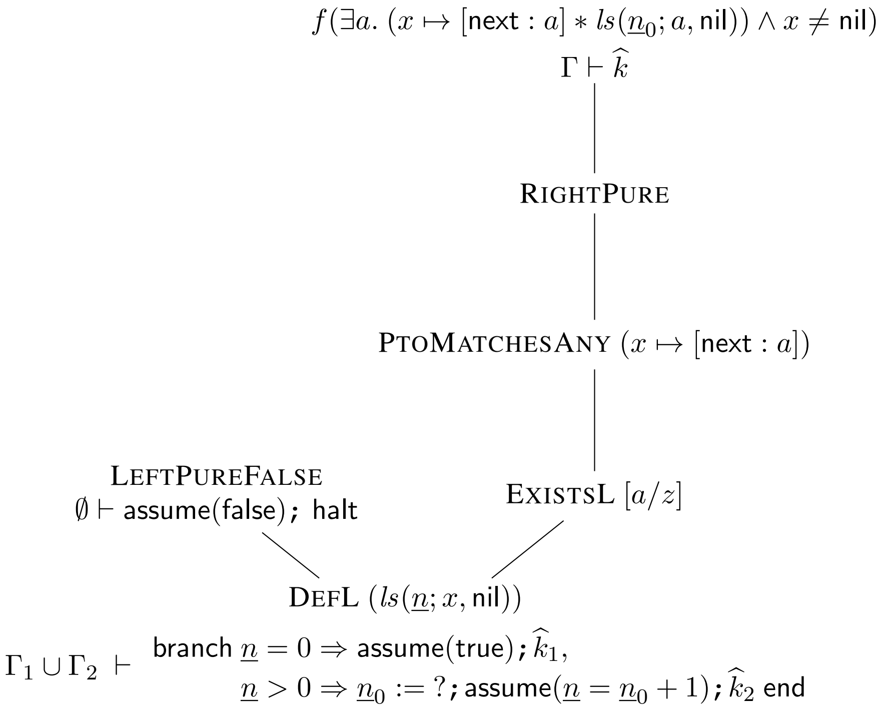

$f(\exists a.\ (x \mapsto [\mathsf{next} : a] * ls(\underline{n}_0; a, \mathsf{nil})) \land x \neq \mathsf{nil})$
$\Gamma \vdash \widehat{k}$
RIGHTPURE
PTOMATCHESANY $(x \mapsto [\mathsf{next} : a])$
LEFTPUREFALSE
$\emptyset \vdash$ assume(false); halt
EXISTSL $[a/z]$
DEFL $(ls(\underline{n}; x, \mathsf{nil}))$
$\Gamma_1 \cup \Gamma_2 \ \vdash$ branch $\underline{n} = 0 \Rightarrow$ assume(true); $\widehat{k}_1$,
$\underline{n} > 0 \Rightarrow \underline{n}_0 :=\ ?$; assume($\underline{n} = \underline{n}_0 + 1$); $\widehat{k}_2$ end

Figure 5.12: Proof for the frame inference query

$$ls(\underline{n}; x, \mathsf{nil}) \land x \neq \mathsf{nil} \underset{\mathbf{S}}{\Longrightarrow}_{f_k} x \mapsto \square \mathbin{/\!\!/} \Gamma \vdash \widehat{k}$$

We use $\Gamma_1, \widehat{k}_1$ to refer to the results from the left branch and $\Gamma_2, \widehat{k}_2$ to refer to the result from the right branch.

Combining this with what we had before, we have now built up the following partial continuation.

```
L_1 :   ❶ branch $x \neq \mathsf{nil} \Rightarrow$ assume(true);
                  branch $\underline{n} = 0 \Rightarrow$ assume(true); assume(false); halt,
                      $\underline{n} > 0 \Rightarrow \underline{n}_0 :=\ ?$; assume($\underline{n} = \underline{n}_0 + 1$);
                            $\{\exists a.\ (x \mapsto [\mathsf{next} : a] * ls(\underline{n}_0; a, \mathsf{nil}))$
                              $\land\ x \neq \mathsf{nil}\}$
                            ②... end
            $x = \mathsf{nil} \Rightarrow$ assume(true); ❹ halt end
```

We now execute `partialPost` to find the post-condition of the invariant at control location ②, reproduced below

$$\exists a.\ (x \mapsto [\mathsf{next} : a] * \mathit{ls}(\underline{n}_0; a, \mathsf{nil})) \wedge x \neq \mathsf{nil}$$

with respect to the command $x := x.\mathsf{next}$. This results in the formula below.

$$\exists a, x'.\ (x' \mapsto [\mathsf{next} : a] * \mathit{ls}(\underline{n}_0; a, \mathsf{nil})) \wedge x' \neq \mathsf{nil} \wedge x = a$$

If we perform some simplification, we obtain the formula below.

$$\exists x'.\ (x' \mapsto [\mathsf{next} : x]) * \mathit{ls}(\underline{n}_0; x, \mathsf{nil}) \tag{5.12}$$

The next command encountered is the goto L_1 command, which causes `geninstCont` to compare the current state against the invariants that have been collected in Γ. The only invariant currently in Γ and associated with location L_1 is the following.

$$\mathit{ls}(\underline{n}; x, \mathsf{nil})$$

This is not implied by (5.12) because, while we can match $\mathit{ls}(\underline{n}_0; x, \mathsf{nil})$ against $\mathit{ls}(\underline{n}; x, \mathsf{nil})$ by inserting the instrumentation command $\underline{n} := \underline{n}_0$, we cannot match the portion of the heap described by $x' \mapsto [\mathsf{next} : x]$. The current formula thus represents states not satisfied by the previous formula at L_1 and `geninstCont` indicates that we should apply `abstract`, add the result to Γ, and then continue processing from this new state.

Here we see the problem with the simple version of `abstract` we defined earlier. With `abstract` defined to be the identity function, we will never converge on a finite set of invariants associated with L_1 that describe all the reachable states of this program.

To show that this is the case, we list the next two invariants that the analysis will discover associated with L_1.

$$\exists x', x_2.\ (x' \mapsto [\mathsf{next} : x_2]) * (x_2 \mapsto [\mathsf{next} : x]) * \mathit{ls}(\underline{n}_2; x, \mathsf{nil})$$

$$\exists x', x_2, x_3.\ (x' \mapsto [\mathsf{next} : x_2]) * (x_2 \mapsto [\mathsf{next} : x_3]) * (x_3 \mapsto [\mathsf{next} : x]) * \mathit{ls}(\underline{n}_3; x, \mathsf{nil})$$

The symbolic state formulae that we generate continue to contain more and more points-to predicates that are not part of the list from x to nil.

This highlights the importance of the `abstract` function. Without it, the algorithm does not terminate. But with a well-chosen `abstract`, as we will see in the next section, the algorithm is able to converge on fixed-points for many programs.

5.7 Abstraction

The final component necessary before we can present a full example run of the algorithm, is the framework for performing *abstraction*. This is similar to the summarization step in TVLA Sagiv et al. [2002] and corresponds to the *abstraction function* used in abstract interpretation Cousot and Cousot [1977].

The motivation for abstraction is that if we only perform post-condition computation and unroll inductive predicates on the left, we will never converge on a finite set of invariants, as we saw in the previous section. Abstraction solves this problem by occasionally intentionally forgetting information about our current symbolic state formula in order to allow it to cover more concrete states. The term *abstraction* refers to the fact that this operation results in a more abstract (weaker) formula.

To give a simple example, consider one of the states we generated when looking at the example in the previous section.

$$\exists x'.\ (x' \mapsto [\mathsf{next} : x]) * \mathit{ls}(\underline{n}_0; x, \mathsf{nil})$$

The formula $x' \mapsto [\mathsf{next} : x]$ describes a list segment of length one. That is, every concrete stack and heap pair which satisfy $x' \mapsto [\mathsf{next} : x]$ also satisfy $\mathit{ls}(1; x', x)$. We are thus free to apply STRENGTHENING to switch the current state formula from $\exists x'.\ (x' \mapsto [\mathsf{next} : x]) * \mathit{ls}(\underline{n}_0; x, \mathsf{nil})$ to $\exists x'.\ \mathit{ls}(1; x', x) * \mathit{ls}(\underline{n}_0; x, \mathsf{nil})$ before storing the state in Γ. This is what `abstract` will do—return a different formula that is implied by the formula supplied as input.

The transformation just described is not enough, however, to cause the analysis to terminate. We will simply obtain the sequence of states

$$\exists x'.\ ls(1; x', x) * ls(\underline{n}_0; x, \mathsf{nil})$$
$$\exists x'.\ ls(2; x', x) * ls(\underline{n}_0; x, \mathsf{nil})$$
$$\exists x'.\ ls(3; x', x) * ls(\underline{n}_0; x, \mathsf{nil})$$
$$\vdots$$

We need to forget the length as well before we can obtain a formula weak enough to describe all reachable states. One way to do this would be to existentially quantify the length, obtaining the invariant

$$\exists n, x'.\ ls(n; x', x) * ls(\underline{n}_0; x, \mathsf{nil})$$

However, we can also use an instrumentation variable to capture the fact that the length is changing. This provides a more precise abstraction, as we will record instrumentation commands describing exactly how the changes to the length occur (in this case, we will record that the length of this segment increases by one each time we reach L_1).

Because we must describe exactly how an instrumentation variable is updated, this method requires more care than the use of an existential variable. However, as we will see, all the information we need is already present in the form of our inductive specifications.

5.7.1 Abstraction Patterns

We will derive formulae termed *abstraction patterns* from the cases of our inductive specifications. These describe exactly how to replace some portion of the state formula with an instance of an inductively specified predicate.

We will again take the singly-linked list specification as our example.

$$\begin{aligned}
&ls(\underline{n}; x, y)\ \texttt{<=>}\\
&\qquad \underline{n} = 0 : \text{let } [\,] \text{ satisfy } \mathsf{true} \text{ in } \mathbf{emp} \wedge x = y\\
&\quad\ \mid\ \underline{n} > 0 : \text{let } \underline{n'} \text{ satisfy } \underline{n} = \underline{n'} + 1 \text{ in}\\
&\qquad\qquad\qquad \exists z.\ (x \mapsto [\mathsf{next} : z]) * ls(\underline{n'}; z, y)
\end{aligned}$$

We first consider the $\underline{n} > 0$ case. Reading the equivalence from right to left, this states that if the heap contains $x \mapsto [\mathsf{next} : z]$ for some z and separately contains $\mathit{ls}(\underline{n}'; z, y)$ for the same z, then this can be viewed as $\mathit{ls}(\underline{n}; x, y)$ for some $\underline{n}$ such that $\underline{n} = \underline{n}' + 1$. This allows us to replace $(x \mapsto [\mathsf{next} : z]) * \mathit{ls}(\underline{n}'; z, y)$ with $\mathit{ls}(\underline{n}; x, y)$ provided we also update the instrumentation variables appropriately. The main issue in terms of implementation of such a replacement method is how to perform the initial matching. That is, how do we determine the instantiation of bound variables in the inductive specification that results in an applicable instance of the rule. Our matching will be guided by the spatial formulae present in the specification and in the current state.

For the example of the non-empty case of the singly-linked list predicate, we want to search for a sub-formula of the current state—call it φ—that has the form below.

$$(e_1 \mapsto [\mathsf{next} : e_2]) * \mathit{ls}(e_4; e_2, e_3)$$

Once we have found such a sub-formula, we can replace it with $\mathit{ls}(\underline{n}; e_1, e_3)$ provided that the following *pattern condition* holds

$$\varphi \Rightarrow \exists \underline{n}.\ \underline{n} = e_4 + 1 \wedge \underline{n} > 0$$

The reason for this check is that we could have a predicate such as the one below, which describes lists of length less than 5.

$$\begin{aligned}
&\mathit{ls}(\underline{n}; x, y) \ \texttt{<=>} \\
&\qquad \underline{n} = 0 : \text{let } [\,] \text{ satisfy } \mathsf{true} \text{ in } \mathbf{emp} \wedge x = y \\
&\quad \mid\ \underline{n} > 0 \wedge \underline{n} < 5 : \text{let } \underline{n}' \text{ satisfy } \underline{n} = \underline{n}' + 1 \text{ in} \\
&\qquad\qquad\qquad \exists z.\ (x \mapsto [\mathsf{next} : z]) * \mathit{ls}(\underline{n}'; z, y)
\end{aligned}$$

Such a specification cannot always be applied right-to-left even if the spatial portion of one of the cases can be matched. In practice, we have never needed to work with such a specification. All the specifications we have written while running our experiments have the property that the check above is always true. We will state the theory in terms of the general case, which requires this check. But it is useful to avoid it whenever possible in the implementation, as proving pure implications involving existential quantification on the right can be a slow process for many theorem provers.

We now consider the general case. Recall that a case of a specification has the form below

$$\Pi : \text{let } \underline{\vec{z}} \text{ satisfy } \Pi' \text{ in } \exists \vec{x_1}.\ \Sigma \wedge \Pi''$$

and is abbreviated as $C(\underline{\vec{x}}; \vec{y})$, where $\underline{\vec{x}}$ is the list of instrumentation parameters for the definition and $\vec{y}$ is the list of non-instrumentation parameters. The meaning of this case as a separation logic formula is the following

$$\Pi \wedge \exists \underline{\vec{z}}.\ (\Pi' \wedge \varphi)$$

which we write $\lceil C(\underline{\vec{x}}; \vec{y}) \rceil$.

When matching such a case against a symbolic state, most of the variables will be interpreted existentially, as they were in our example above. To see why, consider the reasoning process we are trying to establish in executing this replacement. For some case $C(\underline{\vec{x}}; \vec{y})$ of an inductive predicate $d(\underline{x}; y)$, and some symbolic state formula φ, we want to show the following.

$$\varphi \Rightarrow (\varphi' * \lceil C(\underline{\vec{e_1}}; \vec{e_2}) \rceil) \Rightarrow (\varphi' * d(\underline{\vec{e_1}}; \vec{e_2})) \tag{5.13}$$

In the first implication, $C(\underline{\vec{e_1}}; \vec{e_2})$ appears on the right, so we get to choose terms not just for the parameters, but also for any existentially quantified variables in the body of the case. This includes $\vec{x_1}$ and also $\underline{\vec{z}}$, as these appear existentially quantified in the representation of the case as a separation logic formula.

Though these variables are all existential in nature, they do serve different roles, motivated by our desire to use this rewriting process to produce formulae that are more likely to be invariants across multiple iterations of loops. As we saw with the list example, where we obtained a list of length 1, then length 2, then 3, etc., the instrumentation parameters $\underline{x}$ can interfere with the discovery of a loop invariant. Furthermore, it is difficult to find the list of expressions $\underline{\vec{e_1}}$ that witness the validity of the implication in (5.13), as $\underline{\vec{e_1}}$ may be an arithmetic expression not occurring in φ.

To remedy both these issues, we instead use the following line of reasoning.

$$\varphi \Rightarrow \exists \underline{\vec{x_1}}.\ (\varphi' * \lceil C(\underline{\vec{x_1}}; \vec{e_2}) \rceil) \Rightarrow \exists \underline{\vec{x_1}}.\ (\varphi' * d(\underline{\vec{x_1}}; \vec{e_2}))$$

We then insert the instrumentation command $\underline{\vec{x_1}} := ?$ to eliminate the existential on $\underline{\vec{x_1}}$. As we will see when we present the details, we also want to record at this point some assumption linking $\underline{\vec{x_1}}$ to other instrumentation variables. Following this line of reasoning ensures that the symbolic state formulae generated by abstraction always contain variables in the instrumentation parameter positions. This will make it easier to use the INSTL rule in our frame inference system to find instrumentation commands that allow us to re-establish a previously discovered invariant.

Another issue we must take care to avoid is the production of a formula that is too weak to be useful in further analysis of the program. To see an example of this, consider the invariant we obtained at L_1 after a single pass of analysis of our example list traversal program. We had

$$\exists x'.\ (x' \mapsto [\mathsf{next} : x] * \mathit{ls}(\underline{n}_0; x, \mathsf{nil}))$$

We noted previously that this formula implies

$$\exists x'.\ \mathit{ls}(1; x', x) * \mathit{ls}(\underline{n}_0; x, \mathsf{nil})$$

However it is also implies

$$\exists x'.\ \mathit{ls}(\underline{n}_0 + 1, x', \mathsf{nil})$$

But pushing this formula through the analysis will quickly lead us to trouble. The formula does not say anything about x, and so when we next try to execute $x := x.\mathsf{next}$ we are unable to show that x exists in the heap.

