AUTOMATIC MAINTENANCE
OF TRANSITIVE PROPERTIES
WITH APPLICATIONS TO SHAPE ANALYSIS

Thesis submitted for the degree of Doctor of Philosophy
by Tal Lev-Ami

This work was carried out under the supervision of
Prof. Mooly Sagiv

Submitted to the Senate of Tel-Aviv University
September 2009
Abstract

*Shape analysis* algorithms conservatively verify imperative programs and algorithms that use destructive pointer updates to manipulate dynamically allocated linked data structures. One of the challenges in developing such shape analyses is the computation of *abstract transformers*, i.e., the effect of program statements on the abstract representation of the memory. In this thesis, we explore three different approaches for computing these abstract transformers. Special emphasis is given to harnessing existing tools and analyses and adapting them to perform richer analyses.

We start by exploring the use of *first order automated theorem provers*. These powerful tools have advanced considerably in the last decade and are able to deal with complex reasoning problems. The main obstacle in using these tools is that reasoning about data structures requires an understanding of reachability within the program heap, which cannot be expressed in first order logic. We overcome this obstacle by developing a partial axiomatization of *transitive closure*, which allows us to express reachability. The axiomatization is mechanizable and allows automatically proving properties both using hand written invariants and by inferring them using static analysis, as demonstrated by experimental results. A practical obstacle in using static analysis together with first order automated theorem provers is the number of theorem prover calls required when computing the abstract transformers. We develop a technique (of general applicability) that allows to significantly accelerate this process by allowing multiple conjectures to be explored simultaneously over the same axioms using a technique we call *labelled clauses*. We implemented our technique in the SPASS theorem prover. We also explore some of the other possible uses of labelled clauses.

Next, we address the issue of constructing static analyses for which we can compute the *most precise abstract transformer*, i.e., analyses in which the abstraction determines the information loss and no further information is lost by the transformers. First, we build an abstract domain that supports analysis of various data structures including (cyclic) doubly-linked lists and trees with parent pointers. We show how to compute the most precise abstract transformers for this abstraction and demonstrate their efficiency using experimental results. Second, we develop a general methodology for constructing algorithms for *specialized shape analysis problems* (i.e., for a given set of program-type, properties and data structures). We provide algorithms for constructing the most precise abstract transformers for these problems and show their applicability on an extended subset of the data structures supported by the ad-hoc method.

Finally, we turn to the problem of analyzing programs for which the sizes of the data-
structures are important. We build a combination framework that enables a synergy between existing shape analyses and numeric domains to solve problems that require reasoning about the two kinds of information. We have implemented an instance of the framework on top of TVLA (for shape analysis) and the Parma Polyhedra Library (for numeric analysis) and have shown its effectiveness by proving properties of various algorithms including data structure implementations from the Java Collections Framework.
Acknowledgements

I would like to thank all people who have inspired me during my doctoral studies and contributed to this thesis.

First and foremost, I am deeply indebted to my supervisor Prof. Mooly Sagiv, who never ceased to push me forward on various fronts, to promote several research avenues and constantly pursue new directions. His perpetual energy and enthusiasm in research has motivated our entire department.

Neil Immerman and Thomas Reps have been wonderful research collaborators, every joint encounter resulting in new ideas and stimulating thought for continued collaborative research.

The research included in this thesis could not have been completed without the joint efforts of several leading researchers, with which I have had the honor and the pleasure to work. Working with them yielded the publications that this Thesis is built upon. Mooly Sagiv, Neil Immerman, Thomas Reps, Greta Yorsh, Siddharth Srivastava, Christoph Weidenbach, and Sumit Gulwani have all been part of joint brainstorming, late night conference calls, shared dilemmas, and exciting breakthroughs.

Ganesan Ramalingam, Sumit Gulwani, and Byron Cook, have lent their expertise and experience to assist in guiding me through challenges along the way and their advice has often been priceless.

The wonderful group of colleagues studying under Mooly Sagiv’s guidance in the Tel-Aviv University Program Analysis Lab have made the whole experience a joyful and pleasant one. Mutual assistance and deliberations made me feel I was not alone in this journey. Thank you Nurit Dor, Ran Shaham, Eran Yahav, Greta Yorsh, Noam Rinetzky, Roman Manevich, Daphna Amit, Ohad Shacham, Igor Bogudlov, Shachar Itzhaky, and Michal Segalov.

I thank the Adams Fellowship Program of the Israel Academy of Sciences and Humanities for their generous financial support, which has allowed me to devote my time to this research.

And last but not least, I would like to thank my parents for their love and support, and my wife Shira for her patience during the more difficult times of this long process.
# Contents

1 **Introduction**

1.1 Using Automated Theorem Provers for Shape Analysis .......................... 16
   1.1.1 Partial Axiomatization of Transitive Closure .............................. 16
   1.1.2 Labelled Clauses ....................................................................... 19

1.2 Specialized Shape Analysis ................................................................. 20
   1.2.1 Specialized Shape Analysis for Commonly Used Data Structures ...... 20
   1.2.2 Constructing Specialized Shape Analyses for Uniform Change ......... 22

1.3 Combining Shape Analysis Domains ...................................................... 24
   1.3.1 A Combination Framework for Tracking Partition Sizes ................. 24

1.4 Thesis Organization ....................................................................... 25

2 **Background**

2.1 Abstract Interpretation Essentials ....................................................... 27

2.2 FO(TC) Notation .............................................................................. 28
   2.2.1 Syntax ..................................................................................... 28
   2.2.2 Semantics ............................................................................... 28

2.3 Concrete Program States in Logical Shape Analysis ............................... 28
   2.3.1 Concrete Semantics .................................................................. 29

2.4 Canonical Abstraction ....................................................................... 29
   2.4.1 Embedding .............................................................................. 30
   2.4.2 Abstract Transformers for 3-valued Logical Shape Analysis ........... 31

2.5 Symbolically Computing Transformers for Shape Analysis .................... 31
   2.5.1 Symbolically Applying Transformers ........................................ 32
   2.5.2 Improvements ......................................................................... 34

2.6 Transformers for Canonical Abstraction ............................................. 35
   2.6.1 Applying Transformers ............................................................ 36

3 **Partial Axiomatization of Transitive Closure** ................................. 39

3.1 Setting ............................................................................................ 39

3.2 Axiomatization of Transitive Closure .................................................. 40
   3.2.1 Some TC-Sound Axioms .......................................................... 41
   3.2.2 Coloring Axioms ..................................................................... 43

3.3 On TC-Completeness ......................................................................... 44
   3.3.1 More About TC-Completeness ................................................ 45
   3.3.2 Nelson’s Axioms ..................................................................... 46

3.4 Heuristics for Using the Coloring Axioms ............................................ 47
3.4.1 Specification of the Reverse Procedure ........................................ 48
3.4.2 Proving Formulas using the Coloring Axioms ................................. 49
3.4.3 The Search Space of Possible Axioms ........................................... 51
3.4.4 Exploring the Search Space .......................................................... 52
3.4.5 Implementation ........................................................................... 54

4 Labelled Clauses ............................................................................... 55
4.1 Superposition with Labels ............................................................... 55
4.2 Instantiations .................................................................................. 58
  4.2.1 Multiple Labelled Conjectures ..................................................... 58
  4.2.2 Labelled Splitting ....................................................................... 61
  4.2.3 Strategies ................................................................................... 62
  4.2.4 Slicing ....................................................................................... 62
4.3 Applications in Software Verification ................................................ 63
  4.3.1 Cartesian Abstraction .................................................................. 63
  4.3.2 Shape Analysis and Canonical Abstraction ................................. 64
  4.3.3 Experiments ............................................................................. 64

5 Shape Analysis for Commonly Used Data Structures .......................... 67
5.1 Preliminaries ................................................................................... 68
  5.1.1 Notations ................................................................................... 68
5.2 Abstraction ..................................................................................... 68
  5.2.1 Data Structures ......................................................................... 69
  5.2.2 Properties of the Abstraction ...................................................... 69
5.3 Most Precise Abstract Transformers ................................................. 72
  5.3.1 Abstract Transformers ............................................................... 72
  5.3.2 Algorithms ............................................................................... 73
5.4 Evaluation ....................................................................................... 75
  5.4.1 Query Logic ............................................................................... 75
  5.4.2 Translation ................................................................................. 76
  5.4.3 Improved Translation for Evaluation .......................................... 77
5.5 Extensions ....................................................................................... 78
  5.5.1 Loop Boundaries ....................................................................... 78
  5.5.2 Partial Join ................................................................................ 78
  5.5.3 Cycles ........................................................................................ 79
  5.5.4 Parent Pointers .......................................................................... 79
5.6 Implementation ............................................................................... 80

6 Shape Analysis for Uniform Change .................................................... 83
6.1 Overview ......................................................................................... 83
6.2 Preliminaries ................................................................................... 87
  6.2.1 Programming-Language Statements ......................................... 87
  6.2.2 Monadic-Uniform Updates .......................................................... 88
  6.2.3 Canonical Abstraction ............................................................... 89
6.3 Methodology for Developing Computable Transformers ................. 90
6.4 Applications .................................................................................... 92
C  Proofs and Additional Applications for Chapter 6  159
  C.1  Proofs ................................................................. 159
  C.2  More Applications .................................................. 162
    C.2.1  Singly-Linked Lists ........................................ 162
    C.2.2  Trees .......................................................... 163
    C.2.3  No Undirected Cycles ....................................... 166
    C.2.4  Shared Trees ............................................... 167
    C.2.5  Uninterpreted Unary Relations ............................ 168
    C.2.6  Doubly-Linked Lists ....................................... 169
    C.2.7  Ordering .................................................... 169

D  Proofs for Chapter 7  171
  D.1  Pre-Order .......................................................... 171
  D.2  Lemmas used in Widening Proof ............................... 172
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Typical predicates used for representing concrete program states.</td>
<td>29</td>
</tr>
<tr>
<td>2.2</td>
<td>Predicate-update formulas for heap-manipulating statements</td>
<td>29</td>
</tr>
<tr>
<td>2.3</td>
<td>Predicates used for the Canonical Abstraction of singly-linked lists in [SRW02]</td>
<td>30</td>
</tr>
<tr>
<td>3.1</td>
<td>Heuristic for choosing initial colors and its application on reverse</td>
<td>52</td>
</tr>
<tr>
<td>4.1</td>
<td>Comparison between SPASS before and after implementation of labelled-clauses</td>
<td>65</td>
</tr>
<tr>
<td>6.1</td>
<td>Update formulas defining the concrete semantics</td>
<td>88</td>
</tr>
<tr>
<td>6.2</td>
<td>Summary of the shape-analysis problems and their feasibility-check status.</td>
<td>93</td>
</tr>
<tr>
<td>6.3</td>
<td>Monadic-uniform transformers for acyclic singly-linked lists.</td>
<td>94</td>
</tr>
<tr>
<td>7.1</td>
<td>Loop invariants for Copy and the required set constructor</td>
<td>119</td>
</tr>
<tr>
<td>7.2</td>
<td>Loop invariants for acyclic lists and the required numerical domain</td>
<td>120</td>
</tr>
<tr>
<td>7.3</td>
<td>Experimental results for the set cardinality benchmarks</td>
<td>122</td>
</tr>
<tr>
<td>C.1</td>
<td>Monadic-uniform transformers for possibly cyclic singly-linked lists.</td>
<td>164</td>
</tr>
<tr>
<td>C.2</td>
<td>Monadic-uniform transformers for trees.</td>
<td>165</td>
</tr>
<tr>
<td>C.3</td>
<td>Monadic-uniform transformers for graph with no undirected cycles.</td>
<td>166</td>
</tr>
<tr>
<td>C.4</td>
<td>Monadic-uniform transformers for shared trees.</td>
<td>167</td>
</tr>
<tr>
<td>C.5</td>
<td>Monadic-uniform transformers for doubly-linked lists.</td>
<td>169</td>
</tr>
<tr>
<td>C.6</td>
<td>Monadic-uniform transformers for ordering.</td>
<td>170</td>
</tr>
</tbody>
</table>
List of Figures

2.1 An abstract algorithm for applying a transformer using a theorem prover. . . . . 32
2.2 A semantic reduction for sharpening a two-vocabulary abstract element. . . . . 34

3.1 A simple Java-like implementation of the in-place reversal of a singly linked list. . 48
3.2 Example specification of reverse procedure . . . . . . . . . . . . . . . . . . . . 50
3.3 Proving $P_{\text{NoExit}}[r,x,n,n']$. . . . . . . . . . . . . . . . . . . . . . . . . . . 50
3.4 The instances of coloring axioms used in proving reverse. . . . . . . . . . . . . 52
3.5 Axiom scheme incremental instantiation algorithm . . . . . . . . . . . . . . . . 53

4.1 Histograms detailing performance improvement of SPASS with labelled-clauses . . 66

5.1 Notations used in Chapter 5 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
5.2 An example concrete state and its abstraction . . . . . . . . . . . . . . . . . . . . 69
5.3 An abstract state $S_3$. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
5.4 The operations supported and their concrete semantics. . . . . . . . . . . . . . 72
5.5 Example abstract state for exponential focus . . . . . . . . . . . . . . . . . . . . 74
5.6 Translation into logical formulas and query evaluation rules . . . . . . . . . . . . 77
5.7 An abstract state $S_3$ and its improved translation into a logical structure . . . . 78
5.8 The empirical results from running the abstract transformer implementation . . . 81

6.1 Example of concrete and abstract structures and the result of the transformer . . 84
6.2 Example structures resulting from running Focus . . . . . . . . . . . . . . . . . . 86

7.1 Remove method of StringBuffer data-structure . . . . . . . . . . . . . . . . . . 99
7.2 Invariant for Remove method of StringBuffer data-structure . . . . . . . . . . 100
7.3 BubbleSort Routine. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
7.4 Flowchart Nodes. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
7.5 The algorithm for join in the combined domain . . . . . . . . . . . . . . . . . . 110
7.6 The algorithm for existential elimination in the combined domain . . . . . . . . 111
7.7 The algorithm for post-predicate of the combined domain . . . . . . . . . . . . 115
7.8 The algorithm for widening in the combined domain . . . . . . . . . . . . . . . 117
7.9 The algorithm for implication checking in the combined domain . . . . . . . . . 117

A.1 Code for append and mark . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 144
A.2 Example specification of append procedure . . . . . . . . . . . . . . . . . . . . 145
A.3 The instances of coloring axioms used in proving append. . . . . . . . . . . . . 145
A.4 Example specification of mark procedure . . . . . . . . . . . . . . . . . . . . . . 146
Chapter 1

Introduction

Static Analysis is the process of inferring properties of programs statically (i.e., without actually executing the program). This thesis is concerned with Shape Analysis: static analysis of programs manipulating linked data structures focusing on the structure of the program’s memory, i.e., graph of objects and threads, and the relationships between them (e.g., see \[JM81, SRW02\]). We concentrate on proving safety properties for the purpose of program verification. These properties include memory safety properties such as absence of null-dereferences and absence of memory leaks, preservation of data structure invariants, and partial correctness. Specifically, we are interested in transitive properties, i.e., properties concerning paths in the graph depicting the memory. As we shall see, these properties are very useful, but particularly difficult to maintain and prove.

Goals. The goals of this thesis are to: (i) develop techniques for automatically maintaining transitive properties for the purpose of shape analysis, (ii) effectively harness existing tools such as theorem provers and existing static analyses in developing our shape analyses, and (iii) implement the techniques and evaluate them empirically.

Overview

The material in this thesis is based on a journal paper and four conference papers, each one with its corresponding technical chapter. This chapter contains an informal overview of the thesis, describing the contributions of each of the papers, and the connections between them.

The thesis is composed of three parts:

1. Chapters 3 and 4 — using automated theorem provers for program analysis [LAIR\(^+\)09, LAWRS07b],

2. Chapters 5 and 6 — constructing shape analysis algorithms specialized to a given data structure [LAIS06, LASIR07], and

3. Chapter 7 — combining existing shape analysis algorithms with numerical analysis to reason about sizes of data structures [GLAS09].
A Quick Introduction to Shape Analysis

Automatically establishing safety properties of programs that permit dynamic storage allocation and low-level pointer manipulations is very challenging. Dynamic allocation causes the state space to be unbounded. Moreover, in a program with destructive updates to pointer-valued variables and fields 1. modifications through pointers can change the shape of heap graph arbitrarily (e.g., from a tree shape to a grid shape) - shape invariants are usually not specified, maintained, or guaranteed by the language and runtime; and 2. changes to heap graph through one variable indirectly affects the view of the heap graph from other pointer variables. The program invariants used to infer these properties are either hand-written or computed automatically using Abstract Interpretation [CC77].

Canonical Abstraction is a class of abstractions introduced by Sagiv, Reps, and Wilhelm [SRW02] for analyzing programs that use dynamic data structures, including allocation and deallocation of memory cells and destructive updates of pointer-valued fields. In canonical abstraction, data structures are modeled using (3-valued) logical structures. Each element of the universe of the structure represents either a single memory cell, or, if the element is a summary element, a set of memory cells.

The analysis simulates the program step-by-step, updating structures appropriately, mimicking (i.e., approximating soundly) the semantics of program statements. When a fixed-point is reached, the resulting set of structures is a finite summary of relevant properties of the data structures built by the program. Note that any resulting properties of the set of structures are thus proven to hold: they necessarily hold on all runs of the program. This analysis framework has been implemented in the TVLA system [LAS00]. (The acronym stands for Three-Valued Logic Analyzer.)

Chapter 2 contains more detailed background on these concepts.

1.1 Using Automated Theorem Provers for Shape Analysis

We start by exploring the use of first-order automated theorem provers for shape analysis. These powerful tools have advanced considerably in the last decade and are able to deal with complex reasoning problems.

1.1.1 Partial Axiomatization of Transitive Closure

In Chapter 3, we explore how to harness existing theorem provers for first-order logic to prove reachability properties of programs that manipulate dynamically allocated data structures. The approach that we use involves simulating reachability in a conservative way, using first-order formulas—i.e., the formulas describe a superset of the set of program states that can actually arise.

Automatically establishing safety properties of sequential and concurrent programs that permit dynamic storage allocation and low-level pointer manipulations is challenging. Dynamic allocation causes the state space to be infinite; moreover, a program is permitted to mutate a data structure by destructively updating pointer-valued fields of nodes. These features remain challenging even if a programming language has good capabilities for data abstraction. Abstract-datatype operations are implemented using loops, procedure calls, and sequences of
low-level pointer manipulations; consequently, it is hard to prove that a data-structure invariant is reestablished once a sequence of operations has finished [Hoa75]. In languages such as Java, concurrency poses yet another challenge: establishing the absence of deadlock requires establishing the absence of any cycle of threads that are waiting for locks held by other threads.

Reachability is crucial for reasoning about linked data structures. For instance, to establish that a memory configuration contains no garbage elements, we must show that every element is reachable from some program variable. Other cases where reachability is a useful notion include:

- Specifying acyclicity of data-structure fragments, i.e., from every element reachable from node $n$ via a series of field accesses, one cannot reach $n$;
- Specifying the effect of procedure calls when references are passed as arguments: only elements that are reachable from a formal parameter can be modified;
- Specifying the absence of deadlocks; and
- Specifying safety conditions that allow establishing that a data-structure traversal terminates, e.g., there is a path from a node to a “sink-node” of the data structure.

The verification of such properties presents a challenge. Even simple decidable fragments of first-order logic (denoted FO, for short) become undecidable when reachability is added [GOR99, IRR+04a]. In the sequel, we denote first-order logic with a transitive closure operator by FO(TC). Moreover, the utility of monadic second-order logic on trees, which can naturally express reachability, is rather limited because (i) many programs allow non-tree data structures, (ii) expressing the postcondition of a procedure (which is essential for modular reasoning) usually requires referring to the pre-state that holds before the procedure executes, and thus cannot, in general, be expressed in monadic second-order logic on trees—even for procedures that manipulate only singly-linked lists, and (iii) the complexity of the required decision procedures is prohibitive (non-elementary in the general case).

While our work was actually motivated by our experience using abstract interpretation — and, in particular, the TVLA system — to establish properties of programs that manipulate heap-allocated data structures, in Chapter 3, we consider the problem of verifying data-structure operations, assuming that we have user-supplied loop invariants. This is similar to the approach taken in systems like ESC/Java [FLL+02], and Pale [MS01].

The contributions of Chapter 3 can be summarized as follows:

**Handling FO(TC) formulas using FO theorem provers.** We want to use first-order theorem provers and reason about the transitive closure of certain binary predicates, $f$. However, first-order theorem provers do not allow expressing formulas with transitive closure. We solve this conundrum by adding a new relation symbol $f_{tc}$ for each such $f$, together with first-order axioms that assure that $f_{tc}$ is interpreted correctly. The theoretical details of how this is done are presented in Section 3.2. The fact that we are able to handle transitive closure effectively and reasonably automatically is quite surprising.

As explained in Section 3.2, the axioms that we add to control the behavior of the added predicates, $f_{tc}$, are sound but not necessarily complete. One way to think about this is that we are simulating a formula, $\chi$, in which transitive closure occurs, with a pure first-order formula $\chi'$. If our axioms are not complete then we are allowing $\chi'$ to denote more stores than $\chi$
does. This is motivated by the fact that abstraction can be an aid in the verification of many properties; that is, a definite answer can sometimes be obtained even when information has been lost (in a conservative manner).

If $\chi'$ is proven valid in FO then $\chi$ is also valid in FO(TC); however, if we fail to prove that $\chi'$ is valid, it is still possible that $\chi$ is valid: the failure would be due to the incompleteness of the axioms, or the lack of time or space for the theorem prover to complete the proof.

As we will see in Section 3.2, it is easy to write a sound axiom, $T_1[f]$, that is “complete” in the very limited sense that every finite, acyclic model satisfying $T_1[f]$ must interpret $f_{tc}$ as the reflexive, transitive closure of its interpretation of $f$. However, in practice this is not worth much because, as is well-known, finiteness is not expressible in first-order logic. Thus, the properties that we want to prove do not follow from $T_1[f]$. We do prove that $T_1[f]$ is complete for positive transitive-closure properties (Proposition 3.2.2). The real difficulty lies in proving properties involving the negation of $f_{tc}$, i.e., that a certain $f$-path does not exist.

### Induction axiom scheme.

To solve the above problem, we add an induction axiom scheme. Although in general, there is no complete, recursively-enumerable axiomatization of transitive closure (Proposition 3.3.1), we have found, on the practical side, that on the examples we have tried, $T_1$ combined without induction axioms allows us to automatically prove all of our desired properties on a set of relevant benchmarks.

We think of the axioms that we use as aids for the first-order theorem prover we employ (SPASS [WBH+02]) to prove the properties in question. Rather than giving the first-order theorem prover many instances of the induction scheme, our experience is that it finds the proof faster if we give it several axioms that are simpler to use than directly instantiating the induction axioms.