The reason we lost track of x is that we matched x to a variable that did not occur in the parameter list of the predicate. When we replace some piece of the formula representing the body of a case with an instance of an inductive predicate, we only retain spatial information about expressions occurring as parameters of that definition. In [Magill et al., 2006] we introduced a condition on abstraction rewrites that avoids this case. If we want to replace a piece of heap with an inductive predicate instance using a case of the form below

$$\Pi : \text{let } \underline{\vec{z}} \text{ satisfy } \Pi' \text{ in } \exists \vec{x_1}.\ \Sigma \wedge \Pi''$$

the expressions corresponding to $\vec{x_1}$ must not contain program variables. Distefano et al. [2006] present a stronger condition that also requires that variables in the expressions corresponding to $\vec{x_1}$ must not appear elsewhere in the spatial portion of the state. This stronger condition is important in more complicated sharing patterns. Consider the symbolic state below.

$$\exists z.\ ls(\underline{n}_1; x, z) * ls(\underline{n}_2; y, z) * ls(\underline{n}_3; z, \mathsf{nil})$$

Suppose we had a specification like the one below

$ls(\underline{n}; x, y)$ <=>

true : let $\underline{n}_1, \underline{n}_2$ satisfy $\underline{n} = \underline{n}_1 + \underline{n}_2$ in

$\exists z.\ ls(\underline{n}_1; x, z) * ls(\underline{n}_2; z, y)$

The weaker condition would then allow us to replace $ls(\underline{n}_2; y, z) * ls(\underline{n}_3; z, \mathsf{nil})$ with $ls(\underline{n}_2 + \underline{n}_3; y, \mathsf{nil})$ obtaining

$$\exists z.\ ls(\underline{n}_1; x, z) * ls(\underline{n}_2 + \underline{n}_3; y, \mathsf{nil})$$

This formula loses the information about x and y eventually reaching the same heap cell. This does not affect soundness, but would cause problems when, for example, traversing the list at x, as we would be unable to show memory safety beyond the point where x reaches z. The stronger condition would prevent us from combining these lists since z, the variable that is disappearing, occurs in $ls(\underline{n}_1; x, z)$, which does not participate in the replacement. We use the stronger condition in the presentation here and in our implementation.

Now that the motivation for the various checks is clear, we will present the general form of an abstraction pattern. The pattern will have the format below.

$$[\vec{v}]\ (\Sigma)\ \overset{\Pi}{\underset{\Pi'}{-\mathrm{PAT}\!\rightarrow}}\ (\Sigma')\ [\underline{\vec{x}}]$$

The variables in $\vec{v}$ can be instantiated with expressions when matching the pattern. The formula Σ gives the spatial formula that should be matched. The formula Π gives the pattern condition that must hold for the rewrite to be applicable. The variables $\underline{\vec{x}}$ are the new instrumentation variables that will be introduced, and the formula Π' gives the relationship between the new instrumentation variables and the old instrumentation variables present in Σ. The formula Σ' is the replacement for the spatial formula Σ. The variables $\vec{v}$ and $\underline{\vec{x}}$

are considered bound. We derive such a pattern from a case of an inductive specification as follows.

Definition 34. *Let $C(\underline{\vec{x}};\vec{y})$ be a case of an inductive specification of predicate d and suppose $C(\underline{\vec{x}};\vec{y})$ has the following form, where the variables $\underline{\vec{z}}, \vec{x_1}, \underline{\vec{x}}$, and $\vec{y}$ are all distinct.*

$$\Pi \;:\; \text{let } \underline{\vec{z}} \text{ satisfy } \Pi' \text{ in } \exists \vec{x_1}.\, \Sigma \wedge \Pi''$$

Then the ***abstraction pattern associated with*** *$C(\underline{\vec{x}};\vec{y})$ is*

$$[\vec{x_1}, \underline{\vec{z}}, \vec{y}]\ \big(\Sigma\big)\ \overset{\Pi\wedge\Pi'\wedge\Pi''}{\underset{\Pi'}{-\textsc{pat}\!\rightarrow}}\ \big(d(\underline{\vec{x}};\vec{y})\big)\ [\underline{\vec{x}}]$$

We expect patterns to obey the following soundness criterion.

Definition 35. *A pattern $[\vec{x}]\ \big(\Sigma\big)\ \overset{\Pi}{\underset{\Pi'}{-\textsc{pat}\!\rightarrow}}\ \big(\Sigma'\big)\ [\underline{\vec{y}}]$ is* ***sound*** *iff $\vec{x}$ and $\underline{\vec{y}}$ are all distinct, $\underline{y} \cap fv(\Sigma) = \emptyset$, and*

$$\forall \vec{x}.\, \Sigma \wedge (\exists \underline{\vec{y}}.\, \Pi) \Rightarrow \exists \underline{\vec{y}}.\, \Sigma' \wedge \Pi'$$

We then have the following theorem regarding our method for translating cases to patterns.

Theorem 31. *The method given as Definition 34 for converting a case of an inductive specification to an abstraction pattern is sound.*

Proof. The condition on distinction of the variables and the new instrumentation variables being not free in Σ follow from the same conditions on the syntax of our inductive specifications (see Figure 5.3).

For the main soundness condition, recall that an inductive specification

$$d(\underline{\vec{x}};\vec{y}) = C_1 \mid \ldots \mid C_n$$

is interpreted as the separation logic formula

$$\forall \underline{\vec{x}}, \vec{y}.\, d(\underline{\vec{x}}, \vec{y}) \Leftrightarrow \lceil C_1 \rceil \vee \ldots \vee \lceil C_n \rceil$$

This implies

$$\forall \underline{\vec{x}}, \vec{y}.\ \lceil C_i \rceil \Rightarrow d(\underline{\vec{x}}, \vec{y})$$

And this is the formula on which we will base the soundness argument.

Instantiating this with the particular C_i from Definition 34 we obtain

$$\forall \underline{\vec{x}}, \vec{y}.\ \big(\Pi \wedge \exists \underline{\vec{z}}.\ \Pi' \wedge \exists \vec{x_1}.\ \Sigma \wedge \Pi''\big) \Rightarrow d(\underline{\vec{x}}, \vec{y})$$

The restrictions on $\mathit{fv}(\Pi)$ and $\mathit{fv}(\Pi')$ in Figure 5.3 on page 193 give us that $\underline{\vec{z}} \cap \mathit{fv}(\Pi) = \emptyset$ and $\vec{x_1} \cap \mathit{fv}(\Pi, \Pi') = \emptyset$. This lets us rewrite the above as

$$\forall \underline{\vec{x}}, \vec{y}.\ \big(\exists \underline{\vec{z}}, \vec{x_1}.\ \Pi \wedge \Pi' \wedge (\Sigma \wedge \Pi'')\big) \Rightarrow d(\underline{\vec{x}}, \vec{y}) \tag{5.14}$$

This implication is available for use since it follows from one of the inductive specifications and all reasoning is done under the assumption that the inductive specifications hold.

To show soundness of the abstraction pattern, we must show the following.

$$\forall \vec{x_1}, \underline{\vec{z}}, \vec{y}.\ \Sigma \wedge (\exists \underline{\vec{x}}.\ \Pi \wedge \Pi' \wedge \Pi'') \Rightarrow \exists \underline{\vec{x}}.\ d(\underline{\vec{x}}; \vec{y}) \wedge \Pi'$$

We consider some arbitrary $\vec{x_1}, \underline{\vec{z}}, \vec{y}$ and assume $\Sigma \wedge (\exists \underline{\vec{x}}.\ \Pi \wedge \Pi' \wedge \Pi'')$. Since $\underline{\vec{x}} \cap \mathit{fv}(\Sigma) = \emptyset$ we can move the quantifier on $\underline{\vec{x}}$ to the outside, obtaining

$$\exists \underline{\vec{x}}.\ \Sigma \wedge (\Pi \wedge \Pi' \wedge \Pi'')$$

Eliminating the existential quantifier on $\underline{\vec{x}}$ and applying (5.14), then gives us.

$$d(\underline{\vec{x}}, \vec{y})$$

We already have Π', so we can obtain

$$d(\underline{\vec{x}}, \vec{y}) \wedge \Pi'$$

Then we re-introduce the existential quantifier on $\underline{\vec{x}}$, obtaining

$$\exists \underline{\vec{x}}.\ d(\underline{\vec{x}}, \vec{y}) \wedge \Pi'$$

which is our goal. □

5.7.2 Empty Patterns

In the discussion above, we concentrated on patterns that arose from the non-empty cases of our inductive specifications. Patterns based on empty cases pose a problem for automation because the spatial formula $\mathbf{emp}$ can be found in any symbolic state. Thus, patterns derived from empty cases would always be applicable. As a result, we do not generate patterns from empty cases. However, we need to include some sort of pattern derived from the base case or we will never be able to introduce instances of inductive predicates. Consider a routine that creates a linked list. We will get states like the following

$$x \mapsto [\mathsf{next} : \mathsf{nil}]$$
$$\exists x_1.\, x \mapsto [\mathsf{next} : x_1] * x_1 \mapsto [\mathsf{next} : \mathsf{nil}]$$
$$\exists x_1, x_2.\, x \mapsto [\mathsf{next} : x_2] * x_2 \mapsto [\mathsf{next} : x_1] * x_1 \mapsto [\mathsf{next} : \mathsf{nil}]$$

and with no way to introduce an instance of the list predicate, we will never find a finite description of all these states.

One solution is to have the user provide a creation pattern for each data structure. For example, for a linked list, they could provide

$$[x, y]\ \big(x \mapsto [\mathsf{next} : y]\big)\ \overset{\mathsf{true}}{\underset{\underline{k}=1}{-\!\text{PAT}\!\rightarrow}}\ \big(ls(\underline{k}; x, y)\big)\ [\underline{k}]$$

However such patterns can also be generated automatically by expanding inductive predicates repeatedly. For example, suppose we take the doubly-linked list definition below.

$$\begin{aligned}
&\mathrm{dll}(\underline{k};\, p, \mathit{first}, \mathit{last}, n)\ \ \texttt{<=>}\\
&\qquad \underline{k} = 0 : \text{let } [\,] \text{ satisfy } \mathsf{true} \text{ in } \mathbf{emp} \wedge \mathit{first} = n \wedge \mathit{last} = p\\
&\quad \mid\ \underline{k} > 0 : \text{let } \underline{k}' \text{ satisfy } \underline{k} = \underline{k}' + 1 \text{ in}\\
&\qquad\qquad\qquad \exists z.\ (\mathit{first} \mapsto [\mathsf{prev} : p, \mathsf{next} : z]) * \mathrm{dll}(\underline{k}'; \mathit{first}, z, \mathit{last}, n))
\end{aligned}$$

We can expand the predicate $\mathrm{dll}(\underline{k}; a, b, c, d)$ once using the non-empty case, obtaining

$$\begin{aligned}
&\underline{k} > 0 \wedge \exists \underline{k}'.\ \underline{k} = \underline{k}' + 1 \wedge\\
&\qquad\qquad \exists z.\ (b \mapsto [\mathsf{prev} : a, \mathsf{next} : z]) * \mathrm{dll}(\underline{k}'; b, z, c, d)
\end{aligned}$$

and then expand $\mathrm{dll}(\underline{k}'; b, z, c, d)$ using the empty case, obtaining

$$\begin{aligned}\underline{k} > 0 \wedge \exists \underline{k}'.\ \underline{k} = \underline{k}' + 1 \wedge \\ \exists z.\ (b \mapsto [\mathsf{prev} : a, \mathsf{next} : z]) \\ \wedge\ (\underline{k}' = 0 \wedge b = c \wedge z = d)\end{aligned}$$

We now have a description of a list segment that contains no inductive instances of the dll predicate but describes a non-empty heap. We can translate this into the following creation pattern.

$$[a, b, c, d, z]\ \big(b \mapsto [\mathsf{prev} : a, \mathsf{next} : z]\big)\ \underset{(\underline{k}=\underline{k}'+1)\wedge(\underline{k}'=0)}{\overset{(\underline{k}=\underline{k}'+1)\wedge(\underline{k}'=0\wedge b=c\wedge z=d)}{-\text{PAT}\!\!\rightarrow}}\ \big(\mathrm{dll}(\underline{k}; a, b, c, d)\big)\ [\underline{k}, \underline{k}']$$

Now suppose we are faced with a state such as the following.

$$x \mapsto [\mathsf{prev} : \mathsf{nil}, \mathsf{next} : y]$$

We can apply the pattern above by using the substitution $a \rightarrow \mathsf{nil}, b \rightarrow x, c \rightarrow x, d \rightarrow y, z \rightarrow y$. To make the pattern more useful for automation, it helps to eliminate the variable z and propagate the equality $b = c$. Propagating the equality $\underline{k}' = 0$ is also helpful as this results in fewer instrumentation variables. Applying these simplifications leaves us with the pattern below.

$$[a, b, d]\ \big(b \mapsto [\mathsf{prev} : a, \mathsf{next} : d]\big)\ \underset{\underline{k}=1}{\overset{\underline{k}=1}{-\text{PAT}\!\!\rightarrow}}\ \big(\mathrm{dll}(\underline{k}; a, b, b, d)\big)\ [\underline{k}]$$

The pattern condition in this case is equivalent to true (soundness for abstraction patterns states that $\exists \underline{k}.\ \underline{k} = 1$ must hold in this case, but this is a tautology). This enables us to simplify the pattern even further.

$$[a, b, d]\ \big(b \mapsto [\mathsf{prev} : a, \mathsf{next} : d]\big)\ \underset{\underline{k}=1}{\overset{\mathsf{true}}{-\text{PAT}\!\!\rightarrow}}\ \big(\mathrm{dll}(\underline{k}; a, b, b, d)\big)\ [\underline{k}]$$

Our implementation attempts to discover when pattern conditions are tautologies and apply this simplification, as avoiding the theorem proving call associated with checking the pattern condition each time the pattern is applied significantly decreases execution time.

5.7.3 Applying Abstraction Patterns

Now that we have shown how to derive abstraction patterns from inductive predicate specifications, we will show how these patterns are used to abstract a symbolic state formula. In Figure 5.13 we define a relation with syntax $\varphi \xrightarrow[\mathcal{A}]{\text{ABS}} \langle \varphi' \mid \mathbf{c} \rangle$. This relation takes a symbolic state formula φ to a pair consisting of a weaker formula φ' and $\mathbf{c}$, the sequence of instrumentation commands necessary to generate φ' from φ (the empty command list ϵ is used if φ' follows from φ by STRENGTHENING). The rules are parametrized by the set of abstraction patterns $\mathcal{A}$. Note that the side condition of the first rule can always be satisfied by renaming bound variables, as the variables $\underline{\vec{y}}$ are bound in the abstraction pattern. We show on page 275 the code for `abstract`, which uses the relation just described.

The formal specification of

$$\varphi \xrightarrow[\mathcal{A}]{\text{ABS}} \langle \varphi' \mid \mathbf{c} \rangle$$

is that this should hold only if for all $\Gamma, \widehat{k}, k$,

$$\Gamma \vdash \{\varphi'\}\ \widehat{k} \blacktriangleright_{\underline{\text{IVar}}} k$$

implies

$$\Gamma \vdash \{\varphi\}\ (\mathbf{c} \mathbin{;} \widehat{k}) \blacktriangleright_{\underline{\text{IVar}}} k$$

First Rule

The first rule in Figure 5.13 has a number of premises. We go through them each here, explaining their function. First we present a guide to the notation in the figure, using a linked list example. Below is an abstraction pattern that replaces two list-structured heap cells with an instance of the list predicate.

$$[x, y, z]\ \big(x \mapsto [\mathsf{next} : y] * y \mapsto [\mathsf{next} : z]\big) \xrightarrow[\underline{k}=2]{\overset{\mathsf{true}}{\text{PAT}}} \big(ls(\underline{k}; x, z)\big)\ [\underline{k}]$$

We will show how to apply this pattern to the symbolic state below (and several variations on this state).

$$\varphi_0 \overset{\text{def}}{=} \exists b.\ a \mapsto [\mathsf{next} : b] * b \mapsto [\mathsf{next} : \mathsf{nil}] * c \mapsto [\mathsf{next} : b] \wedge g > 0$$

We now describe each meta-variable present in the first rule in Figure 5.13.