### Coloring axiom schemes.

In particular, we use three axiom schemes, having to do with partitioning the set of memory cells by using a small set of colors. We call instances of these schemes “coloring axioms”. Our coloring axioms are simple, and are easily proved using SPASS (in under ten seconds) from the induction axioms. For example, the first coloring axiom scheme, $\text{NoExit}[A, f]$, says that if no $f$-edges leave color class, $A$, then no $f$-paths leave $A$. It turns out that the $\text{NoExit}$ axiom scheme implies – and thus is equivalent to – the induction scheme. However, we have found in practice that explicitly adding other coloring axioms (which are consequences of $\text{NoExit}$) enables SPASS to prove properties that it otherwise fails at.

We first assume that the programmer provides the colors by means of first-order formulas with transitive closure. Our initial experience indicates that the generated coloring axioms are useful to SPASS. In particular, it provides the ability to verify programs like the mark phase of a mark-and-sweep garbage collector. This example has been previously reported as being beyond the capabilities of ESC/Java. TVLA also succeeds on this example; however our new approach provides verification methods that can in some instances be more precise than TVLA.

### Prototype implementation.

Perhaps most exciting, we have implemented the heuristics for selecting colors and their corresponding axioms in a prototype using SPASS. We have used this to automatically choose useful color axioms and then verify a series of small heap-manipulating programs. We believe that the detailed examples presented here give convincing evidence of
the promise of our methodology. In addition, we have developed an instantiation of TVLA that uses SPASS as its reasoning engine using the theory of [YRS04] and the axiomatization introduced in this chapter.

**Strengthening Nelson’s results.** Greg Nelson considered a set of axiom schemes for reasoning about reachability in function graphs, i.e., graphs in which there is at most one $f$-edge leaving any node [Nel83]. He left open the question of whether his axiom schemes were complete for function graphs. We show that Nelson’s axioms are provable from $T_1$ plus our induction axioms. We also show that Nelson’s axioms are not complete: in fact, they do not imply \texttt{NoExit}.

### 1.1.2 Labelled Clauses

In Chapter 3, we explore the issue of reasoning about reachability using first-order theorem provers while deferring the problem of computing loop invariants. The work of Yorsh et al. [YRS04] shows how to leverage the parametric abstraction of TVLA while using theorem provers for computing the effects of program statements. In practice, a major obstacle in using existing automated theorem provers, such as E [Sch02], SPASS [WBH+02] and VAMPIRE [RV02], is that of performance. TVLA requires many calls to the theorem prover to compute the effect of a single program statement. Therefore, the overall time required for analyzing even a simple program is prohibitive. This situation is common in other program analysis methods such as Cartesian Abstraction [BPR01].

In the process of computing the effect of a program statement, TVLA generates multiple conjectures that share a common axiom set. However, current theorem provers can only attempt to prove a single conjecture at a time. Running the theorem prover multiple times, once for each conjecture, has three problems: (1) if the conjectures are proven sequentially, any inferences/reductions made between clauses from the axiom set are reconstructed and no synergies between the proof attempts can be exploited, (2) because not all the conjectures are valid and first-order logic is only semi-decidable, we are required to use timeouts, which may cause us to give up on valid conjectures, (3) if a conjecture is validated by a proof attempt, there is no automatic way inside the prover to transfer the result to other proof attempts.

To overcome these obstacles, we devised algorithms to prove different conjectures simultaneously, inside one proof attempt. We chose to modify the calculus used by the theorem prover to support deduction in parallel, i.e., for multiple conjectures, via a methodology of labelled clauses. The idea is to label each clause with the conjectures for which it is relevant, and update these labels during deduction. As a consequence, we solved the above problems: (1) inferences on axiom clauses are no longer duplicated but are shared — as are variant clauses derived from separate conjectures and reductions, (2) a fair strategy among all proof attempts (with one global timeout parameter) replaces the use of separate timeouts, and (3) valid conjectures can be naturally transferred to other proof attempts.

We have created a prototype implementation of labelled clauses for the purpose of proving multiple conjectures within the automated theorem prover SPASS, and report on convincing experimental results in the context of our original application.

After having seen the success of labelled clauses for multiple conjectures, we believe that the methodology of labelled clauses has more potential in the context of first-order theorem proving. Labelled deductive systems (see e.g., [BDG+00]) as have been pushed forward by
Dov Gabbay in the last fifteen years and have become recognized as a significant component of the logic culture, in particular in the context of non-classical logics. There, labels are used on the one hand to bring the semantics into the syntax by naming possible worlds using labels (e.g., a Kripke structure) and on the other hand they can act as proof-theoretic resource labels. Our motivation for using labels is different. We suggest to use labels to study and implement variants of classical first-order-logic (superposition-based) theorem-proving calculi to eventually improve automation. We show that the methodology of labels carries over beyond proving multiple conjectures by instantiating the methodology for clausal splitting (see e.g., [Wei01]), slicing (see e.g., [Wol98]), and the set-of-support strategy (see e.g., [WRC65]).

We use labels to summarize the derivation tree of a clause. For example, when using labelled clauses for clausal splitting, the labels represent the splits the clause depends on, and when using labelled clauses for multiple conjectures, the labels represent the conjectures for which the clause is valid.

The main contributions of Chapter 4 are as follows:

- We introduce a deduction technique for working on multiple related proof attempts simultaneously. We have successfully created a first prototype implementation within SPASS.
- We describe applications of the method in software verification, and provide experimental results to demonstrate the improvement.
- We propose the concept of superposition with labels as a general framework for the study of deduction techniques and their combination.
- We demonstrate several instantiations of this general framework for implementing different ideas, including clause splitting and slicing.

1.2 Specialized Shape Analysis

The techniques developed in Chapters 3 and 4 allow for effective reasoning about reachability using first-order theorem provers. However, two major issues remain:

- **Performance** — even with the advances of Chapter 4, the TVLA variant using theorem provers is decidedly slower because of the overhead of the theorem prover.
- **Predictability** — the use of partial axiomatization means that when failing to prove a property one does not know if the abstraction loses too much information or the axiomatization is not precise enough.

In this part of the thesis we complement our toolbox, instead of using very generic tools such as first-order theorem provers, we use tailored algorithms for specific shape analysis problems.

1.2.1 Specialized Shape Analysis for Commonly Used Data Structures

In Chapter 5, we develop a simple abstraction method for reasoning about reachability that is provably efficient and precise. This provides both a practical analysis method and a theoretical contribution towards the understanding of how precise and efficient shape analysis can be.
Main Results

We present a method to conservatively verify reachability properties via abstract interpretation. Specifically, we present a new lightweight method for shape analysis that applies to programs on “uniform tree-like” data structures. Furthermore, we compute the most precise abstract transformer [CC79]. That is, we provide an efficient algorithm that incurs the minimal loss of precision when reasoning about loop-free sequences of statements under the given abstraction. A prototype of the algorithm was implemented and is shown to be fast. The system can be seen as a specialization of TVLA to a set of data-structures and a set of properties.

New Abstraction of Heap Shape. We present a simple abstraction for heaps based on contracting segments of the heap into a single summary-node. In contrast to existing methods, our abstraction admits precise and efficient recovery of reachability information concerning the modeled concrete states. For example, every path in the abstraction between “important” nodes is a must-path, i.e., it must exist between the corresponding nodes in each modeled concrete state. Thus, reasoning about reachability between important nodes can be performed efficiently via simple graph traversal of the abstracted graph.

We show that the abstraction of graphs with no undirected cycles yields a linear number of abstract nodes. Therefore, the size of the abstract state space is bounded for such programs, allowing effective state space exploration. Moreover, this also holds for simple cycles such as cyclic singly-linked lists, (cyclic) doubly-linked lists and trees with parent pointers. Furthermore, it is possible to apply our abstraction only in loop boundaries and thus allowing programs to temporarily violate the data-structure invariants.

Efficient Most Precise Abstract Transformers. We present an efficient algorithm for computing the most precise abstract transformers for Java-like atomic program statements including destructive pointer manipulation.

Most existing methods for shape analysis including TVLA do not implement the most precise transformers and may require exponential time to produce a single abstract state. Also, in contrast to existing methods for generating the most precise abstract transformers (e.g. [GS97, YRS04, BR05]), our method does not employ a theorem prover and yet still manages to precisely maintain reachability information under abstraction.

Efficient algorithms for computing the most precise transformers for predicate abstraction in singly-linked lists were developed in [MYRS05]. This chapter can be considered as a continuation of [MYRS05] that handles more complex data-structures.

Information Extraction. It is important to extract information from an abstract state about the concrete states that it models. For example, we sometimes need to verify disjointness of data structures. For safety properties we check that user-specified assertions hold in every execution leading to a given program point.

We provide a conservative and efficient method that extracts such information by evaluating a first-order formula with transitive closure on a given abstract state. Our method is more precise than standard Kleene evaluation (e.g., [SRW02]), although less precise than super-valuational semantics [BG00, RLS02]. We show that our method is exact for “atomic” reachability properties between important nodes. Our preliminary experiments indicate that one of our evaluation methods is precise enough in practice.
1.2.2 Constructing Specialized Shape Analyses for Uniform Change

In Chapter 5 we present a specialized shape analysis for some commonly used data structures. Chapter 6 is concerned with the general question — for a given abstraction and a given set of concrete transformers (that express the concrete semantics of a program), how does one create the associated abstract transformers? We develop a new methodology for addressing this problem, based on a syntactically restricted language for expressing concrete transformers. Of particular interest is that—by employing previous results from dynamic algorithms and dynamic descriptive complexity [Imm99]—our methods allow precise reachability information to be maintained for abstractions of data structures. We use this methodology to produce most precise abstract transformers for abstractions of many important data structures.

Shape Analysis, Canonical Abstraction, and Dynamic Descriptive Complexity: While our approach is quite general, the main application is to shape analysis and to analyses based on canonical abstraction.

A key technical difficulty concerns the summary nodes abstraction. They are needed so that the unbounded-size set of unbounded-size concrete data structures that can arise are always abstracted to a finite set of finite-size logical structures, which guarantees that the analysis always converges. The problem caused by summary nodes abstraction is that some relations between cells in memory can be true for some elements represented by a summary node and false for others. Hence a truth value of “$\frac{1}{2}$” is introduced, and the framework is based on 3-valued logic [SRW02]. As the analysis propagates 3-valued structures, however, there is a tendency for logical values of $\frac{1}{2}$, i.e., “don’t know”, to increase, which limits the quality of information that the analysis can provide.

A good way to combat this problem is to maintain extra, auxiliary relations in the logical structures [SRW02, LRS05]. The same approach is used in dynamic descriptive complexity, although the motivation is completely different:

- In dynamic descriptive complexity, we work with objects that undergo a series of inserts, deletes, changes, and queries; with each query, the goal is to return the answer with respect to the current object. The fundamental issue in dynamic descriptive complexity is one of efficiency: “What auxiliary information should be maintained to answer the query quickly?” The goal of maintaining extra information is to avoid recomputing each answer from scratch.

- In static analysis based on 3-valued logic, the issue is not so much to save computation time, but instead to preserve high-quality information, i.e., definite truth values—“0”s and “1”s, rather than “$\frac{1}{2}$”s—whenever possible.

A second key technical difficulty concerns reachability information, which is needed to express connectivity and separation properties of data structures. There has been extensive work in dynamic descriptive complexity on how to efficiently maintain reachability information. For example, Dong and Su showed that for acyclic graphs reachability may be maintained by first-order formulas [DS95]. Of particular interest to us is the result of Hesse that reachability for (not-necessarily acyclic) functional graphs can be maintained by quantifier-free formulas [Hes03].

Our New Methodology: As explained before, TVLA maintains abstract (3-valued) structures, $\mathcal{A}$, that represent sets of concrete (2-valued) structures, $\gamma(\mathcal{A})$. We say that an abstract structure, $\mathcal{A}$, is feasible iff $\gamma(\mathcal{A}) \neq \emptyset$. Let $\beta$ be the abstraction operator on individual concrete...
1.2. Specialized Shape Analysis

structures, i.e., $\beta(\mathcal{C})$ is the abstract representation of $\mathcal{C}$, so $\beta$ and $\gamma$ are (approximate) inverse operations (adjoined functions).

For each program statement, $st$, TVLA has an update formula $\tau_{st}$ so that on any concrete structure, $\mathcal{C}$, $\tau_{st}(\mathcal{C})$ is the concrete structure produced by executing statement $st$. Furthermore, the update formula is always safe (meaning sound) on abstract structures, meaning that $\tau_{st}(\gamma(A)) \subseteq \gamma(\tau_{st}(A))$.

Given an abstraction, the most precise abstract transformer $bt_{st}$ satisfies the property, $bt_{st}(A) = \{ \beta(\tau_{st}(\mathcal{C})) \mid \mathcal{C} \in \gamma(A) \}$. However, because $\gamma(A)$ may be infinite, the equation above does not provide an algorithm for computing the most precise abstract transformer.

TVLA employs heuristics to efficiently compute a safe transformer that is not necessarily the most precise abstract transformer. In this chapter, we introduce a syntactic condition called monadic uniform with the following property (see also Theorem 6.3.6):

**Main Theorem:** If the update formulas for a data structure are monadic uniform and we have an algorithm that given an abstract structure, $A$, decides whether $A$ is feasible, then we can automatically compute the most precise abstract transformers for the operations on the data structure.

We then show that our main theorem applies to many important situations:

- We use and modify known results from dynamic descriptive complexity to create monadic-uniform update formulas for many important classes of data structures, including linked lists, cyclic linked lists, doubly-linked lists, cyclic doubly-linked lists, trees, shared trees, directed graphs with no undirected cycles, and also some of the above data structures when arbitrary unary relations and an ordering relation are included.

- We also present efficient feasibility algorithms for most of the above. Thus, for these data structures we can implement most precise abstract transformers automatically.

Our vision is to build specialized shape analyses for many of the available programs and observed properties. This chapter is an important step in this direction because it shows that it is possible to build — in a systematic manner — specialized shape analyses with good theoretical properties for many important data structures.

**Predicate Abstraction:** Our results are not limited to the TVLA context; in particular, they provide a way to improve the predicate-abstraction method given by Rakamaric et al. [RBH06, RBH07]. Their linked-list abstraction uses the relation $\text{between}(x, y, z)$ to capture whether there is a path from $x$ to $z$ through $y$. Rakamaric et al. give a complete decision procedure for checking feasibility of a given abstract state, but left open the question of how to handle transformers in the most-precise way. Our methodology solves this problem: we can use the quantifier-free update formulas given by Hesse [Hes03] to build most precise abstract transformers for this abstraction. For example, to compute the abstract transformer for the addition/removal of an edge we would: (1) extend the vocabulary with a constant capturing the current target of the edge; (2) replace each abstract state with the set of states that provide all possible interpretations to the predicates involving the new constant; (3) use the Rakamaric et al. decision procedure to remove the infeasible abstract states; (4) for the remaining states, evaluate Hesse’s update formulas to get the successor states.
1.3 Combining Shape Analysis Domains

We have explored two extremes approaches for shape analysis: using a very generic reasoning method such as an automated theorem prover, and using shape analysis algorithms specialized to specific data structures; both approaches have their merits. The last part of the thesis explores the ability to take an existing shape analysis algorithm and enhance by combining it with other analyses.

1.3.1 A Combination Framework for Tracking Partition Sizes

The theme of Chapter 7 is to automatically establish invariants regarding sizes of memory partitions. Such invariants are crucial in order to bound the size of dynamically allocated data (e.g., in embedded systems). They are also necessary in order to infer the shape of the data in programs that manipulate both arrays and dynamically allocated data structures, which is common in many implementations of abstract data types such as hashing, skip-lists, B-Trees, and string implementations. Moreover, proving such invariants in these programs is required for proving their memory safety.

We describe new algorithms for establishing such invariants by combining two kinds of abstractions: (a) Abstractions that partition the memory into (not necessarily) disjoint parts and (b) Numerical abstractions that can track relationships between numeric variables. Our algorithms are parameterized by both abstractions which allows to leverage existing shape abstractions (e.g., [SRW02, PW05, DOY06, MYRS05]) and existing numerical abstractions (e.g., Polyhedra [CH78], Octagons [Min01], Intervals [CC77]). We call such an analysis a set cardinality analysis.

We first formalize the notion of a set abstract domain that provides abstractions to partition the memory into (not necessarily disjoint) parts called base-sets. We describe the interface that a set domain should export in order for it to be combinable with a numerical domain (Section 7.2). A key component of such an interface is the Witness operator that relates a given base-set with other base-sets that occur in a given set-domain element. (This relationship is transformed into a numerical relationship over the cardinalities of the base-sets by the combination framework, and is the only window to communicate any information about the meaning of a base-set to the numerical domain, which otherwise views base-sets as uninterpreted and simply uses a fresh variable to denote the cardinality of each base-set.) We show that several popular heap/shape analysis domains can be easily made to support such an interface (Section 7.2.3) — this is one of the contributions of the chapter.

We then define the notion of a set cardinality abstract domain that is parameterized by a set domain and a numerical domain (Section 7.3). An element of the set cardinality domain is a pair composed of a set-domain element and a numerical element. The interesting part here is our formalization of the pre-order between the elements in this domain, which defines the level of reasoning built into our set cardinality domain. The pre-order is defined constructively in terms of the partial orders of the individual domains. Hence, given a decision procedure for the set domain and a decision procedure for the numerical domain, our pre-order construction shows how to convert them into a decision procedure for the set cardinality domain. Such a modular construction of the decision procedure for the set cardinality domain is an independent contribution of the chapter, and fits in the stream of work on decision procedures for reasoning about sets and their cardinalities [KR07]. Though we do not prove any completeness results.
1.4. Thesis Organization

The rest of the thesis is organized as follows:

- Chapter 2 contains background materials for Canonical Abstraction [SRW02] and using theorem provers to compute abstract transformers;

- Chapter 3 presents a technique for reasoning about reachability using first-order theorem provers employing a partial axiomatization of transitive closure.

- Chapter 4 describes a technique for proving multiple conjectures from the same axioms simultaneously and shows its usefulness in computing abstract transformers using theorem provers.
Chapter 5 presents an abstract domain for shape analysis of various data structures, which has efficient and precise transformers.

Chapter 6 describes a methodology for constructing specialized shape analysis domains which have precise abstract transformers.

Chapter 7 describes a combination framework for analyzing data structures and their sizes using existing shape and numeric analysis domains.

Chapter 8 discussed related work for the various parts of the thesis.

Chapter 9 concludes the thesis and discusses possible future research directions.

Appendix A presents further examples for Chapter 3.

Appendix B details proofs for Chapter 5.

Appendix C describes additional applications and proofs for Chapter 6.

Appendix D details proofs for Chapter 7.
Chapter 2

Background

In this chapter, we provide a reminder on abstract interpretation and First Order Logic with Transitive Closure (FO(TC)). We also include a brief introduction to the theory of parametric shape analysis via 3-valued logic [SRW02] and, in particular, define Canonical Abstraction.

2.1 Abstract Interpretation Essentials

In this thesis, we assume that an abstract domain $A$ is given by a complete lattice $D_A = \langle \subseteq_A, \sqcup_A, \sqcap_A, \bot_A, \top_A \rangle$ where $A$ is the set of elements; $\subseteq_A$ is a partial ordering on the elements; $\sqcup_A$ is the least upper bound, or join, operator; $\sqcap_A$ is the greatest lower bound, or meet, operator; $\bot_A$ is the least element of the lattice; and $\top_A$ is the greatest element of the lattice. We say that an element $c_1$ is more precise than an element $c_2$ when $c_1 \subseteq c_2$.

In abstract interpretation [CC77], an abstraction function $\alpha^{C,A} : 2^C \rightarrow A$ maps each subset of the concrete domain $C$ to the most precise element that represents it in the abstract domain $A$. The meaning of an abstract element $a \in A$ is given by a concretization function $\gamma^{A,C} : A \rightarrow 2^C$. That is, we say that $a \in A$ represents any element $c \in C$ such that $c \in \gamma^{A,C}(a)$. Moreover, the pair $(\gamma^{A,C}, \alpha^{C,A})$ forms a Galois Connection.

In the sequel, we will drop the subscripts and superscripts denoting the semantic domains when no confusion is likely.

A semantic function $\bar{F}^\# : A \rightarrow A$ is a sound over-approximation of a semantic function $F : 2^C \rightarrow 2^C$ if the following holds:

$$F(\gamma(a)) \subseteq \gamma(\bar{F}^\#(a)) .$$

We call the function $F$ the concrete transformer and the function $\bar{F}^\#$ the abstract transformer. The most precise abstract transformer is defined as $\alpha \circ F \circ \gamma$. Note that for infinite concrete domain this definition does not yield an algorithm.

The semantics of a program is given in terms of the least fixed point $lfp(F)$ and its abstract semantics is given by $lfp(\bar{F}^\#)$\textsuperscript{1}. Properties of a program can be conservatively inferred by starting from an initial element $a_0$ and then applying the function $\bar{F}^\#$ over and over until reaching the fixed point. This process is guaranteed to end when the height of the lattice $A$ is finite.

\textsuperscript{1}Least fixed points are usually computed using the sequence of iterates defined by $X_0 = a_0$ and $X_{n+1} = X_n \sqcup \bar{F}^\#(X_n)$.
When the height of $A$ may be infinite, a widening operator may be used to ensure convergence. A widening operator for a domain $A$ takes as input two elements from $A$ and produces an upper bound of those elements (which may not necessarily be the least upper bound). A widening operator has the property that it guarantees that fixed point computation across loops terminates in a finite number of steps even for infinite height domains.

## 2.2 FO(TC) Notation

### 2.2.1 Syntax

A relational vocabulary $\tau = \{p_1, p_2, \ldots, p_k\}$ is a set of relation symbols, each of fixed arity. We use the letters $u$, $v$, and $w$ (possibly with numeric subscripts) for first order variables. We write FO(TC) formulas over $\tau$ with quantifiers $\forall$ and $\exists$, logical connectives $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, and $\neg$, where atomic formulas include: equality (denoted using infix $=$), predicate evaluation $p_i(v_1, v_2, \ldots v_{a_i})$, and transitive closure $\neg TC[f](v_1, v_2)$, where $p_i \in \tau$ is of arity $a_i$ and $f \in \tau$ is binary. A formula without $TC$ is called a first-order formula.

We use the following precedence of logical operators: $\neg$ has highest precedence, followed by $\wedge$ and $\vee$, followed by $\rightarrow$ and $\leftrightarrow$. Finally $\forall$ and $\exists$ have lowest precedence.

### 2.2.2 Semantics

**Definition 2.2.1** A 2-valued logical structure over a vocabulary $\tau$ is a pair $S = \langle U^S, i^S \rangle$ where $U^S$ is the non-empty universe of the 2-valued structure, and $i^S$ is the interpretation function mapping predicates to their truth-value in the structure: for every predicate $p \in \tau$ of arity $k$, $i^S(p) : U^S^k \rightarrow \{0, 1\}$.