φ The symbolic state formula that is being abstracted. For our example, this is φ_0, defined above.

Σ The left-hand side of the rewrite rule. Specifies the pattern to search φ for. In our example, this is $x \mapsto [\mathsf{next} : y] * y \mapsto [\mathsf{next} : z]$.

Σ' The right-hand side of the rewrite rule. Specifies the replacement for Σ. In our example, this is $ls(\underline{k}; x, z)$.

$\vec{x}$ The list of variables in the pattern that can be instantiated to expressions. In our example this is x, y, z. This can also include instrumentation variables if these are available for replacement.

σ The substitution that makes some portion of φ match Σ. Its domain is $\vec{x}$. In our example, this substitution will be $x \rightarrow a, y \rightarrow b, z \rightarrow \mathsf{nil}$ (other matchings are also possible—the abstraction process is non-deterministic and any matching pattern can be chosen and applied without affecting soundness).

Σ_0 The spatial portion of φ not matched by the pattern. This is $c \mapsto [\mathsf{next} : b]$ in our example.

Π_0 This is the pure portion of φ. In our example this is $g > 0$.

$\vec{x}_0$ The list of quantified variables in φ. In our example, this is the singleton b.

Π The condition that must hold in order for the replacement to occur. This is in addition to the premises on free variables that occur as preconditions in the first abstraction rule. In our example, this is $\mathtt{true}$.

$\underline{\vec{y}}$ The list of new instrumentation variables that are introduced by this pattern. In our example, this is $\underline{k}$.

Π' The relation between instrumentation variables in Σ and the new variables $\underline{\vec{y}}$. In our example this is $\underline{k} = 2$.

We now discuss each premise of the first rule in Figure 5.13.

condition$\big(\big(fv(\sigma(\Sigma)) - fv(\sigma(\Sigma'))\big) \subseteq \vec{x}_0\big)$

The difference $fv(\sigma(\Sigma)) - fv(\sigma(\Sigma'))$ gives the set of free variables that disappear from the formula when applying the patten. In our example, the difference evaluates to b, indicating that by combining $a \mapsto [\mathsf{next} : b] * b \mapsto [\mathsf{next} : \mathsf{nil}]$ into the predicate instance $ls(\underline{k}; x, z)$, we lose track of where b is pointing. The $\subseteq \vec{x}_0$ portion of this check ensures that the variables that are disappearing are existentially quantified. We want to avoid having non-quantified variables disappearing as these correspond to program variables, which may be dereferenced by later commands. In our example, this check passes, since b is quantified.

condition($\big(fv(\sigma(\Sigma)) - fv(\sigma(\Sigma'))\big) \cap fv(\Sigma_0) = \emptyset$)

This condition checks that the variables disappearing do not appear free in the portion of φ that is not participating in the replacement. In our example, this check fails, since b occurs in the predicate $c \mapsto [\mathsf{next} : b]$. We want to avoid losing track of such shared points of reference, as they can also later be accessed by heap commands. Suppose we were to perform our example replacement in spite of this check failing. Then we would obtain $ls(\underline{k}; x, \mathsf{nil}) * c \mapsto [\mathsf{next} : b]$. In such a state, if we execute the commands $v := c.\mathsf{next}$; $v := v.\mathsf{next}$ we will be unable to show that the second heap lookup is safe because we have lost track of the fact that b is in the middle of the two-element list at x.

In order to allow this check to pass and continue examining the other conditions, we will change our example state to the following, which changes the value of the next field of c so that it no longer points into the list.

$$\varphi_0 \stackrel{\text{def}}{=} \exists b.\ a \mapsto [\mathsf{next} : b] * b \mapsto [\mathsf{next} : \mathsf{nil}] * c \mapsto [\mathsf{next} : \mathsf{nil}] \wedge g > 0$$

condition($dom(\sigma) = \vec{x}$)

This condition simply checks that we are only performing substitutions on variables that are bound in the pattern.

condition($\varphi = \exists \vec{x}_0.\ (\Sigma_0 * \sigma(\Sigma)) \wedge \Pi_0$)

This premise separates φ into the portion that satisfies the pattern, $\sigma(\Sigma)$, and the rest, Σ_0 and Π_0. In our example, $a \mapsto [\mathsf{next} : b] * b \mapsto [\mathsf{next} : \mathsf{nil}]$ corresponds to $\sigma(\Sigma)$.

condition($\varphi \Rightarrow \exists \underline{\vec{y}}.\ \sigma(\Pi)$)

This premise checks that the symbolic state being rewritten satisfies the pattern condition Π. In our example, Π is true, so there is nothing to check here. The predicates we have encountered in our experiments have all had conditions of true. However, it is easy to construct examples whose abstraction rules require this check to be performed. An example of such a predicate is given on page 260.

condition($\left([\vec{x}]\ (\Sigma)\ \underset{\Pi'}{\overset{\Pi}{-\text{PAT}\rightarrow}}\ (\Sigma')\ [\underline{\vec{y}}] \right) \in \mathcal{A}$)

This condition ensures that the pattern we are considering is one of the provided patterns. There may be multiple applicable patterns at any single point during the abstraction process. In such cases, any pattern can be chosen without violating soundness. The order in which patterns are applied can affect the performance of our instrumentation analysis. In the implementation, we adopt the heuristic of matching "longest" rules first. That is, we prefer to apply patterns where the left-hand side φ specifies a larger formula, where length is defined as the number of spatial predicates appearing in φ.

Second Rule

The second rule in Figure 5.13 simply discards arithmetic constraints collected during symbolic execution to prevent these from interfering with convergence. An abstract domain for integer variables could also be used, as in [Chang and Rival, 2008].

The rules in Figure 5.13 can be automated provided that the existence of the substitution σ in the first rule can be automatically checked for each element of $\mathcal{A}$. To accomplish this, we guide the search for σ by the assumption $\varphi = \exists \vec{x}_0.\ (\Sigma_0 * \sigma(\Sigma')) \wedge \Pi_0$. Given some symbolic state formula $\varphi_1 = \exists \vec{x}_1.\ \Sigma_1 \wedge \Pi_1$, we search Σ_1 for some collection of spatial predicates matching Σ', modulo some unifying substitution σ. If the search fails, we move on to the next element of $\mathcal{A}$. If the search fails for all elements of $\mathcal{A}$, then we conclude that there is no φ', **c** related to φ by $\underset{\mathcal{A}}{-\text{ABS}\rightarrow}$.

Soundness

We have the following soundness theorem for $\underset{\mathcal{A}}{-\text{ABS}\rightarrow}$.

$$\dfrac{\begin{array}{c}\left([\vec{x}]\ (\Sigma)\ \underset{\Pi'}{\overset{\Pi}{-\text{PAT}\!\!\rightarrow}}\ (\Sigma')\ [\underline{\vec{y}}]\right) \in \mathcal{A} \\ dom(\sigma) = \vec{x} \qquad \varphi = \exists \vec{x}_0.\ (\Sigma_0 * \sigma(\Sigma)) \wedge \Pi_0 \qquad \varphi \Rightarrow \exists \underline{\vec{y}}.\ \sigma(\Pi) \\ \big(fv(\sigma(\Sigma)) - fv(\sigma(\Sigma'))\big) \subseteq \vec{x}_0 \qquad \big(fv(\sigma(\Sigma)) - fv(\sigma(\Sigma'))\big) \cap fv(\Sigma_0) = \emptyset\end{array}}{\varphi \underset{\mathcal{A}}{-\text{ABS}\!\!\rightarrow} \langle \exists \vec{x}_0.\ (\Sigma_0 * \sigma(\Sigma')) \wedge \Pi_0 \mid \underline{\vec{y}} := ?\,\mathtt{;}\ \mathsf{assume}(\sigma(\Pi))\rangle}\ \underline{\vec{y}} \notin fv(\varphi)$$

$$\dfrac{}{\varphi \wedge (e^{\mathrm{i}}_1 \leq e^{\mathrm{i}}_2) \underset{\mathcal{A}}{-\text{ABS}\!\!\rightarrow} \langle \varphi \mid \epsilon \rangle}$$

Figure 5.13: Main rewrite rules for abstraction. We use the notation $\underline{\vec{x}} := ?$ to indicate $\underline{x}_1 := ?\,\mathtt{;}\ \ldots\,\mathtt{;}\ \underline{x}_n := ?$.

Theorem 32. *If all patterns in* $\mathcal{A}$ *are sound, and* $\Gamma \vdash \{\varphi_2\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$ *for some* $\Gamma, \widehat{k}, k$, *and* $\varphi_1 \underset{\mathcal{A}}{-\text{ABS}\!\!\rightarrow} \langle \varphi_2 \mid \mathbf{c}\rangle$, *then* $\Gamma \vdash \{\varphi_1\}\ (\mathbf{c} \fatsemi \widehat{k}) \blacktriangleright_{\underline{\mathrm{IVar}}} k$.

Proof. The proof follows fairly directly from Definition 35 and the rules for instrumentation given in Figure 4.1. The case for the second rule is immediate as $\varphi \wedge (e^{\mathrm{i}}_1 \leq e^{\mathrm{i}}_2) \Rightarrow \varphi$ and so the conclusion follows from STRENGTHENING.

Turning to the first rule, our goal is to show the following.

$$\Gamma \vdash \{\varphi\}\ \underline{\vec{y}} := ?\,\mathtt{;}\ \mathsf{assume}(\sigma(\Pi))\,\mathtt{;}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

We will work backward from this to our assumption that $\Gamma \vdash \{\exists \vec{x}_0.\ (\Sigma_0 {*} \sigma(\Sigma)) {\wedge} \Pi_0\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$. We have from the assumptions of this rule that $\varphi = \exists \vec{x}_0.\ (\Sigma_0 * \sigma(\Sigma)) \wedge \Pi_0$ and $\varphi \Rightarrow \exists \underline{\vec{y}}.\ \sigma(\Pi)$. Together, these give us the following.

$$\varphi \Rightarrow (\exists \vec{x}_0.\ (\Sigma_0 * \sigma(\Sigma)) \wedge \Pi_0) \wedge \exists \underline{\vec{y}}.\ \sigma(\Pi)$$

Our side-condition that $\underline{\vec{y}} \notin fv(\varphi)$ and the fact that $\varphi = \exists \vec{x}_0.\ (\Sigma_0 * \sigma(\Sigma)) \wedge \Pi_0$ gives us that $\underline{\vec{y}} \notin fv(\exists \vec{x}_0.\ (\Sigma_0 * \sigma(\Sigma)) \wedge \Pi_0)$. This lets us move the existential quantifier to the front of the consequent, obtaining

$$\varphi \Rightarrow \exists \underline{\vec{y}}.\ (\exists \vec{x}_0.\ (\Sigma_0 * \sigma(\Sigma)) \wedge \Pi_0) \wedge \sigma(\Pi)$$

Thus, by STRENGTHENING, if we can show the following, we will have proved this case.

$$\Gamma \vdash \{\exists \underline{\vec{y}}.\ (\exists \vec{x}_0.\ (\Sigma_0 * \sigma(\Sigma)) \wedge \Pi_0) \wedge \sigma(\Pi)\}\ \underline{\vec{y}} := ?;\ \mathsf{assume}(\sigma(\Pi));\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

By INST-EXISTS, we will have the goal if we can show

$$\Gamma \vdash \{(\exists \vec{x}_0.\ (\Sigma_0 * \sigma(\Sigma)) \wedge \Pi_0) \wedge \sigma(\Pi)\}\ \mathsf{assume}(\sigma(\Pi));\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

And again working backward from this goal, using rule INST-ASSUME this time, we must show that

$$\Gamma \vdash \{(\exists \vec{x}_0.\ (\Sigma_0 * \sigma(\Sigma)) \wedge \Pi_0) \wedge \sigma(\Pi)\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

We can weaken the precondition by dropping $\sigma(\Pi)$. We do so, applying STRENGTHENING to reduce our goal to

$$\Gamma \vdash \{\exists \vec{x}_0.\ (\Sigma_0 * \sigma(\Sigma)) \wedge \Pi_0\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

This is one of our assumptions, so the case is proved. □

`abstract`

The code for our function `abstract` is given on page 275. We use a comma for concatenation, so the operation $\mathbf{c}, \mathbf{c}'$ gives the concatenation of $\mathbf{c}$ and $\mathbf{c}'$. We will show that this function satisfies the specification given in Figure 5.7.

The invariant for the loop is the following.

Invariant

$$\Gamma \vdash \{\varphi\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k \text{ implies } \Gamma \vdash \{\varphi_0\}\ (\mathbf{c} \fatsemi \widehat{k}) \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

Initially Holds First we show that this is satisfied initially. `abstract`(φ_0) sets φ equal to φ_0 and $\mathbf{c}$ equal to ϵ. Thus, we must show that

$$\Gamma \vdash \{\varphi_0\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k \text{ implies } \Gamma \vdash \{\varphi_0\}\ (\epsilon \fatsemi \widehat{k}) \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

Since $\epsilon \fatsemi \widehat{k} = \widehat{k}$, this is immediate.

Inductively Holds Next, we assume that we have the loop invariant at the current values of φ and $\mathbf{c}$, which we will refer to as φ_1 and $\mathbf{c}_1$.

$$\Gamma \vdash \{\varphi_1\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k \text{ implies } \Gamma \vdash \{\varphi_0\}\ (\mathbf{c}_1 \fatsemi \widehat{k}) \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

We also assume that we have

$$\varphi_1 \xrightarrow[\mathcal{A}]{\textsc{abs}} \langle \varphi' \mid \mathbf{c}' \rangle$$

Now, to show that one execution of the loop preserves this invariant, we assume we have executed $\varphi := \varphi'$ and $\mathbf{c} := \mathbf{c}_1, \mathbf{c}'$. We then show that the loop invariant is re-established. That is, the following holds.

$$\Gamma \vdash \{\varphi'\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k \text{ implies } \Gamma \vdash \{\varphi_0\}\ (\mathbf{c}_1, \mathbf{c}') \fatsemi \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

We first assume $\Gamma \vdash \{\varphi'\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k$. By Theorem 32 we then have $\Gamma \vdash \{\varphi_1\}\ (\mathbf{c}' \fatsemi \widehat{k}) \blacktriangleright_{\underline{\mathrm{IVar}}} k$. The loop invariant from previous iterations then gives us $\Gamma \vdash \{\varphi_0\}\ \mathbf{c}_1 \fatsemi (\mathbf{c}' \fatsemi \widehat{k}) \blacktriangleright_{\underline{\mathrm{IVar}}} k$. Since $(\mathbf{c}_1, \mathbf{c}') \fatsemi \widehat{k} = \mathbf{c}_1 \fatsemi (\mathbf{c}' \fatsemi \widehat{k})$ we have now established the conclusion of the loop invariant for this iteration.

Implies Specification Finally we show that the loop invariant implies the specification. The invariant is

$$\Gamma \vdash \{\varphi\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k \text{ implies } \Gamma \vdash \{\varphi_0\}\ (\mathbf{c} \fatsemi \widehat{k}) \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

and the specification requires that if `abstract`(φ_0) returns $(\varphi, \mathbf{c})$ then the following holds

$$\Gamma \vdash \{\varphi\}\ \widehat{k} \blacktriangleright_{\underline{\mathrm{IVar}}} k \text{ implies } \Gamma \vdash \{\varphi_0\}\ (\mathbf{c} \fatsemi \widehat{k}) \blacktriangleright_{\underline{\mathrm{IVar}}} k$$

As the two implications are the same, the proof is complete.

5.7.4 Additional Comments

There is much more that can be said about abstraction. For some starting points in the context of shape analysis with separation logic, see [Yang et al., 2008, Chang et al., 2007,

Function `abstract(`φ_0`)`. Returns a weaker symbolic state φ' along with a list of instrumentation commands associated with the transition from φ to φ'. The operation $\mathbf{c}, \mathbf{c}'$ gives the concatenation of $\mathbf{c}$ and $\mathbf{c}'$.

```
$\varphi := \varphi_0$
$\mathbf{c} := \epsilon$
while $\exists \varphi', \mathbf{c}'.\ \varphi \xrightarrow[\mathcal{A}]{\text{ABS}} \langle \varphi' \mid \mathbf{c}' \rangle$ do
  $\varphi := \varphi'$
  $\mathbf{c} := \mathbf{c}, \mathbf{c}'$
end
return $(\varphi, \mathbf{c})$
```

Chang and Rival, 2008]. Each of these presents a different take on what criteria to use when deciding whether or not to weaken a formula and by how much. In particular, [Yang et al., 2008] notes the importance of keeping track of whether predicate instances are known to represent non-empty data structures. Depending on other details of the language of symbolic state formulae, this information can be necessary to prove certain examples.