In the following, we use $p^S(v)$ as alternative notation for $i^S(p)(v)$; we also omit the superscript $S$, when no confusion is likely. We write $S \models \varphi$ to mean that the formula $\varphi$ is true in the structure $S$ and say that $S$ is a model of $\varphi$. $S$ is model of $\Sigma$ when $S$ is a model of all $\varphi \in \Sigma$. For $\Sigma$ a set of formulas, we write $\Sigma \models \varphi$ to mean that all models of $\Sigma$ satisfy $\varphi$ ($\Sigma$ semantically implies $\varphi$).

We use $TC[f](v_1, v_2)$ to denote the existence of a finite path of 0 or more $f$ edges from $v_1$ to $v_2$ (when interpreting the universe as graph nodes and each binary relation $f$ as an edge relation).

## 2.3 Concrete Program States in Logical Shape Analysis

We represent the state of a program using a first-order logical structure in which each individual corresponds to a heap-allocated object and predicates of the structure correspond to properties of heap-allocated objects.

In the context of shape analysis, a logical structure is used as a shape descriptor to represent a concrete state, with each individual corresponding to a heap-allocated object and predicates of the structure corresponding to properties of heap-allocated objects.

---

2We use relation symbols and predicates interchangeably.
Table 2.1: Typical predicates used for representing concrete program states.

<table>
<thead>
<tr>
<th>Predicates</th>
<th>Intended Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ x(v) : x ∈ PVar }</td>
<td>reference variable x points to the object v</td>
</tr>
<tr>
<td>n(v₁, v₂)</td>
<td>next field of the object v₁ points to the object v₂</td>
</tr>
</tbody>
</table>

Table 2.2: Predicate-update formulas that define the semantics of heap-manipulating statements.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Update formulas</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = null</td>
<td>x'(v) := 0</td>
</tr>
<tr>
<td>x = t</td>
<td>x'(v) := t(v)</td>
</tr>
<tr>
<td>x = t.n</td>
<td>x'(v) := ∃v₁ : t(v₁) ∧ n(v₁, v)</td>
</tr>
<tr>
<td>x.n = null</td>
<td>n'(v₁, v₂) := n(v₁, v₂) ∧ ¬x(v₁)</td>
</tr>
<tr>
<td>x.n = t (assuming x.n == null)</td>
<td>n'(v₁, v₂) = n(v₁, v₂) ∧ (x(v₁) ∧ t(v₂))</td>
</tr>
</tbody>
</table>

Table 2.1 shows the predicates we typically use to record properties of individuals. A unary predicate x(v) holds when the object v is pointed-to by the reference variable x. We assume that the set of predicates includes a unary predicate for every reference variable in a program. We use PVar to denote the set of all reference variables in a program. A binary predicate n(v₁, v₂) records the value of the reference field n.

2.3.1 Concrete Semantics

Program statements are modeled by actions that specify how statements transform an incoming logical structure into an outgoing logical structure. This is done primarily by defining the values of the primed predicates (representing the outgoing structure) using formulas of first-order logic with transitive closure over the unprimed predicates (representing the incoming structure) [SRW02]. The update formulas for heap-manipulating statements are shown in Table 2.2 (predicates not shown are preserved). For brevity, we omit the treatment of the allocation statement new T(), the interested reader may find the details in [SRW02].

To simplify update formulas, we assume that every assignment to the n field of an object is preceded by first assigning null to it.

2.4 Canonical Abstraction

The goal of an abstraction is to create a finite representation of a potentially unbounded set of 2-valued structures (representing heaps) of potentially unbounded size. The abstractions we use are based on 3-valued logic [Kle87, SRW02], which extends Boolean logic by introducing a third value \( \frac{1}{2} \) denoting values that may be 0 or 1. We say that the values 0 and 1 are definite values and that \( \frac{1}{2} \) is an indefinite value. A partial (information) order on truth values is defined
Table 2.3: Predicates used for the Canonical Abstraction of singly-linked lists in [SRW02], and their meaning.

<table>
<thead>
<tr>
<th>Predicates</th>
<th>Intended Meaning</th>
<th>Defining formulas</th>
</tr>
</thead>
<tbody>
<tr>
<td>( { x(v) : x \in PVar } )</td>
<td>reference variable ( x ) points to ( v )</td>
<td></td>
</tr>
<tr>
<td>( n(u, v) )</td>
<td>next field of ( u ) points to ( v )</td>
<td></td>
</tr>
<tr>
<td>( { r_x(v) : x \in PVar } )</td>
<td>( v ) is reachable from ( x ) by dereferencing ( n ) fields</td>
<td>( \exists v_x. x(v_x) \land n^*(v_x, v) )</td>
</tr>
<tr>
<td>( c_n(v) )</td>
<td>( v ) resides on a cycle of ( n ) fields</td>
<td></td>
</tr>
<tr>
<td>( is(v) )</td>
<td>( v ) is heap-shared</td>
<td>( \exists v_1, v_2. n(v_1, v) \land n(v_2, v) \land (v_1 \neq v_2) )</td>
</tr>
<tr>
<td>( eq(v_1, v_2) )</td>
<td>( v_1 ) equals ( v_2 )</td>
<td>( v_1 = v_2 )</td>
</tr>
</tbody>
</table>

as follows \( l_1 \sqsubseteq l_2 \) if \( l_1 = l_2 \) or \( l_2 = \frac{1}{2} \). The symbol \( \sqsubseteq \) denotes the least-upper-bound operation with respect to \( \sqsubseteq \).

We represent an abstract state of a program using a 3-valued first-order structure.

**Definition 2.4.1** A 3-valued logical structure over a set of predicates \( \tau \) is a pair \( S = \langle U, \iota \rangle \) where \( U \) is the universe of the 3-valued structure (an individual in \( U \) may represent multiple heap-allocated objects), and \( \iota \) is the interpretation function mapping predicates to their truth-value in the structure: for every predicate \( p \in \tau \) of arity \( k \), \( \iota(p) : U^k \rightarrow \{0, 1, \frac{1}{2}\} \).

### 2.4.1 Embedding

We now formally define how states are represented using abstract states. The idea is that each individual from the (concrete) state is mapped into an individual in the abstract state. More generally, it is possible to map individuals from an abstract state into an individual in another, less precise, abstract state.

Formally, let \( S = \langle U, \iota \rangle \) and \( S' = \langle U', \iota' \rangle \) be abstract states. A function \( f : U \rightarrow U' \) such that \( f \) is surjective is said to embed \( S \) into \( S' \) if for each predicate \( p \) of arity \( k \), and for each \( u_1, \ldots, u_k \in U \), one of the following holds:

\[
\iota(p(u_1, \ldots, u_k)) = \iota'(p(f(u_1), \ldots, f(u_k))) \quad \text{or} \quad \iota'(p(f(u_1), \ldots, f(u_k))) = \frac{1}{2}.
\]

We say that \( S' \) represents \( S \) when there exists such an embedding \( f \).

One way of creating an embedding function \( f \) is by using Canonical Abstraction. Canonical Abstraction maps concrete individuals to an abstract individual based on the values of the individuals’ unary predicates. All individuals having the same values for unary predicate symbols are mapped by \( f \) to the same abstract individual.

An abstract state may include summary nodes, i.e., an individual which corresponds to one or more individuals in a concrete state represented by that abstract state\(^3\). In the rest of the thesis, we assume that the set of predicates \( P \) includes a distinguished unary predicate \( sm \) to indicate if an individual is a summary individual.

\(^3\)Note that a summary node \( u \) has \( eq(u, u) = \frac{1}{2} \), indicating that it may represent more than a single individual.
Table 2.3 presents the set of predicates used in [SRW02] to abstract singly-linked lists. The predicates \( r_x(v), c_n(v), \) and \( is(v) \), referred to in [SRW02] as *instrumentation predicates*, record derived information and are used to refine the abstraction.

### 2.4.2 Abstract Transformers for 3-valued Logical Shape Analysis

In the original TVLA implementation [LAS00] the result of applying an abstract transformer to an abstract state is computed by a three step process:

- First, a heuristic is used to perform case splits, which locally refine the partition induced by the abstraction predicates for a specific statement. This process is called *Focus*.

- Second, the formulas comprising the concrete transformer are used to conservatively approximate the effect of the concrete transformer on all the represented memory states. Update formulas are either handwritten or derived using finite differencing [RSL03].

- Third, a constraint solver called *Coerce* is used to improve the precision of the abstract state by taking advantage of the inter-dependencies between the predicates dictated by the defining formulas of the instrumentation predicates and constraints of the programming language semantics.

### 2.5 Symbolically Computing Transformers for Shape Analysis

Graf and Saïdi [GS97] showed that, by working symbolically, decision procedures and theorem provers can be used to implement an algorithm that achieves the same result as applying the most-precise abstract transformer, for *predicate-abstraction domains* (i.e., abstract domains that are finite products of predicates on *State*)\(^4\). Subsequently, Yorsh et al. showed how to create algorithms for applying the most-precise abstract transformer for domains other than predicate-abstraction domains [RSY04, YRS04]. All of these algorithms exhibit an interesting interplay between abstract elements of an abstract domain and formulas in a logic. Such ideas are pursued further in this chapter (where the logic is first-order logic with transitive closure). This chapter generalizes the work of Yorsh et al. [YRS04], in particular it is can work with abstractions other than Canonical Abstraction. In addition, several partial contribution are presented in Section 2.5.2.

We make use of the following operations:

- The operation \( \gamma(a) \) expresses the concretization of abstract value \( a \) as a logical formula, which sidesteps the fact that, in general, the result of applying \( \gamma \) to \( a \) is an infinite set of concrete states. \( \gamma(a) \) converts \( a \) into a formula that characterizes \( \gamma(a) \) exactly; that is, the set of concrete states that satisfy \( \gamma(a) \) are exactly those in \( \gamma(a) \).

- The operation \( \alpha(\varphi) \) identifies the most-precise abstract value of a given abstract domain that overapproximates a set of concrete states that satisfy formula \( \varphi \).

\(^4\)The rest of this chapter is based on material that appears in [LASRG07].
The primitives are useful, yet—for pragmatic reasons—we are satisfied with something that is not as precise. In that case, $\hat{\gamma}(a)$ should overapproximate the states represented by $a$.

### 2.5.1 Symbolically Applying Transformers

In this section, we present a general algorithm for applying transformers using a theorem prover. We assume that the abstract domain used is a powerset lattice (possibly with a Hoare order—i.e., the elements of the sets have a partial order between them). We will refer to the sets in the lattice as abstract sets and to the individual elements in the sets as abstract elements. For a given transformer application, a **concrete state pair** is a pair of concrete states such that the concrete execution of the transformer on the first state may yield the second state. We call the first component of such a pair a pre-state and second component a post-state.

The algorithm takes a transformer and an abstract set representing the pre-states. The algorithm returns an abstract set that is an overapproximation of the post-states that match the pre-states represented by the input abstract set. The algorithm is written to apply transformers forwards, but could be trivially modified to apply transformers backwards (i.e., receive as input an abstract set that represents the post-states and return an abstract set that represents the pre-states).

The basis of the algorithm is the use of two-vocabulary abstract elements, i.e., abstract elements that directly represent pairs of concrete states. A two-vocabulary abstract element can be thought of as approximating the relation $st$ when restricted to the post-states represented by the input abstract element. We assume that each transformer $st$ has a two-vocabulary formula $\varphi_{st}$ that captures the meaning of the transformer by relating the possible pairs of pre- and post-states.

The algorithm operates on one abstract element at a time; it first applies the operation $\text{EXTEND} \text{POST}$, which takes an abstract element and extends it to a two-vocabulary abstract element.
2.5. **Symbolically Computing Transformers for Shape Analysis**

element that assumes nothing about the post-state. Thereafter, it repeatedly applies semantic reductions [CC79] (i.e., operations that move to a different element in the abstract lattice without changing the set of represented concrete states)\(^5\). When no more semantic reductions can be applied, it projects out the pre-state vocabulary, leaving just the post-state vocabulary.

The algorithm given in Figure 2.1 is based on a division of labor between two operations.

- The **SHARPEN** operation performs a semantic reduction that returns a tighter overapproximation of the concrete state pairs that the two-vocabulary abstract element represents—without moving to a set of two-vocabulary abstract elements. **SHARPEN** can also discover that the two-vocabulary abstract element does not represent any concrete state pairs at all. In that case, it returns a special element \(\ominus\), which causes the algorithm to give up on the current two-vocabulary abstract element and move on to the next one.

- The **FOCUS** operation is a semantic reduction that performs case splits by replacing a two-vocabulary abstract element by a set of two-vocabulary abstract elements that represent the same concrete state pairs. When **FOCUS** can perform no more case splits, it simply returns its input two-vocabulary abstract element\(^6\).

The algorithm uses the **SHARPEN** operation to tighten the overapproximation as much as possible, and then applies **FOCUS** to perform case splits to further improve the precision. These are iterated to increasingly tighten the overapproximation. To ensure termination of the algorithm, we require **FOCUS** to return a bounded number of two-vocabulary abstract elements and perform a bounded number of case splits.

Finally, we use the **PROJECTPOST** operation to project the two-vocabulary abstract element onto its second (post-state) component.

An algorithm for the **SHARPEN** operation is given in Figure 2.2. The algorithm requires that the abstract domain provide, in addition to \(\hat{\gamma}\), the following two operations:

- The **ASSERTIONS** operation returns a finite set of formulas, in some limited language, that describe the possible ways in which the abstract element can be restricted (i.e., to move down in the ordering on abstract elements).

- The **IMPOSE** operation takes an assertion and returns a new abstract element that incorporates the fact that the assertion holds.

The first step of the algorithm is to check that the two-vocabulary abstract element represents at least one pair of concrete states. This is done by checking validity of the negated \(\hat{\gamma}\) formula using a theorem prover. If the negated formula is valid, there can be no concrete state pair represented by this two-vocabulary abstract element, and the special indicator \(\ominus\) is returned. Afterwards, the **SHARPEN** algorithm simply enumerates the assertions returned by the **ASSERTIONS** operation, and for each of them uses the theorem prover to check whether they are implied by \(\hat{\gamma}\) of the two-vocabulary abstract element, conjoined with the transformer formula; that is, it checks symbolically that for every pair of concrete pre- and post-states of the transformer—as constrained by the two-vocabulary abstract element—the assertion holds. If this is the case, **SHARPEN** uses the **IMPOSE** operation to incorporate that information into the two-vocabulary abstract element.

---

\(^5\)These are semantic reductions in the domain that takes \(\varphi_{st}\) as part of \(\hat{\gamma}\).

\(^6\)Note that this differs from the original Focus semantic reduction which accepts a formula and returns a set of structures in which the formula evaluates to a definite value.
procedure SHARPEN\(S, st\)
if PROVER\((\neg(\overline{\gamma}(S) \land \varphi_{st}))\) returns valid then return \(\ominus\)
foreach \(\psi \in \text{ASSERTIONS} (S)\) do
if PROVER\((\overline{\gamma}(S) \land \varphi_{st} \rightarrow \psi)\) returns valid then
\(S := \text{IMPOSE}(S, \psi)\)
end
return \(S\)
end

Figure 2.2: A semantic reduction for sharpening a two-vocabulary abstract element.

2.5.2 Improvements

Using Model Finders

One of the main problems with working with a theorem prover to implement SHARPEN is that many provers are very good at proving validity, but very bad at proving invalidity (this is reasonable when considering the semi-decidable nature of first-order logic). When using such a theorem prover, most of the time is spent on trying to prove invalid assertions about the abstract state.

If the theory used for the abstract domain and the transformers has an effective model finder (e.g., a bounded model finder for first-order logic, such as Paradox [CS03]), we can do better (heuristically). The model finder has to be sound, but not necessarily complete, i.e., every model has to satisfy the formula, but there may be a satisfiable formula for which it does not produce a model. The model returned by the model finder is used to prune assertions before they are given to the theorem prover. We are interested in models of \(\Sigma = \overline{\gamma}(S) \land \varphi_{st}\). If evaluating an assertion \(\psi\) on such a model yields \(\text{false}\), then necessarily the assertion is not implied by the abstract element. A single model can be used to rule out many of the assertions. Furthermore, any time we pass the formula \(\Sigma \land \Phi \rightarrow \psi\) to the prover, we can pass \(\Sigma \land \Phi \land \neg\psi\) to the model finder and possibly accumulate more models of \(\Sigma\).

The use of the models is especially important because of the presence of the FOCUS operation. When considering the abstract elements resulting from FOCUS, one has to consider many of the assertions that we have not been able to prove on the original abstract element. This repeated work can become a major bottleneck in the algorithm. However, if we have models for the original abstract element, we can use them to prune many of these repeated calls. Note that a model for the abstract element before FOCUS is necessarily a model for at least one of the abstract elements in the result (because FOCUS is a semantic reduction). However, it is usually the case that it is not a model for all of the returned abstract elements. We can easily check which of the abstract elements the model belongs to by evaluating the appropriate \(\overline{\gamma}\) formula on the model.

Improving Performance

When working with a theorem prover, the algorithm checks whether the formula that characterizes the abstract structure implies any of the assertions. This can be done sequentially; however, using the technique from Chapter 4 we can extend a theorem prover to accept multiple conjectures in parallel and reason about them simultaneously, thus saving much of the
2.6 Transformers for Canonical Abstraction

In this section, we present a method for computing transformers for heap-manipulating programs. The method is based on the algorithm in Section 2.5. The abstract domain used is the Canonical Abstraction domain as explained above.

For the purpose of this section, we assume that the transformer cannot change the universe of the concrete state. Allocation and deallocation can be easily modeled by using a designated unary predicate which holds for the allocated heap cells. Similarly, we assume that the universe of the concrete state is non-empty.

We shall translate each 3-valued abstract element $S$ to the formula $\tilde{\gamma}(S)$ which holds on the concrete elements that it represents.

The individuals of a 3-valued logical structure are called abstract nodes. We use an auxiliary unary predicate for each abstract node to capture the concrete nodes that are mapped to it. For an abstract structure with universe $\{\text{node}_1, \ldots, \text{node}_n\}$ let $\{a_1, \ldots, a_n\}$ be the corresponding unary predicates. For each $k$-ary predicate $p$ in the vocabulary, each $k$-tuple $\langle \text{node}_1, \ldots, \text{node}_k \rangle$ in the abstract structure (called an abstract tuple) can have one of the following truth values $\{0, 1, \frac{1}{2}\}$. The truth value 1 means that the predicate $p$ universally holds for all of the concrete nodes mapped to this abstract node, i.e.,

$$\forall v_1, \ldots, v_k. a_1(v_1) \land \ldots \land a_k(v_k) \rightarrow p(v_1, \ldots, v_k) \quad (2.1)$$

Similarly the truth value 0 means that the predicate $p$ universally does not hold, i.e.,

$$\forall v_1, \ldots, v_k. a_1(v_1) \land \ldots \land a_k(v_k) \rightarrow \neg p(v_1, \ldots, v_k) \quad (2.2)$$

The truth value $\frac{1}{2}$ means that we have no information about this abstract tuple, and thus the value of the predicate $p$ is not restricted.

Recall that we use unary predicates to control the distinctions among concrete nodes that can be made in an abstract element, which also places a bound on the size of abstract elements. For each abstract node $\text{node}_i$, $A_i$ denotes the set of unary predicates for which $\text{node}_i$ has the truth value 1, and $\overline{A}_i$ denotes the set of unary predicates for which $\text{node}_i$ has the truth value 0. Each abstract node is uniquely determined by the values of $A_i$ and $\overline{A}_i$. In addition, every pair $\text{node}_i, \text{node}_j$ of different abstract nodes either $A_i \cap \overline{A}_j \neq \emptyset$ or $\overline{A}_i \cap \overline{A}_j \neq \emptyset$. This ensures that each concrete node can be represented by at most one abstract node. Because the number of unary predicates is fixed, this yields a bounded number of possible abstract nodes.

There are two additional requirements on abstract nodes. First, each node must represent at least one concrete node, i.e., $\exists v. a_i(v)$. Second, the abstract nodes in the structure represent all the concrete nodes, i.e., $\forall v. \bigvee_i a_i(v)$.

In addition, the definition formulas of all of the instrumentation predicates are part of $\tilde{\gamma}$.

We say that $S_1 \subseteq S_2$ if there exists a total mapping $m : U^{S_1} \rightarrow U^{S_2}$ such that $S_2$ represents all of the concrete states that $S_1$ represents when considering each abstract node of $S_2$ as a union of the abstract nodes of $S_1$ mapped to it by $m$. Formally, $\hat{\gamma}(S_1) \land \psi_m \rightarrow \hat{\gamma}(S_2)$ where

$$\psi_m = \bigwedge_{\text{node}_i \in U^{S_1}} \forall v. a_i(v) \rightarrow a'_j(v)$$

$$m(\text{node}_i) = \text{node}'_j \in U^{S_2}$$

This yields a redundant work of the separate proofs.
The order is extended to sets using the induced Hoare order (i.e., \( XS_1 \sqsubseteq XS_2 \) if for each element \( S_1 \in XS_1 \) there exists an element \( S_2 \in XS_2 \) such that \( S_1 \sqsubseteq S_2 \)).

The abstract domain can easily handle two-vocabulary elements by considering the predicates in the pre- and post-states together. To avoid a name clash, we use primed versions of the predicates in the post-state. The transformer formulas for basic statements can be easily written using first-order logic over this shared vocabulary. Note that the transformer formula should relate primed and unprimed versions of all the non-instrumentation predicates. On the other hand, it is not necessary to specify the update in the instrumentation predicates as it can be extracted from their defining formula.

**Transitive Closure**

The use of transitive closure in the defining formulas of instrumentation predicates means that PROVER is required to reason about transitive closure. Handling of transitive closure is based on the coloring axioms from Chapter 3. The coloring axioms are instantiated by considering combinations of abstract nodes.

Using bounded model finders for first-order logic as discussed in Section 2.5.2, has another advantage when applied to formulas with transitive closure. Because the model size is bounded, the transitive closure of a formula can be accurately formulated by using auxiliary binary predicates. These predicates iteratively represent paths of increasing length. For each \( j \) that is a power of 2, we add a predicate \( p_j \) that represents all paths of length less or equal to \( j \). The property is recursively defined using the formulas of the form

\[
\forall v_1, v_2. p_2(v_1, v_2) \leftrightarrow p_1(v_1, v_2) \lor \exists w. p_i(v_1, w) \land p_i(w, v_2)
\]

The bounded model size gives us a bound on the path length needed. Thus, we add a logarithmic number of new predicates.

### 2.6.1 Applying Transformers

To apply the abstract transformers for this domain, we use the algorithm in Figure 2.1. The abstract operations are instantiated as follows.

- **EXTENDPOST** by using the predicates in the pre-state as the unprimed predicates in the two-vocabulary structure. Because we assume that no transformer can change the universe of the concrete nodes, this corresponds to assuming nothing about the post-state.

- For **PROJECTPOST**, we throw away all the tuple formulas containing unprimed predicates and rename the primed predicates to their unprimed versions (this corresponds to projecting out the unprimed vocabulary in the 3-valued structure and renaming the primed predicates to their unprimed versions). This may cause several nodes to have the same values on all the abstraction predicates. In this case, they are merged by using the same abstract node for all of them\(^7\). The appropriate abstract tuples are also merged by only keeping universal formulas that still hold after the merge (i.e., that have the same truth value for all of the abstract tuples mapped to the same abstract tuple).