Non-emptiness information is not preserved by the abstraction patterns presented in the previous section, though our implementation does have a command line parameter to toggle tracking of non-emptiness information. In the treatment of abstraction just presented, we chose to concentrate on the core idea of abstraction, which is the use of the spatial portion of the heap to guide the selection and application of abstraction rules. The rules themselves can be made to keep more or less information, and the conditions that trigger them can be adjusted, but the basic matching strategy is the same in all current systems of which the authors are aware.

5.8 Example (continued)

Now that we have a definition for `abstract`, we return to our list traversal example, reproduced below.

$$\begin{array}{rl} L_1 : & \textcircled{1}\ \mathsf{branch}\ x \neq \mathsf{nil} \Rightarrow \textcircled{2}\ x := x.\mathsf{next}\texttt{;}\ \textcircled{3}\ \mathsf{goto}\ L_1, \\ & \qquad\qquad x = \mathsf{nil} \Rightarrow \textcircled{4}\ \mathsf{halt\ end} \end{array}$$

We had previously obtained the following formula just prior to evaluating the goto L_1 statement which triggered a call to `abstract`.

$$\exists x'.\ (x' \mapsto [\mathsf{next} : x] * ls(\underline{n}_0; x, \mathsf{nil}))$$

We will now execute our new definition of `abstract` with the following abstraction patterns. These are the actual patterns used by our tool for singly-linked lists.

$$[x, y, z, \underline{n}_0]\ \big(x \mapsto [\mathsf{next} : y] * ls(\underline{n}_0; y, z)\big) \overset{\mathsf{true}}{\underset{\underline{n}=\underline{n}_0+1}{\xrightarrow{\text{PAT}}}} \big(ls(\underline{n}; x, z)\big)\ [\underline{n}] \tag{5.15}$$

$$[x, y, z, \underline{n}_0]\ \big(ls(\underline{n}_0; x, y) * y \mapsto [\mathsf{next} : z]\big) \overset{\mathsf{true}}{\underset{\underline{n}=\underline{n}_0+1}{\xrightarrow{\text{PAT}}}} \big(ls(\underline{n}; x, z)\big)\ [\underline{n}] \tag{5.16}$$

$$[x, y, z, \underline{n}_1, \underline{n}_2]\ \big(ls(\underline{n}_1; x, y) * ls(\underline{n}_2; y, z)\big) \overset{\mathsf{true}}{\underset{\underline{n}=\underline{n}_1+\underline{n}_2}{\xrightarrow{\text{PAT}}}} \big(ls(\underline{n}; x, z)\big)\ [\underline{n}] \tag{5.17}$$

$$[x, z]\ \big(x \mapsto [\mathsf{next} : z]\big) \overset{\mathsf{true}}{\underset{\underline{n}=1}{\xrightarrow{\text{PAT}}}} \big(ls(\underline{n}; x, z)\big)\ [\underline{n}] \tag{5.18}$$

We can abstract $\exists x'.\ (x' \mapsto [\mathsf{next} : x] * ls(\underline{n}_0; x, \mathsf{nil}))$ by applying (5.18) to obtain

$$\exists x'.\ ls(\underline{n}_1; x', x) * ls(\underline{n}_0; x, \mathsf{nil}) \tag{5.19}$$

along with the instrumentation commands $\underline{n}_1 := ?$; $\mathsf{assume}(\underline{n}_1 = 1)$. This formula will be an invariant at L_1, as we can see by executing `geninstCont` starting from this state. If we do this, the formula we obtain at location $\textcircled{3}$, just before goto L_1, is

$$\exists x', x_2.\ ls(\underline{n}_1; x', x_2) * (x_2 \mapsto [\mathsf{next} : x]) * ls(\underline{n}_2; x, \mathsf{nil})$$

along with the instrumentation command $\underline{n}_2 := ?$; $\mathsf{assume}(\underline{n}_0 = \underline{n}_2 + 1)$. Now we can execute `implies` to verify that this formula in fact implies the invariant (5.19). `implies` first calls `abstract`, obtaining instrumentation commands $\underline{n}_3 := ?$; $\mathsf{assume}(\underline{n}_3 = \underline{n}+1)$ and state formula

$$\exists x'.\ ls(\underline{n}_3; x', x) * ls(\underline{n}_2; x, \mathsf{nil})$$

Next we search for a frame inference proof, using InstL to match $\underline{n}_2$ to $\underline{n}_0$ and $\underline{n}_3$ to $\underline{n}_1$. This results in instrumentation commands $\underline{n}_0 := \underline{n}_2$; $\underline{n}_1 := \underline{n}_3$. Note that `implies` calls `abstract` before performing the frame inference proof. This compensates for the fact that the frame inference system does not contain a rule to expand inductive predicate instances on the right (and not having such a rule in frame inference is useful as this reduces the proof space that must be searched).

Combining all this, the entire process results in the instrumented continuation in Figure 5.14. Note that since there are two symbolic state formulae associated with L_1 in the final version of Γ (the initial state and the discovered invariant) we have a non-deterministic choice between the instrumentations corresponding to each element of $\Gamma(L_1)$.

There are a number of simplifications that can be made to this program while retaining the same semantics. For example, the sequence of commands $\underline{n}_1 := ?$; $\mathsf{assume}(\underline{n}_1 = 1)$ is equivalent to $\underline{n}_1 := 1$. We proved this in Section 4.1.3 in the context of the derivability of the Inst-Assign rule. Similarly, $\underline{n}_3 := ?$; $\mathsf{assume}(\underline{n}_3 = \underline{n}_1 + 1)$ is equivalent to $\underline{n}_3 := \underline{n}_1 + 1$. Noting that $\mathsf{assume}(\underline{n} = \underline{n}_0 + 1)$ is equivalent to $\mathsf{assume}(\underline{n}_0 = \underline{n} - 1)$ allows us to also rewrite $\underline{n}_0 := ?$; $\mathsf{assume}(\underline{n} = \underline{n}_0 + 1)$ to the command $\underline{n}_0 := \underline{n} - 1$.

We can also eliminate intermediate writes. The sequence $\underline{n}_3 := \underline{n}_1 + 1$; $\ldots$; $\underline{n}_1 := \underline{n}_3$; $\ldots$ can be reduced to $\underline{n}_1 := \underline{n}_1 + 1$ in cases where $\underline{n}_3$ is not read or written by other commands. Simplification based on these equivalences is implemented in our tool for list-based data structures. This results in a quite dramatic reduction in the size of the instrumented program. The simplified program for this example is given in Figure 5.15.

Such simplifications are possible because the instrumentation commands for lists are deterministic. For data structures like trees, where an instrumentation based on tracking the size of the tree is inherently non-deterministic, such translations of assume statements to assignments no longer apply. That is not to say, however, that there are is no hope of simplifying more complex examples. Even though the non-determinism is an important

```
$L_1$ :  branch
           true ⇒
             branch
               $x \neq$ nil ⇒ assume(true);
                 branch
                   $\underline{n} = 0$ ⇒ assume(true); assume(false); halt,
                   $\underline{n} > 0$ ⇒
                     $\underline{n}_0 := ?$; assume($\underline{n} = \underline{n}_0 + 1$); $x := x$.next;
                     $\underline{n}_1 := ?$; assume($\underline{n}_1 = 1$); goto $L_1$
                 end,
               $x =$ nil ⇒ assume(true); halt
             end
           true ⇒
             branch
               $x \neq$ nil ⇒ assume(true);
                 branch
                   $\underline{n}_0 = 0$ ⇒ assume(true); assume(false); halt,
                   $\underline{n}_0 > 0$ ⇒
                     $\underline{n}_2 := ?$; assume($\underline{n}_0 = \underline{n}_2 + 1$); $x := x$.next;
                     $\underline{n}_3 := ?$; assume($\underline{n}_3 = \underline{n}_1 + 1$); $\underline{n}_0 := \underline{n}_2$; $\underline{n}_1 := \underline{n}_3$;
                     goto $L_1$
                 end
               $x =$ nil ⇒ assume(true); assume(false) halt
             end
         end
```

$$\begin{aligned}\Gamma(L_1) \quad = \{ \quad & ls(\underline{n}; x, \mathsf{nil}), \\ & \exists x'.\ (ls(\underline{n}_1; x', x) * ls(\underline{n}_0; x, \mathsf{nil})) \ \}\end{aligned}$$

Figure 5.14: The full instrumentation of the singly-linked list example.

```
$L_1$ :  branch
            true ⇒
               branch
                  $x \neq$ nil ⇒ assume(true);
                     branch
                        $\underline{n} = 0$ ⇒ assume(true); assume(false); halt,
                        $\underline{n} > 0$ ⇒
                           $\underline{n}_0 := \underline{n} - 1$; $x := x$.next;
                           $\underline{n}_1 := 1$; goto $L_1$
                     end,
                  $x =$ nil ⇒ assume(true); halt
               end
            true ⇒
               branch
                  $x \neq$ nil ⇒ assume(true);
                     branch
                        $\underline{n}_0 = 0$ ⇒ assume(true); assume(false); halt,
                        $\underline{n}_0 > 0$ ⇒
                           $\underline{n}_0 := \underline{n}_0 - 1$; $x := x$.next;
                           $\underline{n}_1 = \underline{n}_1 + 1$; goto $L_1$
                     end
                  $x =$ nil ⇒ assume(true); assume(false) halt
               end
         end
```

$$
\begin{aligned}
\Gamma(L_1) \quad = \{ \quad & \mathit{ls}(\underline{n}; x, \mathsf{nil}), \\
& \exists x'.\ (\mathit{ls}(\underline{n}_1; x', x) * \mathit{ls}(\underline{n}_0; x, \mathsf{nil}))\ \}
\end{aligned}
$$

Figure 5.15: A simplified version of the instrumentation given in Figure 5.14.

part of the instrumentation for branching data structures, the approach presented in this section still produces unnecessary intermediate variables. When passing our numeric programs to external tools, the number of variables is often an important quantity that we would like to minimize. Finding methods of eliminating these unnecessary intermediate variables in the general case is ongoing work.

5.9 Tracking Flow of Control

Note that the instrumented program produced for our example contains some paths that we know to be infeasible. For example, it should not be possible to start at the initial state and immediately execute the second case of the main branch. This case was generated from the precondition

$$\exists x'. \ (ls(\underline{n}_1; x', x) * ls(\underline{n}_0; x, \mathsf{nil}))$$

but this formula does not hold in the initial state of $ls(\underline{n}; x, \mathsf{nil})$ (the variables $\underline{n}_0$ and $\underline{n}_1$ have not yet been assigned values). We can rule out such spurious paths in the following way. We number each element of $\Gamma(L_1)$ and add an instrumentation variable that tracks which precondition was supplied for the current execution of the code at L_1. This counter is initially set to the value corresponding to the initial state. If we make this change, giving the initial state number 1 and the invariant number 2, and using $\underline{p}$ to track the precondition from which we are executing, we obtain the code in Figure 5.16. Control now begins at L_0 so that $\underline{p}$ can be assigned the correct value.

We can apply this control-flow-tracking transformation to the general case. Currently, when we emit the final instrumented continuation in `instrument`, we iterate over each continuation in the original program, emitting a branch of the form $\mathsf{branch\ true} \Rightarrow \widehat{k}_1, \ldots, \mathsf{true} \Rightarrow \widehat{k}_n \ \mathsf{end}$ where $\widehat{k}_1, \ldots, \widehat{k}_n$ are instrumentations of the original continuation starting from different preconditions. If we number the preconditions from 1 to n, we can track viable paths more precisely by emit a branch of the form

$$\mathsf{branch}\ (\underline{p} = 1) \Rightarrow \widehat{k}_1, \ldots, (\underline{p} = n) \Rightarrow \widehat{k}_n\ \mathsf{end}$$

Then, in `geninstCont`, when we process a goto l command and discover that the current state implies the i^{th} element in the set $\Gamma(l)$, we emit the instrumentation command $\underline{p} = i$ just prior to the goto l statement.

This records in the code more information about feasible paths. However, not all external tools will make use of this information. It is common for program analysis tools to handle control flow and data differently. Thus, our trick of encoding control flow information in an extra integer-valued variable may not work. In such cases, since the domain of

```
$L_0$ :   $\underline{p} := 1$; goto $L_1$
$L_1$ :   branch
            $\underline{p} = 1 \Rightarrow$
                branch
                    $x \neq$ nil $\Rightarrow$ assume(true);
                        branch
                            $\underline{n} = 0 \Rightarrow$ assume(true); assume(false); halt,
                            $\underline{n} > 0 \Rightarrow$
                                $\underline{n}_0 := \underline{n} - 1$; $x := x$.next; $\underline{n}_1 := 1$;
                                $\underline{p} := 2$; goto $L_1$
                        end,
                    $x =$ nil $\Rightarrow$ assume(true); halt
                end
            $\underline{p} = 2 \Rightarrow$
                branch
                    $x \neq$ nil $\Rightarrow$ assume(true);
                        branch
                            $\underline{n}_0 = 0 \Rightarrow$ assume(true); assume(false); halt,
                            $\underline{n}_0 > 0 \Rightarrow$
                                $\underline{n}_0 := \underline{n}_0 - 1$; $x := x$.next; $\underline{n}_1 := \underline{n}_1 + 1$;
                                $\underline{p} := 2$; goto $L_1$
                        end
                    $x =$ nil $\Rightarrow$ assume(true); halt
                end
          end
```

$$
\begin{array}{lll}
\Gamma(L_0) & = \{ & \mathit{ls}(\underline{n}; x, \mathsf{nil}) \ \} \\
\Gamma(L_1) & = \{ & \mathit{ls}(\underline{n}; x, \mathsf{nil}) \wedge \underline{p} = 1, \\
 & & \exists x'.\ (\mathit{ls}(\underline{n}_1; x', x) * \mathit{ls}(\underline{n}_0; x, \mathsf{nil})) \wedge \underline{p} = 2 \ \}
\end{array}
$$

Figure 5.16: An instrumentation of the singly-linked list example that tracks flow of control using a variable $\underline{p}$.

our $\underline{p}$ variable is finite, we can fully unroll the program with respect to $\underline{p}$, as is commonly done in bounded model checking [Biere et al., 1999], before passing it to the analysis tool.

5.10 Translating Branch Conditions

We will now consider what happens when we want to prove a property of our example program. Suppose we are interested in showing termination, and in using an external termination prover to do the termination reasoning. Then we first convert the instrumented program that we have produced to a numeric program using the projection operation defined in Section 4.4. The result of the operation is given in Figure 5.17, where we have projected the program onto the set of instrumentation variables $\underline{\mathrm{IVar}}$. The result is that the branch conditions involving x become true and the $x := x.\mathsf{next}$ commands disappear.

The example does terminate in all cases, as the branch that executes goto L_1 in the $\underline{p} = 2$ case is guarded by $\underline{n}_0 > 0$. This condition cannot remain true forever since this branch also decreases $\underline{n}_0$. However, there are important properties of the program that are not captured by this abstraction. Specifically, while the program will always terminate, it is allowed to "terminate early." The instrumented program terminates exactly when $\underline{n}_0 = 0$, however the numeric abstraction may terminate with any value of $\underline{n}_0$ (by executing the second true branch in the $\underline{p} = 2$ case of L_1.

As with our discussion of flow of control in the previous section, the result is still sound, but the program contains paths that are known to be spurious. Thus we can obtain a more precise abstraction if we can rule out these paths.