\(^7\)Similar to the Blur operation of [SRW02].
The **Assertions** operation takes every abstract tuple whose truth value is $\frac{1}{2}$ and returns two formulas, one to represent the case in which the truth value is 0 and another to represent the case in which the truth value is 1 (see Eq. (2.1) and Eq. (2.2)).

**Impose** sets the appropriate tuple to 0 or 1 according to the assertion proven.

To explain the **Focus** operation, we first define the concept of focusing a unary predicate on an abstract node. Let $p$ be a unary predicate and $node_i$ be an abstract node such that the truth value of $p$ on $node_i$ is $\frac{1}{2}$. In this case we return the following abstract elements:

- An element in which the value of $p$ on $node_i$ is changed to 0.
- An element in which the value of $p$ on $node_i$ is changed to 1.
- If $node_i$ is a summary node it can represent more than one concrete node. Thus, we return a third element in which $node_i$ is bifurcated into two nodes $node'_0$ and $node'_1$; all the abstract tuples which contain $node_i$ are duplicated to reflect that they hold on both $node'_0$ and $node'_1$. In addition, the value of $p$ on $node'_0$ is changed to 0, and the value of $p$ on $node'_1$ is changed to 1.

It is easy to see that the three abstract elements represent all the concrete states that the original abstract element represented. **Focus** chooses a single abstract node and a single abstraction predicate, and performs the aforementioned operation on it. If there is no predicate $p$ with value $\frac{1}{2}$, **Focus** returns the original element. **Focus** is thus guaranteed to return a bounded number of structures (up to three). Furthermore, because the condition is maintained that each pair of abstract nodes has an abstraction predicate with different values, there can be only a bounded number of times that **Focus** returns more than one element. Thus, the **ForwardTransform** algorithm is guaranteed to terminate.
Chapter 3
Partial Axiomatization of Transitive Closure

This chapter shows how to harness existing theorem provers for first-order logic to automatically verify safety properties of imperative programs that perform dynamic storage allocation and destructive updating of pointer-valued structure fields. One of the main obstacles is specifying and proving the (absence) of reachability properties among dynamically allocated cells.

The main technical contributions are methods for simulating reachability in a conservative way using first-order formulas—the formulas describe a superset of the set of program states that can actually arise. These methods are employed for semi-automatic program verification (i.e., using programmer-supplied loop invariants) on programs such as mark-and-sweep garbage collection and destructive reversal of a singly linked list. (The mark-and-sweep example has been previously reported as being beyond the capabilities of ESC/Java.) Furthermore, details are given on integrating these techniques with TVLA to support automatic program verification using abstract interpretation.

Outline
This chapter is organized as follows:

- Section 3.1 explains the problem setting;
- Section 3.2 fills in our formal framework, introduces the induction axiom scheme, and presents the coloring axiom schemes;
- Section 3.3 provides more detail about TC-completeness including a description of Nelson’s axioms, a proof that they are not TC-complete for the functional case;
- Section 3.4 presents our heuristics including the details of their successful use on a variety of examples.

3.1 Setting

We are primarily interested in formulas that arise while proving the correctness of programs. We assume that the programmer specifies pre and post-conditions for procedures and loop
invariants using first-order formulas with transitive closure on binary relations. The transformer for a loop body can be produced automatically from the program code.

For instance, to establish the partial correctness with respect to a user-supplied specification of a program that contains a single loop, we need to establish three properties: First, the loop invariant must hold at the beginning of the first iteration; i.e., we must show that the loop invariant follows from the precondition and the code leading to the loop. Second, the loop invariant provided by the user must be maintained; i.e., we must show that if the loop invariant holds at the beginning of an iteration and the loop condition also holds, the transformer causes the loop invariant to hold at the end of the iteration. Finally, the postcondition must follow from the loop invariant and the condition for exiting the loop.

In general, these formulas are of the form

$$\psi_1[\tau] \land Tr[\tau, \tau'] \rightarrow \psi_2[\tau']$$

where $\tau$ is the vocabulary of the before state, $\tau'$ is the vocabulary of the after state,¹ and $Tr$ is the transformer, which may use both the before and after predicates to describe the meaning of the module to be executed. If symbol $f$ denotes the value of a predicate before the operation, then $f'$ denotes the value of the same predicate after the operation.

An interesting special case is the proof of the maintenance formula of a loop invariant. This has the form:

$$LC[\tau] \land LI[\tau] \land Tr[\tau, \tau'] \rightarrow LI[\tau']$$

Here $LC$ is the condition for entering the loop and $LI$ is the loop invariant. $LI[\tau']$ indicates that the loop invariant remains true after the body of the loop is executed.

The challenge is that the formulas of interest contain transitive closure; thus, the validity of these formulas cannot be directly proven using a theorem prover for first-order logic.

### 3.2 Axiomatization of Transitive Closure

The original formula that we want to prove, $\chi$, contains transitive closure, which first-order theorem provers cannot handle. To address this problem, we replace $\chi$ by a new formula, $\chi'$, where all appearances of $TC[f]$ have been replaced by the new binary relation symbol, $f_{tc}$.

We show in this chapter that from $\chi'$, we can often automatically generate an appropriate first-order axiom, $\sigma$, with the following two properties:

1. if $\sigma \rightarrow \chi'$ is valid in FO, then $\chi$ is valid in FO(TC).
2. A theorem prover successfully proves that $\sigma \rightarrow \chi'$ is valid in FO.

We now explain the theory behind this process. A **TC model**, $A$, is a model such that if $f$ and $f_{tc}$ are in the vocabulary of $A$, then $(f_{tc})^A = (f^A)^*$; i.e., $A$ interprets $f_{tc}$ as the reflexive, transitive closure of its interpretation of $f$.

¹In some cases it is useful for the postcondition formula to refer to the original vocabulary as well. This way the postcondition can summarize some of the behavior of the transformer, e.g., summarize the behavior of an entire procedure.
A first-order formula $\varphi$ is **TC valid** iff it is true in all TC models. We say that an axiomatization, $\Sigma$, is **TC sound** if every formula that follows from $\Sigma$ is TC valid. Since first-order reasoning is sound, $\Sigma$ is TC sound iff every $\sigma \in \Sigma$ is TC valid.

We say that $\Sigma$ is **TC complete** if for every TC-valid $\varphi$, $\Sigma \models \varphi$. If $\Sigma$ is TC complete and TC sound, then for all first-order $\varphi$,

$$\Sigma \models \varphi \iff \varphi \text{ is TC valid}$$

Thus a TC-complete set of axioms proves exactly the first-order formulas, $\chi'$, such that the corresponding FO(TC) formula, $\chi$, is valid.

All the axioms that we consider are TC valid. There is no recursively enumerable TC-complete axiom system (Proposition 3.3.1). However, the axiomatization that we give does allow SPASS to prove all the desired properties on the examples that we have tried.

### 3.2.1 Some TC-Sound Axioms

We begin with our first TC axiom scheme. For any binary relation symbol, $f$, let,

$$T_1[f] \equiv \forall u, v. f_{tc}(u, v) \leftrightarrow (u = v) \lor \exists w. f(u, w) \land f_{tc}(w, v)$$

We first observe that $T_1[f]$ is “complete” in a very limited way for finite, acyclic graphs, i.e., $T_1[f]$ exactly characterizes the meaning of $f_{tc}$ for all finite, acyclic graphs. The reason that we say this is limited is that it does not give us a complete set of first-order axioms: as is well known, there is no first-order axiomatization of “finite”.

**Proposition 3.2.1** Any finite and acyclic model of $T_1[f]$ is a TC model.

**Proof:** Let $A \models T_1[f]$ where $A$ is finite and acyclic. Let $a_0, b \in |A|$. Assume that there is an $f$-path from $a_0$ to $b$. Since $A \models T_1[f]$, it is easy to see that $A \models f_{tc}(a_0, b)$. Conversely, suppose that $A \models f_{tc}(a_0, b)$. If $a_0 = b$, then there is a path of length 0 from $a_0$ to $b$. Otherwise, by $T_1[f]$, there exists an $a_1 \in |A|$ such that $A \models f(a_0, a_1) \land f_{tc}(a_1, b)$. Note that $a_1 \neq a_0$ since $A$ is acyclic. If $a_1 = b$ then there is an $f$-path of length 1 from $a$ to $b$. Otherwise, there must exist an $a_2 \in |A|$ such that $A \models f(a_1, a_2) \land f_{tc}(a_2, b)$ and so on, generating a set \{a_1, a_2, \ldots \}. None of the $a_i$ can be equal to $a_j$, for $j < i$, by acyclicity. Thus, by finiteness, some $a_i = b$. Hence $A$ is a TC model.

Let $T_1'[f]$ be the $\longleftrightarrow$ direction of $T_1[f]$:

$$T_1'[f] \equiv \forall u, v. f_{tc}(u, v) \leftrightarrow (u = v) \lor \exists w. f(u, w) \land f_{tc}(w, v)$$

**Proposition 3.2.2** Let $f_{tc}$ occur only positively in $\varphi$. If $\varphi$ is TC valid, then $T_1'[f] \models \varphi$.

**Proof:** Suppose that $T_1'[f] \not\models \varphi$. Let $A \models T_1'[f] \land \neg \varphi$. Note that $f_{tc}$ occurs only negatively in $\neg \varphi$. Furthermore, since $A \models T_1'[f]$, it is easy to show by induction on the length of the path, that if there is an $f$-path from $a$ to $b$ in $A$, then $A \models f_{tc}(a, b)$. Define $A'$ to be the model formed from $A$ by interpreting $f_{tc}$ in $A'$ as $(f^A)^\ast$. Thus $A'$ is a TC model and it only differs from $A$ by the fact that we have removed zero or more pairs from $(f_{tc})^A$ to form $(f_{tc})^{A'}$. Because $A \models \neg \varphi$ and $f_{tc}$ occurs only negatively in $\neg \varphi$, it follows that $A' \models \neg \varphi$, which contradicts the assumption that $\varphi$ is TC valid.
Proposition 3.2.2 shows that proving positive facts of the form \( f_{tc}(u, v) \) is easy; it is the task of proving that paths do not exist that is more subtle.

Proposition 3.2.1 shows that what we are missing, at least in the acyclic case, is that there is no first-order axiomatization of finiteness. Traditionally, when reasoning about the natural numbers, this problem is mitigated by adding induction axioms. We next introduce an induction scheme that, together with \( T_1 \), seems to be sufficient to prove any property we need concerning TC.

**Notation:** In general, we will use \( F \) to denote the set of all binary relation symbols, \( f \), such that \( TC[f] \) occurs in a formula we are considering. If \( \varphi[f] \) is a formula in which \( f \) occurs, let \( \varphi[F] = \bigwedge_{f \in F} \varphi[f] \). Thus, for example, \( T_1[F] \) is the conjunction of the axiom \( T_1[f] \) for all binary relation symbols, \( f \), under consideration.

**Definition 3.2.3** For any first-order formulas \( Z(u), P(u) \), and binary relation symbol, \( f \), let the induction principle, \( \text{IND}[Z, P, f] \), be the following first-order formula:

\[
(\forall w.Z(w) \rightarrow P(w)) \land (\forall u, v.P(u) \land f(u, v) \rightarrow P(v)) \\
\rightarrow \forall u, w.Z(w) \land f_{tc}(w, u) \rightarrow P(u)
\]

In order to explain the meaning of \( \text{IND} \) and other axioms it is important to remember that we are trying to write axioms, \( \Sigma \), that are,

- **TC valid**, i.e., true in all TC models, and
- **useful**, i.e., all models of \( \Sigma \) are sufficiently like TC models that they satisfy the TC-valid properties we want to prove.