Consider the program below, which iterates through a list and then checks that $x = \mathsf{nil}$ following the traversal (aborting if this does not hold). Triggering the abort in this program is not possible.

```
$L_1$ :   ① branch $x \neq \mathsf{nil} \Rightarrow$ ② $x = x.\mathsf{next}$; ③ goto $L_1$,
                    $x = \mathsf{nil} \Rightarrow$ ④ goto $L_2$ end
$L_2$ :   ⑤ branch $x \neq \mathsf{nil} \Rightarrow$ ⑥ abort,
                    $x = \mathsf{nil} \Rightarrow$ ⑦ halt end
```

```
$L_0$ :   $\underline{p} := 1$; goto $L_1$
$L_1$ :   branch
              $\underline{p} = 1 \Rightarrow$
                  branch
                      true ⇒ assume(true);
                          branch
                              $\underline{n} = 0 \Rightarrow$ assume(true); assume(false); halt,
                              $\underline{n} > 0 \Rightarrow$
                                  $\underline{n}_0 := \underline{n} - 1$; $\underline{n}_1 := 1$;
                                  $\underline{p} := 2$;goto $L_1$
                          end,
                      true ⇒ assume(true); halt
                  end
              $\underline{p} = 2 \Rightarrow$
                  branch
                      true ⇒ assume(true);
                          branch
                              $\underline{n}_0 = 0 \Rightarrow$ assume(true); assume(false); halt,
                              $\underline{n}_0 > 0 \Rightarrow$
                                  $\underline{n}_0 := \underline{n}_0 - 1$; $\underline{n}_1 := \underline{n}_1 + 1$;
                                  $\underline{p} := 2$;goto $L_1$
                          end
                      true ⇒ assume(true); halt
                  end
          end
```

Figure 5.17: The numeric program corresponding to the program in Figure 5.16.

A simplified version of a numeric program for this code is given below. For each branch condition, we write in square brackets the original program branch condition, if any, associated with that branch. We have eliminated the branches of the form $\underline{n} = 0 \Rightarrow$ assume(true); assume(false) since the assume(false) ensures that there are no executions along this branch. We then replaced the single remaining "$\underline{n} > 0 \Rightarrow \ldots$"

branch with “assume($\underline{n} > 0$); ...,” which is equivalent.

```
$L_0$ :  $\underline{p} := 1$; goto $L_1$
$L_1$ :  branch
           $\underline{p} = 1 \Rightarrow$
               branch
                   true $[x \neq \mathsf{nil}] \Rightarrow$ assume(true); assume($\underline{n} > 0$);
                       $\underline{n}_0 := \underline{n} - 1$; $\underline{n}_1 := 1$;
                       $\underline{p} := 2$; goto $L_1$
                   true $[x = \mathsf{nil}] \Rightarrow$ assume(true); goto $L_2$
               end
           $\underline{p} = 2 \Rightarrow$
               branch
                   true $[x \neq \mathsf{nil}] \Rightarrow$ assume(true); assume($\underline{n}_0 > 0$);
                       $\underline{n}_0 := \underline{n}_0 - 1$; $\underline{n}_1 := \underline{n}_1 + 1$;
                       $\underline{p} := 2$; goto $L_1$
                   true $[x = \mathsf{nil}] \Rightarrow$ assume(true); goto $L_2$
               end
         end
$L_2$ :  branch
           true $[x \neq \mathsf{nil}] \Rightarrow$ abort
           true $[x = \mathsf{nil}] \Rightarrow$ halt
         end
```

There are two types of assume commands that have been inserted here. The assume($\underline{n} > 0$) and assume($\underline{n}_0 > 0$) commands came from expanding the list segment predicate in order to prove that x is in the heap for the processing of the $x := x.\mathsf{next}$ command. The assume(true) statements come from the call to `branchAnnot` in `geninstCont`. Because the DEFL rule in frame inference is the only operation that inserts instrumentation branches into the code, we will only record information about $\underline{n}$ and $\underline{n}_0$ when we are forced to expand an inductive predicate. Branches such as those associated with the $x \neq \mathsf{nil}$ conditions in L_1 and L_2, which do not access the heap following the branch, do not result in information about $\underline{n}$ and $\underline{n}_2$ being recorded.

What we would like to do is incorporate into the automated analysis some version of the INST-BRANCHTRANS derived rule from Section 4.1.3. To do so, we need some method of finding pure formulae implied by the current symbolic heap. One approach is suggested by our DefL rule and the fact that branches that make use of DefL already end up recording some information about the instrumentation variables. This occurs because DefL case splits on the conditions associated with an inductive predicate and then LeftPureFalse effectively prunes any impossible branches, thus recording in the code which values of the instrumentation variables are consistent with the current symbolic state.

One approach to recording more information at branch points is to have `branchAnnot` eagerly try to expand all inductive predicates in the current symbolic state in order to test which expansions are consistent. This can be accomplished fairly easily and generally by augmenting our system for frame inference. We add support for *pure abduction*, which is similar to the abductive inference of spatial predicates discussed in [Calcagno et al., 2009] but discovers pure rather than spatial assumptions. The pure abduction problem is to produce from φ and φ' a pure formula Π such that $\varphi \wedge \Pi \Rightarrow \varphi'$. To accomplish this we modify the form of our sequents to the following.

$$\Pi_a + \Sigma_a \;[]\; \varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \;/\!\!/\; \Gamma \vdash \widehat{k}$$

We have added a component Π_a to the left, which is the pure hypothesis necessary to guarantee the conclusion. Π_a is considered an output in the algorithmic interpretation of our inference system. A derivation of the new sequent form above guarantees that the following is derivable in the old system.

$$\Sigma_a \;[]\; \varphi \wedge \Pi_a \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \;/\!\!/\; \Gamma \vdash \widehat{k}$$

For all rules except DefL, Π_a is simply passed unchanged from the hypothesis to the conclusion. So, for example, PtoMatches becomes

PtoMatches

$$\frac{\Pi_a + \Sigma_a * (e \mapsto \rho) \;[]\; \varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \;/\!\!/\; \Gamma \vdash \widehat{k}}{\Pi_a + \Sigma_a \;[]\; (e \mapsto \rho) * \varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' * (e \mapsto \rho) \;/\!\!/\; \Gamma \vdash \widehat{k}}$$

The axioms set Π_a to true, since when they hold no additional assumptions are necessary.

$$\textsc{RightPure}\quad \frac{\Pi \Rightarrow \exists\vec{x}.\,\Pi' \qquad f_k(\exists\vec{x}.\,(\Sigma_a * \Sigma) \wedge \Pi') = \mathsf{Some}\big(\Gamma, \widehat{k}\big)}{\mathsf{true} + \Sigma_a \,[\!]\, \Sigma \wedge \Pi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \exists\vec{x}.\,\mathbf{emp} \wedge \Pi' \;/\!\!/\; \Gamma \vdash \widehat{k}}$$

The DEFL rule then becomes the following which, rather than requiring all cases to be provable, instead checks that the conclusion is provable for some subset of the cases. It then includes the negation of all the cases which are not provable in the constraint Π_a that is returned. The idea is that, if these negations had been provided as assumptions, then all the non-provable cases would have followed from LEFTPUREFALSE due to the conditions for those cases being inconsistent with these assumptions. We will present an example shortly.

We write I to represent a set of integers and write $\underset{i\in I}{\mathsf{branch}}$ to represent the branch with one case for each element i of I (just as $\bigcup_{i\in I}$ represents the union with one component for each $i \in I$). As is standard, the empty iterated conjunction is equal to true. We write $\neg I$ for the complement of I. This is all cases that are not in I. So if the cases are $\{1 \ldots n\}$ and $I \subseteq \{1 \ldots n\}$ (as the rule requires), then $\neg I$ is $\{1 \ldots n\} - I$.

$$\textsc{DefL}\quad \frac{\begin{array}{c} \big(d(\vec{v}) \;\texttt{<=>}\; C_1(\vec{v}) \mid \ldots \mid C_n(\vec{v})\big) \in \mathbf{S} \\ C_i(\vec{e}) = \big(\Pi_i : \text{let } \underline{\vec{z}}_i \text{ satisfy } \Pi'_i \text{ in } \varphi_i\big) \qquad I \subseteq \{1, \ldots, n\} \\ \forall i \in I.\ \big(\Pi_{a_i} + \Sigma_a \,[\!]\, (\varphi * \varphi_i) \wedge \Pi_i \wedge \Pi'_i \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \;/\!\!/\; \Gamma_i \vdash \widehat{k}_i\big) \end{array}}{\begin{array}{l} \bigwedge\limits_{i\in I}(\Pi_i \Rightarrow \Pi_{a_i}) \wedge \bigwedge\limits_{i\in\neg I}(\neg\Pi_i) + \Sigma_a \,[\!]\, \varphi * d(\vec{e}) \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' /\!\!/ \\ \quad \big(\bigcup_{i\in I}\Gamma_i\big) \vdash \underset{i\in I}{\mathsf{branch}}\ \ldots, \Pi_i \Rightarrow \underline{\vec{z}}_i := ?\,\mathsf{;}\,\mathsf{assume}(\Pi'_i)\,\mathsf{;}\,\widehat{k}_i, \ldots \mathsf{end} \end{array}}\ \forall i.\ \vec{z}_i \notin fv(\varphi, \Sigma_a, \Pi_i)$$

The assumptions Π_a that build up can be simplified using rules of Boolean logic, as we show later in an example.

The soundness result then becomes the following.

Theorem 33. *If* $\Pi_a + \varphi \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \varphi' \;/\!\!/\; \Gamma \vdash \widehat{k}$ *then*

$$\Gamma \vdash \{\varphi \wedge \Pi_a\}\; \widehat{k} \;\blacktriangleright_{\underline{\text{IVar}}}\; k$$

Soundness of the augmented proof system is straightforward. For most rules it follows directly from the induction hypothesis, since Π_a is not changed from premise to conclusion. For the axioms, the same proof can be reused since $\varphi \wedge \mathsf{true} \Leftrightarrow \varphi$. For DEFL, the reasoning is similar to that for the original rule in terms of reducing cases to instances of the inductive hypothesis. The main addition is that we must show that the omitted cases have proofs if we assume Π_a. But Π_a contains the negation of the case conditions for all omitted cases, so $\varphi \wedge \Pi_a$ implies false in every omitted case, allowing us to prove each of these cases with LEFTPUREFALSE.

We can now give a definition of `branchAnnot` that uses this augmented frame inference procedure to introduce assumptions on instrumentation variables at every branch case present in the original program. The code for the function is listed on this page. Given the current symbolic state formula φ, the function tries to prove for each branch condition e_i that $\varphi \wedge e_i \Rightarrow \mathsf{false}$. It does this by making a call into frame inference. If the proof search succeeds, then Π_a will contain the conditions under which this implication holds. This makes Π_a an *under-approximation* of the *negation* of the branch condition. To obtain an over-approximation of the branch condition, we simply negate Π_a.

Function `branchannot(`$\varphi, e_1, e_2, \ldots, e_n$`)`. Function for annotating original branches with pure formulae over the instrumentation variables that are guaranteed to hold by each original branch. φ is the current symbolic state and $e_1, \ldots, e_n$ are the conditions to be translated.

```
fun $f(\varphi)$ =
  return $\mathsf{Some}(\emptyset, \mathsf{halt})$
in
  foreach $e_i$ do
    if $\Pi_a + \varphi \wedge e_i \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \mathbf{emp} \wedge \mathsf{false} \mathbin{/\!\!/} \Gamma \vdash \widehat{k}$ then
      $e_i' := \neg\Pi_a$
    end
  end
  return $(e_1', \ldots, e_n')$
```

We will now show an example demonstrating the use of our augmented version of frame inference to infer conditions on instrumentation variables. Suppose we have the following state, using the $ls(\underline{n}; x, y)$ predicate from earlier.

$$(ls(\underline{n}_1; x, y) * ls(\underline{n}_2; y, x)) \wedge \underline{n}_1 + \underline{n}_2 > 0$$

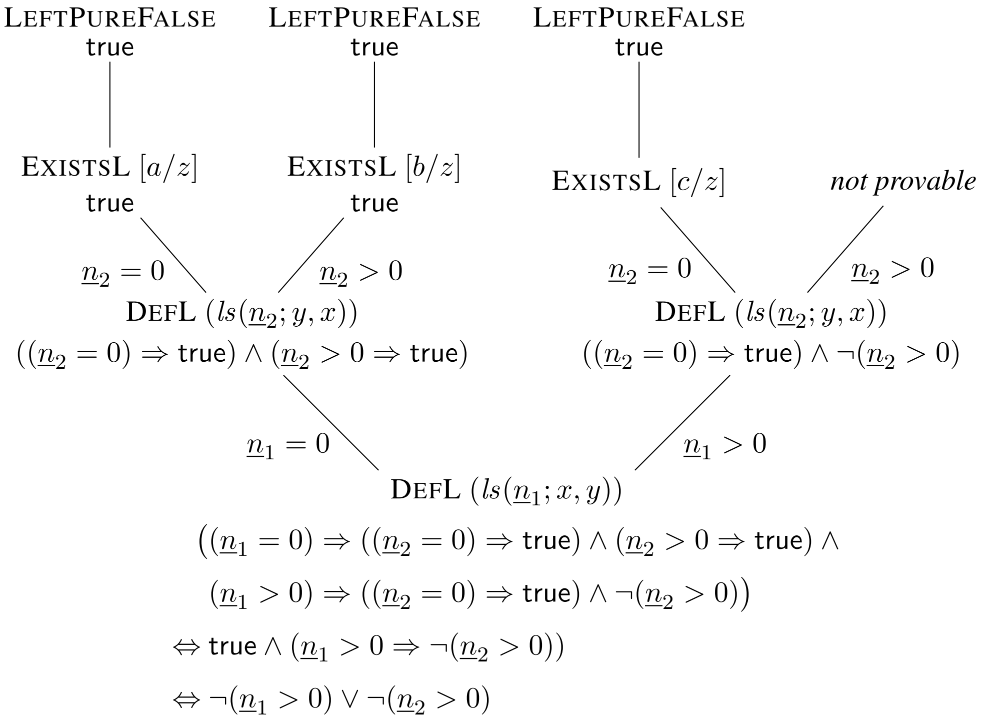
This indicates that the heap consists of a non-empty cyclic list with x and y pointing into it. We will translate the branch condition $x \neq y$ into a condition on $\underline{n}_1$ and $\underline{n}_2$. We give the proof tree in Figure 5.18, following the syntax from section 5.6, where we annotate each node in the tree with the name of the rule that is applied and list any parameters that must be chosen next to the name. Below the name of the rule, we write the output. Since we are only interested in the set of assumptions that are returned, we only list Π_a and omit Γ and $\widehat{k}$. We write *not provable* for the cases for which no proof can be found.

The derivation below the root of the tree in the figure demonstrates how the condition that is returned can be simplified to $\neg(\underline{n}_1 > 0) \vee \neg(\underline{n}_2 > 0)$. This then gets negated and used as the assumption for this case. Thus, we have discovered that in the state

$$(ls(\underline{n}_1; x, y) * ls(\underline{n}_2; y, x)) \wedge \underline{n}_1 + \underline{n}_2 > 0$$

if $x \neq y$ then it is also the case that $\underline{n}_1 > 0 \wedge \underline{n}_2 > 0$. We can perform a similar analysis working from the condition $x = y$. We will get a proof tree like that in Figure 5.18, but the *not provable* and LEFTPUREFALSE cases will be flipped. The condition returned will simplify to $(\underline{n}_1 \neq 0) \wedge (\underline{n}_2 \neq 0)$ resulting in an assumption for the case of $(\underline{n}_1 = 0) \vee (\underline{n}_2 = 0)$, exactly the conditions under which the state allows us to conclude $x = y$ (although the result is not always exact; in general it is an over-approximation of the condition we are analyzing).

We now return to our list traversal example from page 284, in order to insert branch assumptions and obtain an abstraction that is more precise. Figure 5.19 gives the result. In the $\underline{p} = 1$ case, the condition that we obtain for $x \neq \mathsf{nil}$ is $\underline{n} > 0$ and for $x = \mathsf{nil}$ we obtain $\underline{n} = 0$. For $\underline{p} = 2$ the conditions are $\underline{n}_0 > 0$ and $\underline{n}_0 = 0$. We have also expanded the continuation at L_2 to account for the fact that it is executed from two different preconditions.


LEFTPUREFALSE
true
LEFTPUREFALSE
true
LEFTPUREFALSE
true
EXISTSL $[a/z]$
true
EXISTSL $[b/z]$
true
EXISTSL $[c/z]$
not provable
$\underline{n}_2 = 0$
$\underline{n}_2 > 0$
$\underline{n}_2 = 0$
$\underline{n}_2 > 0$
DEFL $(ls(\underline{n}_2; y, x))$
$((\underline{n}_2 = 0) \Rightarrow \mathsf{true}) \wedge (\underline{n}_2 > 0 \Rightarrow \mathsf{true})$
DEFL $(ls(\underline{n}_2; y, x))$
$((\underline{n}_2 = 0) \Rightarrow \mathsf{true}) \wedge \neg(\underline{n}_2 > 0)$
$\underline{n}_1 = 0$
$\underline{n}_1 > 0$
DEFL $(ls(\underline{n}_1; x, y))$
$\big((\underline{n}_1 = 0) \Rightarrow ((\underline{n}_2 = 0) \Rightarrow \mathsf{true}) \wedge (\underline{n}_2 > 0 \Rightarrow \mathsf{true}) \wedge$
$(\underline{n}_1 > 0) \Rightarrow ((\underline{n}_2 = 0) \Rightarrow \mathsf{true}) \wedge \neg(\underline{n}_2 > 0)\big)$
$\Leftrightarrow \mathsf{true} \wedge (\underline{n}_1 > 0 \Rightarrow \neg(\underline{n}_2 > 0))$
$\Leftrightarrow \neg(\underline{n}_1 > 0) \vee \neg(\underline{n}_2 > 0)$

Derivation of

$$\Pi_a + (ls(\underline{n}_1; x, y) * ls(\underline{n}_2; y, x)) \wedge \underline{n}_1 + \underline{n}_2 > 0 \wedge x \neq y \underset{\mathbf{S}}{\Longrightarrow}_{f_k} \mathbf{emp} \wedge \mathsf{false} \mathbin{/\!\!/} \Gamma \vdash \widehat{k}$$

Figure 5.18: Proof for the given frame inference query. Below each rule name we show the value that Π_a has in the conclusion of that rule.

It is now clear due to the additional assume statements that goto L_2 can only be executed in the $\underline{p} = 1$ case if $\underline{n} = 0$. The $\mathsf{assume}(\underline{n} > 0)$ that guards the abort command in L_2 then ensures that abort will not be reached in any execution. A similar situation holds with $\underline{n}_0$ for $\underline{p} = 2$.

In this example, unreachability of abort could have been proved with pure heap reasoning (integer values are not required). However, for more complicated properties, such as computing upper bounds on variables, and for more complex examples with multiple integer quantities involved, it can be useful to have a more accurate numeric abstraction.

```
$L_0$ :  $\underline{p} := 1$; goto $L_1$
$L_1$ :  branch
           $\underline{p} = 1 \Rightarrow$
              branch
                 true $[x \neq \mathsf{nil}] \Rightarrow$ assume($\underline{n} > 0$); assume($\underline{n} > 0$);
                    $\underline{n}_0 := \underline{n} - 1$; $\underline{n}_1 := 1$;
                    $\underline{p} := 2$; goto $L_1$
                 true $[x = \mathsf{nil}] \Rightarrow$ assume($\underline{n} = 0$); $\underline{p} := 1$; goto $L_2$
              end
           $\underline{p} = 2 \Rightarrow$
              branch
                 true $[x \neq \mathsf{nil}] \Rightarrow$ assume($\underline{n}_0 > 0$); assume($\underline{n}_0 > 0$);
                    $\underline{n}_0 := \underline{n}_0 - 1$; $\underline{n}_1 := \underline{n}_1 + 1$;
                    $\underline{p} := 2$; goto $L_1$
                 true $[x = \mathsf{nil}] \Rightarrow$ assume($\underline{n}_0 = 0$); $\underline{p} := 2$; goto $L_2$
              end
         end
$L_2$ :  branch
           $\underline{p} = 1 \Rightarrow$
              branch
                 true $[x \neq \mathsf{nil}] \Rightarrow$ assume($\underline{n} > 0$); abort
                 true $[x = \mathsf{nil}] \Rightarrow$ assume($\underline{n} = 0$); halt
              end
           $\underline{p} = 2 \Rightarrow$
              branch
                 true $[x \neq \mathsf{nil}] \Rightarrow$ assume($\underline{n}_0 > 0$); abort
                 true $[x = \mathsf{nil}] \Rightarrow$ assume($\underline{n}_0 = 0$); halt
              end
         end
```

$$\begin{aligned}
\Gamma(L_1) \quad &= \{ \quad ls(\underline{n}; x, \mathsf{nil}) \wedge \underline{p} = 1, \\
&\qquad \exists x'.\, (ls(\underline{n}_1; x', x) * ls(\underline{n}_0; x, \mathsf{nil})) \wedge \underline{p} = 2 \ \} \\
\Gamma(L_2) \quad &= \{ \quad ls(\underline{n}; x, \mathsf{nil}) \wedge \underline{p} = 1, \\
&\qquad \exists x'.\, (ls(\underline{n}_1; x', x) * ls(\underline{n}_0; x, \mathsf{nil})) \wedge \underline{p} = 2 \ \}
\end{aligned}$$

Figure 5.19: The numeric program corresponding to the program from page 284 after perform branch condition annotation. The original branch conditions are given in square brackets.

5.11 Experimental Results

We have implemented the techniques described here in the tool THOR [Magill et al., 2008]. The program takes as input a file containing specifications of inductive predicates and a C language source file. The source file can optionally be annotated with function pre- and post-conditions. If pre- and post-conditions are not provided, they are inferred by the analysis (with the assumption that the heap is empty at the beginning of execution). The program is analyzed using the data structure specification provided and a numeric program is generated which can be passed to an external tool for further analysis. The numeric program can be generated in several formats, matching the input languages of various analysis tools. The most useful output format is C language source code, as many verification tools can accept C language source either directly or after some simple translation.

THOR is written in Ocaml and uses Yices [Dutertre and Moura, 2006] as the external theorem prover for discharging pure entailments. It uses the CIL [Necula et al., 2002] program analysis framework to handle parsing of the C code and to convert the input to a more regular form (e.g. eliminating switch statements by encoding them using if statements and gotos).

5.11.1 Simple Examples

Table 5.2 summarizes the experimental results of verifying safety and termination of some programs that manipulate different inductive data structures. For each program, we use THOR to produce the numeric abstraction of the original program. Then we use BLAST [Henzinger et al., 2002] and ARMC [Podelski and Rybalchenko, 2007] to verify assertion safety and ARMC-LIVE to check termination of the numeric abstraction. The results of BLAST, ARMC, and ARMC-LIVE are all consistent with the expected results and thus we only list the timing information.

Most of the programs are common data structure manipulations that involve looping, e.g. to insert an element into a binary search tree. In such cases termination is the main property of interest. The first two doubly-linked list examples require the proving of in-

			Safety		Termination
Program	Expected Result	T_{NA}	T_{BLAST}	T_{ARMC}	$T_{\text{ARMC-LIVE}}$
Doubly Linked Lists					
copy_zip	safe / terminates	4.862	0.238	7.674	31.683
iter_sum	safe / terminates	1.204	0.342	8.036	9.589
Circular Doubly-Linked Lists					
traverse	safe / terminates	1.526	0.046	0.908	1.383
delete	safe / terminates	2.245	0.068	11.138	20.204
meet	safe / diverges	0.760	0.126	1.734	0.180
Circular Linked Lists					
sum	safe / terminates	0.827	0.065	1.621	2.582
add_after	safe / terminates	1.072	0.061	4.846	12.342
add_after_loop	safe / diverges	0.997	0.065	1.945	3.364
Skip Lists					
create	safe / terminates	9.651	0.122	10.546	34.960
lift	unsafe / diverges	10.464	0.356	5.814	971.090
find_loop	safe / diverges	4.431	0.106	36.860	45.709
Binary Search Trees					
insert	safe / terminates	1.550	0.046	0.458	0.895
mem	safe / terminates	0.573	0.042	0.387	2.690

Table 5.2: Experimental results. Time is in seconds. T_{NA} represents the time required to produce the numeric abstraction. T_{BLAST}, T_{ARMC}, and $T_{\text{ARMC-LIVE}}$ represent the time taken to verify the numeric abstraction by BLAST, ARMC, and ARMC-LIVE respectively.

teger properties in order to guarantee memory safety. For example copy_zip defines a zip routine that takes in two lists and returns a list of pairs. The routine assumes that both lists have the same length and is only memory safe if this holds. The main function then calls zip with a list plus the result of a list copy operation.

Attempting to construct a standard memory safety proof for such a program fails, as we cannot show that certain memory accesses do not involve dereferencing nil. To fix this, we can take each command $A[x]$ that requires a heap cell to exist at x and replace it with "if $x \neq$ nil then $A[x]$ else abort." This yields a program where the assumption that $x \neq$ nil is available to us when we execute the command $A[x]$, but we are left with potential aborts in the code. If we can then show that abort is unreachable, by running a safety checker on the numeric program we generate, then we will have shown memory safety of the original program. Essentially, we have used the error operation represented by abort to capture a class of memory errors (those that result from dereferencing nil). The copy_sum and iter_sum examples are both based on taking this approach to proving memory safety.

5.11.2 Complex Examples

We have also run some experiments involving more complicated data structures and algorithms. These were chosen as motivating examples for work on circuit translation [Cook et al., 2009a] that requires, as a first step, the computation of a bound on the amount of memory allocated by a program. To compute this bound, we take a program and replace instances of alloc$(f_1, \ldots, f_n)$ with the command alloc$(f_1, \ldots, f_n)$; $mem := mem + 1$. We also replace free x with free x; $mem := mem - 1$. If we initialize mem to 0 at the beginning of the program, then mem will always be a count of the number of memory cells currently allocated in the heap.

We can then ask a tool for computing bounds on integer variables to give a bound on mem in terms of the program inputs. For example, a program that reads in n integers may store these values in a list, allocating n heap cells in the process. If it performs some sorting of this list, it then might use auxiliary storage, which we can also bound in terms of n. Generating a numeric program that captures the connection between the integer n

that is input and subsequent data structure allocations and transformations is the key to obtaining such bounds.

Priority Queue This example repeatedly reads inputs, inserting them into a sorted list. It then outputs the list in sorted order.

Merge Sort This example implements a merge of two sorted sequences.

Packet Sorting This example processes pairs of identifiers and data. The program reads in a list of identifier, data pairs and filters them as they are read to ensure that if a duplicate identifier is encountered, the data is discarded. Once it has read in a certain number of unique elements, it sorts them according to identifier and then outputs the sorted list. This example mimics the behavior of a simple network device, which would use a similar setup to process network packets.

Dictionary This example uses a binary search tree to implement a dictionary.

Huffman Encoder This example implements the Huffman encoding algorithm. It reads in a list of symbols paired with their frequency. It builds a list of one-element trees using this data. It then repeatedly merges the two trees in the list with the lowest frequencies, assigning the sum of their frequencies to the resulting tree. The building phase finishes when the list contains a single tree. The program then processes queries, repeatedly reading symbols from the input and outputting the binary string corresponding to the encoding of that symbol.

Results Table 5.3 lists the results from this set of experiments. In each case, the bound on allocated memory in terms of input sizes is listed along with the number of lines of code in the example.

Program	Bound	LOC
merge	$8 * n_1 + 8 * n_2$	80
prio	$8 * n$	56
packet	$12 * n + 8$	95
huffman	$52 * n - 12$	202
bst_dict	$24 * n$	142

Table 5.3: Heap bounds and lines of code.

Numeric programs were produced for all examples and bounds were inferred by the bounds inference algorithm for all examples except huffman. In this case, the numeric program was too large for the bounds analysis tool, indicating a need for better methods of simplifying the numeric abstraction and eliminating unnecessary instrumentation variables.

5.11.3 Summary and Challenges

Our implementation demonstrates the viability of this approach for reasoning about safety and liveness of heap-manipulating programs. However, there are still issues to be solved before such an approach can scale to large programs. The biggest issue is the size of the numeric programs that are generated. The algorithm presented in this dissertation and implemented in THOR produces a number of temporary variables that could potentially be eliminated, either with a post-processing pass or during the instrumentation process. Extra variables generally degrade performance of the analysis tools that we run on the numeric programs. Finding a general method for eliminating these temporary variables is ongoing work.

Another contributor to the size of the numeric program is the disjunction and subsequent extra branching that is introduced by the analysis. This is hard to avoid, as much of it is needed for the memory safety proof. Better abstraction procedures and better abstract domains that benefit shape analysis also provide an immediate benefit to an algorithm such as the one in THOR, which is heavily based on these techniques. A smaller state space during the memory safety proof translates directly to a smaller numeric program. Much progress has been made in terms of abstract domains for shape analysis that permit more concise proofs of memory safety [Yang et al., 2008], so we are optimistic that there is room for improvement in numeric program size based on these techniques.

It may also be worth investigating whether performing additional abstraction on the numeric program would help with these issues. For example, abstract interpretation methods could possibly be used to simplify the update relations involved. Such investigations are left to future work.

Chapter 6

Related Work

We now present some background material and describe existing work in the area of static analysis for heap-manipulating programs, termination proving of such programs, and translations from heap programs to numeric programs.

6.1 Approaches to Analyzing the Heap

First, we will discuss various approaches to reasoning about imperative programs that manipulate the heap and highlight the advantages that separation logic provides over previous methods.

Alias Analysis The simplest static analysis for programs that use the heap is an alias analysis [Shapiro and Horwitz, 1997b, Landi and Ryder, 1992]. These analyses fall into the general category of data-flow analysis and originate from the compiler community. At each program point, a set of equivalence classes is computed. Depending on the analysis, these equivalence classes either represent variables that *must alias* or those that *may alias* [Deutsch, 1994]. This information is useful for code optimization, but also when doing program verification. For example, consider the sequence of commands `[x] = 3; [y] = 4`, where we use brackets to indicate dereferencing. This results in a state where $(x = y) \wedge y = 4$ if x and y must alias. If they are known to not alias, it results in $x = 3 \wedge y = 4$. And if they may alias, we must consider that both possibilities could hold. That is, the postcondition would be $(x = y \wedge y = 4) \vee (x = 3 \wedge y = 4)$. In general, if n variables may alias, we must consider 2^n cases (in each case assuming that a distinct subset of the variables alias). This quickly becomes intractable even for small n. And n is generally not small, particularly when dynamic allocation and deallocation are involved [Shapiro and Horwitz, 1997a]. It should be noted that the imprecision of alias analysis is not a problem for compiler transformations. If the alias analysis results are too imprecise to be useful, the compiler simply forgoes any alias-based optimizations it would otherwise apply. Thus, for compiler optimizations, it provides a good tradeoff between usefulness of results and analysis time.

Shape Analysis Shape analysis is the next step up in precision for the analysis of programs that manipulate the heap. Rather than tracking alias sets of variables, it tracks invariants of pointer structures. For example, in the case of doubly-linked lists, a shape analysis would check the fact that if the forward link of memory cell a points to cell b, then the back link of cell b points to cell a. Shape properties also encompass heap reachability properties. Continuing with the example of linked lists, we might want to track whether the list is null-terminated. That is, whether a cell holding the value *null* is reachable from the head of the list by following "next" pointers.

TVLA One of the most thoroughly-studied shape analysis frameworks is TVLA (Three-Valued Logic Analysis) [Sagiv et al., 2002]. As the name suggests, it is based on using a three-valued logic to represent abstract states. More specifically, the logical foundation consists of first-order logic with transitive closure. The set of individuals corresponds to the set of heap cells, and unary predicates are used to record which cell a stack variable points to. So, for example, if x and y are pointer-valued variables in the program, we would have two predicates p_x and p_y. If x and y alias, then this situation would be represented by the formula $\exists c.\ p_x(c) \wedge p_y(c)$. Fields are represented by binary predicates, $f(a, b)$, where f is the field name, a is a memory cell with field f, and b is the cell pointed to by the f

field of a (or equivalently, b is the value stored in the f field of a). So if x is a pointer to a record that contains a *next* field, and the next field points to the same memory location as y, this would be written $\exists c, d.\ p_x(c) \wedge \textit{next}(c, d) \wedge p_y(d)$. The analysis itself uses models rather than formulas to represent the program state at each point. The effect is the same in that abstract states in both approaches represent sets of concrete states.

Shape Analysis Based on Separation Logic As part of this thesis, I present a shape analysis based on separation logic, which we originally described in [Magill et al., 2006]. Similar analyses have also been presented in [Distefano et al., 2006] and [Chang et al., 2007]. Significant advances to the style of analysis we utilize are present in [Berdine et al., 2007] and [Calcagno et al., 2009]. Berdine et al. [2007] give a framework with support for inferring the predicates necessary to describe higher-order structures, such as lists-of-lists. Calcagno et al. [2009] give a procedure for using *bi-abduction* to infer not only invariants and post-conditions, but also preconditions. This helps to eliminate the need for any programmer-supplied annotations.

Other work includes [Chang et al., 2007], which gives a shape analysis framework that allows data structures to be defined by routines for checking their structural invariants. Chang et al. have extended their approach to support numeric invariants of data structures Chang and Rival [2008], but not via reduction to numeric programs. [Guo et al., 2007] give a method of automatically inferring the appropriate inductive definitions based on the code being analyzed. Finally, Distefano and Parkinson [2008] give a shape analysis with support for user-provided rewrite rules, although the rules are not automatically generated from inductive definitions, as they are in our implementation.

There has also been previous work on extending shape analysis with support for tracking integer properties. Calcagno et al. handle the case where arithmetic is allowed in the domain of the heap Calcagno et al. [2006]. For approaches based on TVLA, there is the work of Beyer et al. Beyer et al. [2006]. Rugina develops an analysis targeting balance properties of tree-shaped data structures Rugina [2004]. Nguyen et al. present a verification condition-based procedure that can handle shape plus size properties when loop invariants and pre- and post-conditions are provided Nguyen et al. [2007]. However, none of these use the method described here of generating numeric programs as an intermediate step in the verification process.

Relation with TVLA There are some similarities between these approaches and TVLA. For example, they can all be described using the framework of abstraction interpretation. Also, their approach to abstraction is similar in that they all have operations that can be seen as folding and unfolding of an inductive definition of the data structure. However, there are marked differences as well. In TVLA, one describes a data structure by stating a number of properties of that structure. For example, a list is defined in terms of the basic predicates for stack variables and field dereference plus reachability and cyclicity. Reasoning about doubly-linked lists requires the addition of predicates relating dereferences of "forward" and "back" fields. In the shape analysis based on separation that we presented as part of this thesis, the data structure as whole is defined inductively. We believe this allows for a more straightforward definition from the user's point of view.

On the other hand, there are also advantages to the TVLA approach. Because it tracks individual data structure properties, rather than descriptions of specific structures, it is more general than the approach followed in our work. When faced with a data structure that was not considered when defining the instrumentation predicates, it may still be able to provide some information.

Another notable difference between the two approaches is in their treatment of disjoint data structures. In TVLA, two structures that do not overlap are described by explicitly stating that elements in one are not reachable from elements in the other. The treatment based on separation logic has support in the logic for expressing disjointness, but no explicit support for expressing reachability (instead, reachability information is implicitly encoded in the inductive definitions we use for data structures). Taking disjointness as a fundamental property allows for local reasoning, which has advantages in terms of scalability of the analysis.

6.2 Termination Proving

Termination proving for heap-manipulating programs has been described in Loginov et al. [2006a] and Podelski et al. [2008]. Both of these approaches utilize a different shape analysis framework and Loginov et al. [2006a] does not involve the production of numeric abstractions, instead incorporating a rank-finding algorithm directly in the analysis.

The work in Podelski et al. [2008] does involve the production of numeric abstractions, but they are produced from counter-example traces generated by the termination analysis and used to communicate with the heap analysis, which is run only on-demand. By contrast, we convert an entire program to a numeric abstraction before doing any termination analysis, which permits a looser coupling between the termination tool and the shape analysis tool.

In Brotherston et al. [2008a], Brotherston et al. give a method of showing termination of programs using separation logic, based on the notion of cyclic proofs. However, they do not give a static analysis capable of automatically generating these proofs. It is also not clear that such an approach can handle cases where more complicated termination arguments, such as lexicographic orderings, are needed.

In Berdine et al. [2006] a method is presented for using a separation logic shape analysis to prove termination. However, that work is tied to a specific rather weak abstract domain for tracking size changes. The approach described here is able to obtain much more precise information by tracking the actual change in data structure size rather than only the presence and direction of change.

The closest work to ours is that of Boujjani et al. Bouajjani et al. [2006] which gives a bi-simulation between programs manipulating singly-linked lists and counter automata and Habermehl et al. Habermehl et al. [2007] which provides a termination result for trees by relating tree-manipulating programs to tree automata. By focusing on specific data structures, these papers are able to obtain very precise results. In our work, we obtain a simulation result rather than bi-simulation, but the result holds of arbitrary inductively-defined data structures.

6.3 Program Logics

In this section we discuss related work in logics for reasoning about programs and, in particular, logics with a notion of auxiliary variables, logics designed to relate two programs, and logics designed for goto languages.

Auxiliary Variables Our instrumentation variables are similar in usage to auxiliary variables in Hoare logic [Owicki and Gries, 1976]. Both auxiliary variables and instrumentation variables are not permitted to affect the values of the original variables nor the control flow of the original program. However, deciding whether one program has been derived from another by the addition of auxiliary variables is a purely syntactic operation. Our rules for placing commands involving instrumented variables are based in part on the invariant that holds at the point where the command is being added. The process of instrumenting a program can also change the structure of the code by inserting or removing branches. As such, there is not a simple syntactic relationship between the two programs. Our treatment of existential quantifiers also differentiates our work as mentioned above and in Chapter 4. By virtue of the fact that we are relating two programs and focusing on simulation as the defining concept for soundness, we obtain rules that relate existential quantification to nondeterministic assignment and disjunction to nondeterministic choice in a novel way.

History Variables History variables Abadi and Lamport [1988] are a generalization of auxiliary variables. An augmented transition system is obtained from an original transition system via the addition of history variables if the systems satisfy properties H1-H5 in Abadi and Lamport [1988], the first four of which informally correspond to the following.

H1. The state space of the augmented system consists of the state space of the original plus the addition of some new variables.

H2. Initial states in the original system and augmented system agree on the values of the original variables.

H3. If the augmented system takes a step, and we project out the new variables, then this corresponds to a step in the original system.

H4. The augmented system can simulate any step of the original system.

The condition H5 specifies how fairness constraints for the properties of these systems should be related, and we omit it here since it does not constrain the transition systems.

In this thesis, we have proved H1, H2, and H4 for our instrumented programs. We do not give a formal treatment of H3 for instrumented programs, though we conjecture that it holds. In either case, clearly our instrumented variables have much in common with history variables.

If H3 holds, one could view our theory of instrumented programs as giving a particular method of adding history variables to heap-manipulating programs using separation logic annotations to guide the process. As with auxiliary variables, the connection between added variables and existential quantification in the separation logic formulae is novel. The conditions above on history variables give another clue as to why such a connection is reasonable. Existential quantification is, in a sense, the logical analogue of the projection operation referenced in H3 and H4.

Relating Programs The concept of relating two programs at different levels of abstraction is used heavily in the area of program refinement [Wirth, 1971]. However, the goal of our work, and thus the approach, is different. In program refinement, the goal is typically to start from a high-level description of the program and produce successively lower-level refinements until a concrete implementation is reached. By contrast, our goal is to take a concrete implementation and produce a more abstract version. Furthermore, the relation between the two programs in our approach is looser than would generally be acceptable in a program refinement context. This is motivated and justified by our goal of passing the numeric abstractions to automated program verification tools.

Another approach to relating programs, based on a relational version of Hoare logic, is given in [Benton, 2004]. The goal is to relate two programs when their total correctness properties are the same. In our work, since we are only concerned with obtaining an over-approximation of the original program, the numeric program may diverge in cases where the original program terminates. We also are able to get by with a logic where the annotations represent sets of states rather than relations. Indeed, the main goal of our work is to offload the relational reasoning to separate analysis tools.

Yang [2007] gives a relational logic like Benton's for separation logic and uses it to prove that the Schorr-Waite graph marking algorithm is equivalent to a depth-first traversal. This approach differs from ours in that we are only concerned with preserving properties of the stack variables, whereas the logic Yang presents tracks relations between heaps as well. The other main difference is that we are focused on a logic that can be automated and a means of automating it, whereas the logic in [Yang, 2007] is currently only suitable for by-hand proofs.

Our treatment of existential quantifiers is also a key difference between this work and other work in logics for relating programs. Because we state soundness in terms of simulation, we are able to use the EXISTS rule, which is explained in Chapter 4, Figure 4.1 to insert and update variables representing values that are quantified in the original program proof. We thus obtain information about how quantified values change without resorting to relational invariants.

Verification of Goto Languages Clint and Hoare Clint and Hoare [1972] present a logic for functions that can be interrupted by goto. Here the idea is already present of viewing "goto" as a special type of function that is known to never return. This is essentially the same as our treatment, where gotos are viewed as executing a continuation. The proof system that Clint and Hoare develop handles the goto construct by allowing the program prover to assume that the triple $\{Q\}$ $\mathsf{goto}\ l$ $\{\mathsf{false}\}$ holds of any goto statement, where Q is a precondition associated with label l. In this thesis, we note the redundancy of the post-condition for a goto statement and instead work solely with preconditions. A more significant difference exists in the general approach of Clint and Hoare [1972] versus the approach taken here. Clint and Hoare view gotos as exceptional cases in an otherwise well-structured program. We instead view gotos as the main control flow construct and provide no support for structured control constructs such as while loops. This has the advantage of making the treatment extremely uniform. Arbib and Alagic [1979] and de Bruin [1981] also present similar systems for proving partial correctness of goto programs and note the connection to continuations.

Chapter 7

Conclusion

In this thesis work, we have done the following

1. Developed a *logic of instrumentation* for relating a heap-manipulating program to a numeric abstraction, which tracks how numeric properties of the data structures are changing.
2. Developed a static analysis algorithm that generates numeric abstractions, the soundness of which is justified using the logic of instrumentation.
3. Implemented the static analysis and used this implementation to prove properties of programs of various sizes and operating over various data structures.

We now discuss each of these items in turn, summarizing our contributions and remaining future work in each area.

7.1 Logic of Instrumentation

The logic we developed in Chapter 4 gives a program proving method based on adding additional variables to the program. The basic judgment in the logic relates a program to an instrumentation of that program. This instrumentation consists of the commands from the original program plus some additional commands and branches involving new variables not present in the original program.

This proof system is adapted to proving properties preserved by simulation and thus has a different character than most traditional logics based on pre- and post-condition reasoning. In particular, the simulation-based view of verification has led us to elevate nondeterminism to a more prominent role. We obtain proof rules that use nondeterministic choice in the language to encode disjunctions from the logic and which use nondeterministic assignment to capture existential quantification.

The logic is proved sound where the notion of soundness is that if two programs are related by the logic, then a simulation relation exists between them. The direction of simulation is such that the instrumented program is an abstraction of the original program and the notion of simulation is stuttering simulation. This implies that all LTL$\backslash$X properties that hold of the instrumentation also hold of the original program. We define a version of LTL$\backslash$X where the state properties can contain separation logic formulae. These formulae are then shown to be invariant under stuttering equivalence and thus respect stuttering simulation.

Future Work We only considered the soundness question in the work presented here. A remaining open question is what can be attained in terms of completeness. There are many possible questions to investigate here. Bouajjani et al. [2006] obtain a bi-simulation result for list programs and counter automata, implying that our logic of instrumentation or something similar could potentially be shown complete for this class of programs. It would also be interesting to investigate completeness results that are relative to completeness of the underlying shape analysis.

The instrumentation variables which we add when constructing Instrumented programs function similarly to auxiliary variables Owicki and Gries [1976], but are less restricted in their interactions with existing program variables and control flow. Such variables may be useful in other situations where auxiliary variables are used, such as in proofs of parallel programs.

Finally, considering under-approximations would provide a means of proving non-termination and other properties that are existentially quantified over paths. Combined these could potentially allow the sound handling of a more expressive temporal logic such as CTL*.

7.2 Analysis Algorithm

We also presented an automated analysis based on the logic just described. This corresponds to a restricted subset of the derivations in the logic of instrumentation and its soundness is justified by showing that a derivation in this logic exists for every output returned by the analysis.

The analysis is based on a shape analysis that uses separation logic to represent abstract states. In the process of describing how to automatically add instrumentation commands, we also show how we can automatically obtain shape invariants for data structures.

Our analysis accepts user-provided descriptions of inductive data structures and uses these during the shape analysis and instrumentation process. By altering these description files, the user can add support for new inductive data structures or change the notion of size that is tracked by the instrumentation variables.

Future Work The numeric programs that are produced by the automated analysis can sometimes be quite large. However, generally a much shorter proof is possible according to the logic presented in the first part of the thesis. Adding optimizations and simplification passes to the analysis in order to have it produce a numeric program closer to the short program that a human can often discover is an ongoing challenge. That this issue arises is not surprising since the same issue arises with shape analysis using separation logic. In that case, the invariants discovered automatically are often more complex than those discovered by hand and finding better abstract domains that permit the discovery of these simpler invariants has clear benefits in terms of scalability of the approach. Much progress has been made in this direction for the pure shape analysis problem [Yang et al., 2008], so we are optimistic that similar improvements may be possible for instrumentation analyses.

7.3 Implementation

We implemented the analysis algorithm described above and ran experiments involving a number of programs over a variety of data structures, including composite data structures such as lists of trees. The implementation analyses C code and generates a new C language program that is a numeric abstraction of the input. Support for various data structures is implemented by defining a language of inductive specifications, which describe inductive properties of the data structures. For example, a description of a doubly-linked list would specify that it can be unfolded from the front or the back and that the concatenation of two list segments is also a list.

The implementation is written in Ocaml and uses CIL to parse the C code provided as input. Yices is used to prove pure entailments and an implementation of the frame inference procedure described in Section 5.5.3 is used to reason about spatial formulae. A number of optimizations and command line options affecting analysis behavior have been incorporated into the implementation in order to efficiently handle a larger set of programs.

Future Work A great deal of implementation efficiency comes down to heuristics. For example, quick checks that indicate an implication is not provable, and save the time required to do a full proof search, can significantly program decrease analysis time. Heuristics for generating abstraction patterns from inductive specifications and choosing good points at which to apply abstraction are also important. For example, suppose we have an inductive definition for a list segment and are analyzing a loop that generates a null-terminated list at x. We could perform abstraction once we have a single points-to $x \mapsto [\mathsf{next} : \mathsf{nil}]$ or we could wait for a pair of points-to predicates $\exists z.\ x \mapsto [\mathsf{next} : z] * z \mapsto [\mathsf{next} : \mathsf{nil}]$. Choosing the first option results in shorter analysis times, but sometimes prevents programs from being proved memory safe that could be proved by taking the second approach of waiting longer before performing abstraction.

Similarly, when analyzing programs that call non-recursive functions, these functions can be inlined and the program treated as if it were written as a single large function. Alternatively, we can view function call sites as an opportunity to apply abstraction, which simplifies the symbolic static formulas at that call site, but may result in too much information loss and a failure to prove memory safety.

Currently, we choose a reasonable default for these options and provide command-line flags that allow the user to alter the behavior of the analysis. One approach that may provide a better solution would be to incorporate counter-example guided abstraction refinement [Clarke et al., 2003]. This technique, which originated in the software model checking community, is based on the idea of performing abstraction as aggressively as possible but providing a means of backtracking and keeping more precise information if this abstraction is found to cause problems.

While the frequency of calls to abstraction has a large effect on the running time of the analysis, the actual abstraction function used is at least as important. We have chosen a relatively simple abstraction function for our implementation and exploring other options from the literature may provide additional improvements. For example, in [Yang et al., 2008], an abstraction function is described that provides predicates for empty, non-empty, and possibly-empty lists. While only one of these predicates is needed to reason about list programs, including all of them allows for a fairly precise abstraction function that still results in the small state space sizes that are usually associated with coarser abstraction functions. In [Chang et al., 2007] an abstraction function is described that uses the symbolic execution history to guide the abstraction process. The current symbolic state is compared to the symbolic state obtained during the previous iteration of a loop and this combined information is used to guide abstraction.

It should be possible to incorporate techniques such as these into our instrumentation analysis in order to further improve performance.

Appendix A

Guide to Notation

A.1 Programs, States, and Transition Systems

a	The type of variables and expressions denoting addresses.
i	The type of variables and expressions denoting integers.
τ	An arbitrary type. Either a or i.
x^τ	Variable of type τ. Figure 2.1, page 16.
e^τ	Expression of type τ. Figure 2.1, page 16.
c	Command. Figure 2.1, page 16.
k	Continuation. Figure 2.1, page 16.
P	Program. Figure 2.1, page 16.
$fv(t)$	Free variables in some term t (t can be an expression, command, continuation, program, logical formula, etc. Definitions 2, 3, and 2.2.1.
Values	The set of *values*. Page 15.
Stores	The set of *stores*. Page 15.
Records	The set of *records*. Page 16.

Heaps	The set of *heaps*. Page 16.
v	An element of *Values*. Page 15.
s	An element of *Stores*. Page 15.
h	An element of *Heaps*. Page 16.
(s, h)	Memory State. A store, heap pair.
$\llbracket e \rrbracket$	Denotation of expression e. A function from *Stores* to *Values*. Figure 2.2, page 18.
$\llbracket c \rrbracket$	Denotation of command c. A function from *Stores* × *Heaps* to $2^{\mathit{Stores}\times\mathit{Heaps}\cup\{\mathbf{error}\}}$. Figure 2.3, page 115.
G	Set of *execution states*. Page 24.
γ	An element of G. Page 24.
$\rightsquigarrow$	Transition relation for continuations. A subset of $G \times G$. Figure 2.4, page 115.
$\xrightarrow[P]{}$	Transition relation for programs. A subset of $G \times G$. Definition 13, page 115.
S	Transition System. A tuple of the form $(A, I, F, \dashrightarrow)$. Definition 11, page 47.
T	A trace of a transition system. Definition 12, page 48.
traces(S)	The set of traces of transition system S. Definition 48, page 48
$(\!(P \mid Q_0)\!)$	The transition system corresponding to program P with precondition Q_0. Definition 14, page 48.

A.2 Relations

R	An arbitrary relation.
E	An equivalence relation.
R^+	The transitive closure of relation R. Definition 16, page 49.

$s =_V s'$ s and s' agree on the values of variables in V. Definition 1, page 17.

$\gamma \doteq \gamma'$ The execution states γ and γ' agree on all but the current continuation. Page 89.

$\gamma =_V \gamma'$ The execution states γ and γ' include the same heap and their stores are $=_V$-related. Definition 23, page 91.

$\gamma \overset{\mathrm{s}}{=}_V \gamma'$ The execution states γ and γ' have stores that are $=_V$-related. Their heaps are not required to be the same. Definition 24, page 93.

A.3 Separation Logic

$p^{\vec{\tau}}$ An inductive predicate name with arity $\vec{\tau}$. Also written as p when the arity is clear from context. Figure 2.6, page 27.

ρ A record expression. Figure 2.6, page 27.

Ξ A spatial predicate. Figure 2.6, page 27.

Q A separation logic formula. Figure 2.6, page 27.

$[\![\rho]\!]$ The denotation of record expression ρ. A mapping from *Stores* to *Records*.

$(s, h) \models_X Q$ The memory state (s, h) satisfies separation logic formula Q given inductive predicate meanings X. Figure 2.7, page 28.

$(s, h) \models Q$ The memory state (s, h) satisfies separation logic formula Q. Used when the set of inductive predicate meanings X is clear from context or otherwise unnecessary (all of the technical development is independent of the particular choice of X).

A.4 LTSL

LTSL^E The set of E-invariant LTSL formulae. Definition 25, page 94.

LTSLV The set of LTSL formulae containing only variables in the set V. All these formulae are $\sim_{=_V}$-invariant. Definition 26, page 98.

LTSLPV The set of LTSL formulae containing only pure state formulas over variables in the set V. All these formulae are $\sim_{\overset{s}{=}_V}$-invariant. Definition 27, page 99.

$\boxed{\exists}(V', \phi)$ The function on LTSL formulae defined in Figure 3.7 on page 103

$S_1 \underset{\sim}{\sqsubset}_{R,E} S_2$ S_2 E-stuttering simulates S_1 and R is the simulation relation witnessing this. Definition 29, page 119.

$\mathbf{T}_1 \lesssim_E \mathbf{T}_2$ $\mathbf{T}_2$ E-stuttering contains $\mathbf{T}_1$. Definition 21, page 86.

$\mathbf{T}_1 \approx_E \mathbf{T}_2$ $\mathbf{T}_1$ and $\mathbf{T}_2$ are E-stuttering equivalent. Definition 21, page 86.

Appendix B

Pseudo-code

We use an ML-like pseudo-code when describing our algorithms. The type system includes the standard type constructors for tuples and option types. We also assume a "set" type exists and use standard set notation to describe values of set type. The main language constructs are **match**, **let**, and **return**.

return simply returns the value following it. So **return** 1 returns the integer value 1. **match** examines a value and executes different code depending on the form of the value. For example, the code below returns 1 if c is an assignment statement or 2 if it is an allocation.

```
match $c$ with
  case $x := e$
    return $1$
  case $x := \mathsf{alloc}(\ldots)$
    return $2$
end
```

The **let** command is used to introduce binding an perform pattern matching. The command **let** $e_1 = e_2$ **in** pattern matches e_2 against e_1, introducing bindings if the match succeeds. If the match fails, the **match failed** clause is executed. The code below returns $\mathsf{Some}(x)$ if the continuation k starts with an assignment to x and returns None otherwise.

```
let $x := e$; $k' = k$ in
  return Some$(x)$
match failed ⇒ return None
```

Finally, we note that **let** statements can be sequenced and let bindings of the form $x = t$ where x is a variable and t is an arbitrary term can never fail (since they involve no pattern matching. Also, functions can be recursive. As an example, the code in Figure 9 converts all assignment statements into non-deterministic assignments in the continuation k.

Function make_nondet(k). Pseudo-code example. Converts assignment statements into non-deterministic assignments to the same variable.

```
match $k$ with
  case $c$; $k'$
    let $(x := e) = c$ in
    let $k''$ = make_nondet($k'$) in
      return $(x := ?; k'')$
    match failed ⇒
      let $k''$ = make_nondet($k'$) in
        return $(c; k'')$
  case branch $e_1 \Rightarrow k_1, \ldots, e_n \Rightarrow k_n$ end
    let $k'_1$ = make_nondet($k_1$) in
      ⋮
    let $k'_n$ = make_nondet($k_n$) in
      return branch $e_1 \Rightarrow k'_1, \ldots, e_n \Rightarrow k'_n$ end
  case goto $l$ return goto $l$  case halt return halt  case abort return abort
end
```

B.1 Local Functions

We will also occasionally define functions that are local to the primary function being presented in a figure. The syntax for this is as below, where localfun is the name of the local function begin defined.

```
fun localfun(args) =
  body of local function
in
  body of primary function
```

Bibliography

Martn Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1988. 6.3

Michael Arbib and Suad Alagic. Proof rules for gotos. *Acta Informatica*, pages 139–148, 1979. 6.3

T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, pages 203–213. ACM Press, 2001. 1

Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *In POPL*, pages 14–25. ACM Press, 2004. 6.3

J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, LNCS 4590, pages 178–192. Springer, 2007. 3.3.3, 5, 6.1

Josh Berdine. personal communication, 2006. 2.4.2

Josh Berdine, Cristiano Calcagno, and Peter O'Hearn. A decidable fragment of separation logic. In *In FSTTCS*, pages 97–109. Springer, 2004. 1, 5.5.1

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, pages 52–68. Springer, 2005. 5.5.3

Josh Berdine, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *CAV*, pages 386–400. Springer, 2006. 1.1, 6.2

D. Beyer, T. A. Henzinger, and G. Théoduloz. Lazy shape analysis. In *CAV*, LNCS 4144, pages 532–546. Springer, 2006. 6.1

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, London, UK, 1999. Springer-Verlag. ISBN 3-540-65703-7. 5.9

A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV*, LNCS 4144, pages 517–531. Springer, 2006. ISBN 3-540-37406-X. 6.2, 7.1

M. Bozga, P. Habermehl, R. Iosif, F. Konecny, and T. Vojnar. Automatic verification of integer array programs. In *Computer Aided Verification*, 2009. 2.4.3

Marius Bozga, Radu Iosif, and Swann Perarnau. Quantitative separation logic and programs with lists. In *IJCAR '08: Proceedings of the 4th international joint conference on Automated Reasoning*, pages 34–49, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-71069-1. doi: http://dx.doi.org/10.1007/978-3-540-71070-7_4. 1

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. The polyranking principle. In *Proc.* 32^{nd} *International Colloquium on Automata, Languages and Programming*, volume 3580 of *Lecture Notes in Computer Science*, pages 1349–1361. Springer Verlag, 2005a. 1.1

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination analysis of integer linear loops. In Martin Abadi and Luca de Alfaro, editors, *Proc. 16th Intl. Conference on Concurrency Theory (CONCUR)*, volume 3653 of *Lecture Notes in Computer Science*, pages 488–502. Springer Verlag, August 2005b. 1.1

J. Brotherston. Formalised inductive reasoning in the logic of bunched implications. In *SAS*, LNCS 4634, pages 87–103. Springer, 2007. ISBN 978-3-540-74060-5. 2.2.2

J. Brotherston, R. Bornat, and C. Calcagno. Cyclic proofs of program termination in separation logic. In *POPL*, pages 101–112. ACM, 2008a. 6.2

James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. *SIGPLAN Not.*, 43(1):101–112, 2008b. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1328897.1328453. 1.1

M. C. Browne, E. M. Clarke, and O. Grümberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1-2):115–131, 1988. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(88)90098-9. 3

C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *SAS*, LNCS 4134, pages 182–203, 2006. 6.1

Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Context logic and tree update. *SIGPLAN Not.*, 40(1):271–282, 2005. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1047659.1040328. 4.2

Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *SIGPLAN Not.*, 44(1):289–300, 2009. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1594834.1480917. 1.1, 5.10, 6.1

B.-Y. E. Chang, X. Rival, and G. C. Necula. Shape analysis with structural invariant checkers. In *SAS*, LNCS 4634, pages 384–401. Springer, 2007. 1.1, 5.7.4, 6.1, 7.3

Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *POPL*, 2008. 5.7.3, 5.7.4, 6.1

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50:752–794, September 2003. 7.3

Edmund M. Clarke and Bernd-Holger Schlingloff. Model checking. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 1635–1790. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9, 0-262-18223-8. 3.3

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, January 1999. ISBN 0262032708. 3, 3.1, 3.3

M. Clint and C.A.R. Hoare. Program proving: Jumps and functions. *Acta Informatica*, pages 214–224, 1972. 6.3

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 415–426, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: http://doi.acm.org/10.1145/1133981.1134029. 1, 1.1

Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. Proving conditional termination. In *CAV '08: Proceedings of the 20th international conference on Computer Aided Verification*, pages 328–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70543-7. doi: http://dx.doi.org/10.1007/978-3-540-70545-1_32. 1.1

Byron Cook, Ashutosh Gupta, Stephen Magill, Andrey Rybalchenko, Jiri Simsa, Satnam Singh, and Viktor Vafeiadis. Finding heap-bounds for hardware synthesis. In *FMCAD'09*, 2009a. 1.3, 5.11.2

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. CFL-termination. Technical report, Microsoft Research, 2009b. 1.1

P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY. 5, 5.7

P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY. 2.4.3

P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTREÉ analyzer. In *ESOP*, pages 21–30, 2005. 1

Arie de Bruin. Goto statements: Semantics and deduction systems. *Acta Informatica*, pages 385–424, 1981. 6.3

Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *PLDI '94*, pages 230–241, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. doi: http://doi.acm.org/10.1145/178243.178263. 6.1

D. Distefano and M. J. Parkinson. jStar: towards practical verification for Java. In *OOPSLA*, pages 213–226. ACM, 2008. 3.3.3, 6.1

D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, LNCS 3920, pages 287–302. Springer, 2006. 1.1, 5, 5.7.1, 6.1

Bruno Dutertre and Leonardo De Moura. The YICES SMT Solver. Technical report, SRI International, 2006. 5.5.1, 5.11

J. Giesl, P. Schneider-Kamp, and R. Thiemann. Aprove 1.2: Automatic termination proofs in the dependency pair framework. In *Proceedings IJCAR '06*, LNAI 4130, pages 281–286. Springer, 2006. 1.1

Denis Gopan, Thomas Reps, and Mooly Sagiv. A framework for numeric analysis of array operations. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 338–350, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X. doi: http://doi.acm.org/10.1145/1040305.1040333. 2.4.3

Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. Thread-modular shape analysis. In *PLDI*, pages 266–277, New York, NY, USA, 2007. ACM. 3.3.3

Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 127–139, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. doi: http://doi.acm.org/10.1145/1480881.1480898. 1, 1.3

B. Guo, N. Vachharajani, and D. I. August. Shape analysis with inductive recursion synthesis. *SIGPLAN Notices*, 42(6):256–265, 2007. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1273442.1250764. 6.1

P. Habermehl, R. Iosif, A. Rogalewicz, and T. Vojnar. Proving termination of tree manipulating programs. In *ATVA*, LNCS 4762, pages 145–161. Springer, 2007. ISBN 978-3-540-75595-1. 6.2

Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 339–348, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: http://doi.acm.org/10.1145/1375581.1375623. 2.4.3

T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70. ACM Press, 2002. 1, 5.11.1

William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 235–248, New York, NY, USA, 1992. ACM Press. ISBN 0-89791-475-9. doi: http://doi.acm.org/10.1145/143095.143137. 6.1

A. Loginov, T. W. Reps, and M. Sagiv. Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In *SAS*, LNCS 4134, pages 261–279. Springer, 2006a. 6.2

Alexey Loginov, Thomas Reps, and Mooly Sagiv. Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In *Proc. of SAS-06 Sagiv, M.; Reps, T.; and*. Springer, 2006b. 1.1

S. Magill, A. Nanevski, E. M. Clarke, and P. Lee. Inferring invariants in separation logic for imperative list-processing programs. In *SPACE*, 2006. 1.1, 5.7.1, 6.1

S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. THOR: A tool for reasoning about shape and arithmetic. In *CAV*, LNCS 5123, pages 428–432. Springer, 2008. 5, 5.11

Panagiotis Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, University of Texas at Austin, 2001. 3.2, 3.4, 3.4, 3.4

Narciso Martí-Oliet, José Meseguer, and Miguel Palomino. Algebraic stuttering simulations. *Electron. Notes Theor. Comput. Sci.*, 206:91–110, 2008. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/j.entcs.2008.03.077. 3.2

Robin Milner. An algebraic definition of simulation between programs. In *IJCAI*, pages 481–489, 1971. 3

George C. Necula, Scott Mcpeak, S. P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *In International Conference on Compiler Construction*, pages 213–228, 2002. 2.4, 5.11

H. H. Nguyen and W.-N. Chin. Enhancing program verification with lemmas. In *CAV 2008*, LNCS 5123, pages 355–369. Springer, 2008. ISBN 978-3-540-70543-7. doi: http://dx.doi.org/10.1007/978-3-540-70545-1_34. 5.2, 5.5.1

Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape and size properties via separation logic. In *VMCAI*, pages 251–266, 2007. 6.1

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL '01: Proceedings of the 15th International Workshop on Computer Science Logic*, pages 1–19, London, UK, 2001. Springer-Verlag. ISBN 3-540-42554-3. 1.1

Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6:319–340, 1976. 4.1, 4.7, 6.3, 7.1

A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, pages 32–41. IEEE Computer Society, 2004. ISBN 0-7695-2192-4. doi: http://dx.doi.org/10.1109/LICS.2004.50. 1.1, 1.3

A. Podelski and A. Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *PADL*, LNCS 4354, pages 245–259. Springer, 2007. 1, 5.11.1

A. Podelski, A. Rybalchenko, and T. Wies. Heap assumptions on demand. In *CAV 2008*, LNCS 5123, pages 314–327. Springer-Verlag, 2008. ISBN 978-3-540-70543-7. doi: http://dx.doi.org/10.1007/978-3-540-70545-1_31. 6.2

J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002. 2.2, 5.4.3

Radu Rugina. Quantitative shape analysis. In *SAS*, pages 228–245, 2004. 6.1

M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *TOPLAS*, 2002. 5.7, 6.1

M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Static Analysis Symposium*, 1997a. 6.1

Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, New York, NY, USA, 1997b. ACM Press. ISBN 0-89791-853-3. doi: http://doi.acm.org/10.1145/263699.263703. 6.1

N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14 (4):221–227, 1971. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/362575.362577. 6.3

Hongseok Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3): 308–334, 2007. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/j.tcs.2006.12.036. 6.3

Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, pages 385–398, 2008. 1.1, 5.7.4, 5.11.3, 7.2, 7.3