To make the meaning of our axioms intuitively clear, in this section we will say, for example, that “\( y \) is \( f_{tc} \)-reachable from \( x \)” to mean that \( f_{tc}(x, y) \) holds. Later, we will assume that the reader has the idea and just say “reachable” instead of “\( f_{tc} \)-reachable”.

The intuitive meaning of the induction principle is that if every zero point satisfies \( P \), and \( P \) is preserved when following \( f \)-edges, then every point \( f_{tc} \)-reachable from a zero point satisfies \( P \). Obviously this principle is TC valid, i.e., it is true for all structures such that \( f_{tc} = f^* \).

As an easy application of the induction principle, consider the following cousin of \( T_1[f] \),

\[
T_2[f] \equiv \forall u, v. f_{tc}(u, v) \leftrightarrow (u = v) \lor \exists w. f_{tc}(u, w) \land f(w, v)
\]

The difference between \( T_1 \) and \( T_2 \) is that \( T_1 \) requires that each path represented by \( f_{tc} \) starts with an \( f \) edge and \( T_2 \) requires the path to end with an \( f \) edge. It is easy to see that neither of \( T_1[f] \), \( T_2[f] \) implies the other. However, in the presence of the induction principle they do imply each other. For example, it is easy to prove \( T_2[f] \) from \( T_1[f] \) using \( \text{IND}[Z, P, f] \) where \( Z(v) \equiv v = u \) and \( P(v) \equiv u = v \lor \exists w. f_{tc}(u, w) \land f(w, v) \). Here, for each \( u \) we use \( \text{IND}[Z, P, f] \) to prove by induction that every \( v \) reachable from \( u \) satisfies the right-hand side of \( T_2[f] \).

Another useful axiom scheme provable from \( T_1 \) plus \( \text{IND} \) is the transitivity of reachability:

\[
\text{Trans}[f] \equiv \forall u, v, w. f_{tc}(u, w) \land f_{tc}(w, v) \rightarrow f_{tc}(u, v)
\]
3.2.2 Coloring Axioms

We next describe three TC-sound axioms schemes that are not implied by $T_1[F] \land T_2[F]$, and are provable from the induction principle. We will see in the sequel that these coloring axioms are very useful in proving that paths do not exist, permitting us to verify a variety of algorithms. In Section 3.4, we will present some heuristics for automatically choosing particular instances of the coloring axiom schemes that enable us to prove our goal formulas.

The first coloring axiom scheme is the NoExit axiom scheme:

$$(\forall u, v . A(u) \land \neg A(v) \rightarrow \neg f(u, v)) \rightarrow \forall u, v . A(u) \land \neg A(v) \rightarrow \neg f_{tc}(u, v)$$

for any first-order formula $A(u)$, and binary relation symbol, $f$, NoExit$[A, f]$ says that if no $f$-edge leaves color class $A$, then no point outside of $A$ is $f_{tc}$-reachable from $A$.

Observe that although it is very simple, NoExit$[A, f]$ does not follow from $T_1[f] \land T_2[f]$. Let $G_1 = (V, f, f_{tc}, A)$ be a model consisting of two disjoint cycles: $V = \{1, 2, 3, 4\}$, $f = \{(1, 2), (2, 1), (3, 4), (4, 3)\}$, and $A = \{1, 2\}$. Let $f_{tc}$ have all 16 possible pairs. Thus $G_1$ satisfies $T_1[f] \land T_2[f]$ but violates NoExit$[A, f]$. Even for acyclic models, NoExit$[A, f]$ does not follow from $T_1[f] \land T_2[f]$ because there are infinite models in which the implication does not hold (Proposition 3.3.7).


The second coloring axiom scheme is the GoOut axiom: for any first-order formulas $A(u), B(u)$, and binary relation symbol, $f$, GoOut$[A, B, f]$ says that if the only $f$-edges leaving color class $A$ are to $B$, then any $f_{tc}$-path from a point in $A$ to a point not in $A$ must pass through $B$.

$$(\forall u, v . A(u) \land \neg A(v) \land f(u, v) \rightarrow B(v)) \rightarrow \forall u, v . A(u) \land \neg A(v) \land f_{tc}(u, v) \rightarrow \exists w . B(w) \land f_{tc}(u, w) \land f_{tc}(w, v)$$

To see that GoOut$[A, B, f]$ follows from the induction principle, assume that the only $f$-edges out of $A$ enter $B$. For any fixed $u$ in $A$, we prove by induction that any point $v$ $f_{tc}$-reachable from $u$ is either in $A$ or has a predecessor, $b$ in $B$, that is $f_{tc}$-reachable from $u$.

The third coloring axiom scheme is the NewStart axiom, which is useful in the context of dynamically changing graphs: for any first-order formula $A(u)$, and binary relation symbols $f$ and $g$, think of $f$ as the previous edge relation and $g$ as the current edge relation. NewStart$[A, f, g]$ says that if there are no new edges between $A$ nodes, then any new path, i.e., $g_{tc}$ but not $f_{tc}$, from $A$ must leave $A$ to make its change:

$$(\forall u, v . A(u) \land A(v) \land g(u, v) \rightarrow f(u, v)) \rightarrow \forall u, v . g_{tc}(u, v) \land \neg f_{tc}(u, v) \rightarrow \exists w . \neg A(w) \land g_{tc}(u, w) \land g_{tc}(w, v)$$

NewStart$[A, f, g]$ follows from the induction principle by a proof that is similar to the proof of GoOut$[A, B, f]$.
Linked Lists

The spirit behind our consideration of the coloring axioms is similar to that found in a paper of Greg Nelson’s in which he introduced a set of reachability axioms for a functional predicate, \( f \), i.e., there is at most one \( f \) edge leaving any point [Nel83]. Nelson asked whether his axiom schemes are complete for the functional setting. We remark that Nelson’s axiom schemes are provable from \( T_1 \) plus our induction principle. However, Nelson’s axiom schemes are not complete: we constructed a functional graph that satisfies Nelson’s axioms but violates \texttt{No Exit}[A, f] (Proposition 3.3.7).

At least one of Nelson’s axiom schemes seems orthogonal to our coloring axioms and may be useful in certain proofs. Nelson’s fifth axiom scheme states that the points reachable from a given point are linearly ordered. The soundness of the axiom scheme is due to the fact that \( f \) is functional. We make use of a simplified version of Nelson’s ordering axiom scheme: Let \( \text{Func}[f] \equiv \forall u, v, w. f(u, v) \land f(u, w) \rightarrow v = w \); then,

\[
\text{Order}[f] \equiv \text{Func}[f] \rightarrow \forall u, v, w. f_{\text{tc}}(u, v) \land f_{\text{tc}}(u, w) \rightarrow f_{\text{tc}}(v, w) \lor f_{\text{tc}}(w, v)
\]

Trees

When working with programs manipulating trees, we have a fixed set of selectors \( \text{Sel} \) and transitive closure is performed on the \( \text{down} \) relation, defined as

\[
\forall v_1, v_2. \text{down}(v_1, v_2) \leftrightarrow \bigvee_{s \in \text{Sel}} s(v_1, v_2)
\]

Trees have no sharing (i.e., the \( \text{down} \) relation is injective), thus a similar axiom to \( \text{Order}[f] \) is used:

\[
\forall u, v, w. \text{down}_{\text{tc}}(v, u) \land \text{down}_{\text{tc}}(w, u) \rightarrow \text{down}_{\text{tc}}(v, w) \lor \text{down}_{\text{tc}}(w, v)
\]

Another important property of trees is that the subtrees below distinct children of a node are disjoint. We use the following axioms to capture this, where \( s_1 \neq s_2 \in \text{Sel} \):

\[
\forall v, v_1, v_2, w. \neg(s_1(v, v_1) \land s_2(v, v_2) \land \text{down}_{\text{tc}}(v_1, w) \land \text{down}_{\text{tc}}(v_2, w))
\]

3.3 On TC-Completeness

In this section we consider the concept of TC-Completeness in detail. We first show that there is no recursively enumerable TC-complete set of axioms.

**Proposition 3.3.1** Let \( \Gamma \) be an r.e. set of TC-valid first-order sentences. Then \( \Gamma \) is not TC-complete.
Proof: By the proof of Corollary 9, page 11 of [IRR+04a], there is a recursive procedure that on input any Turing machine, $M_n$, produces a first-order formula $\varphi_n$ in a vocabulary $\tau_n$ such that $\varphi_n$ is TC-valid iff Turing machine, $M_n$, on input 0 never halts. The vocabulary $\tau_n$ consists of the two binary relation symbols, $E, E_{tc}$, constant symbols, $a, d$, and some unary relation symbols. It follows that if $\Gamma$ were TC-complete, then it would prove all true instances of $\varphi_n$ and thus the halting problem would be solvable.

Proposition 3.3.1 shows that even in the presence of only one binary relation symbol, there is no r.e. TC-complete axiomatization.

In [Avr03], Avron gives an elegant finite axiomatization of the natural numbers using transitive closure, a successor relation and the binary function symbol, “+”. Furthermore, he shows that multiplication is definable in this language. Since the unique TC-model for Avron’s axioms is the standard natural numbers it follows that:

**Corollary 3.3.2** Let $\Gamma$ be an arithmetic set of TC-valid first-order sentences over a vocabulary including a binary relation symbol and a binary function symbol (or a ternary relation symbol). Then $\Gamma$ is not TC-complete.

In Proposition 3.2.1 we showed that any finite and acyclic model of $T_1[f]$ is a TC model. This can be strengthened to

**Proposition 3.3.3** Any finite model of $T_1$ plus $\text{IND}$ is a TC-model.

**Proof:** Let $\mathcal{A}$ be a finite model of $T_1$ plus $\text{IND}$. Let $f$ be a binary relation symbol, and let $a, b$ be elements of the universe of $\mathcal{A}$. Since $\mathcal{A} \models T_1$, if there is an $f$ path from $a$ to $b$ then $\mathcal{A} \models f_{tc}(a, b)$.

Conversely, suppose that there is no $f$ path from $a$ to $b$. Let $R_a$ be the set of elements of the universe of $\mathcal{A}$ that are reachable from $a$. Let $k = |R_a|$. Since $\mathcal{A}$ is finite we may use existential quantification to name exactly all the elements of $R_a : x_1, \ldots, x_k$. We can then define the color class: $C(y) \equiv y = x_1 \lor \cdots \lor y = x_k$. Then we can prove using $\text{IND}$, or equivalently $\text{NoExit}$, that no vertex outside this color class is reachable from $a$, i.e., $\mathcal{A} \models \neg f_{tc}(a, b)$. Thus, as desired, $\mathcal{A}$ is a TC-model.

### 3.3.1 More About TC-Completeness

Even though there is no r.e. set of TC-complete axioms in general, there are TC-complete axiomatizations for certain interesting cases. Let $\Sigma$ be a set of formulas. We say that $\psi$ is **TC-valid wrt** $\Sigma$ iff every TC-model of $\Sigma$ satisfies $\psi$. Let $\Gamma$ be TC-sound. We say that $\Gamma$ is **TC-complete wrt** $\Sigma$ iff $\Gamma \cup \Sigma \vdash \psi$ for every $\psi$ that is TC-valid wrt $\Sigma$. We are interested in whether $T_1$ plus $\text{IND}$ is TC-complete with respect to interesting theories, $\Sigma$.

Since $\text{TC}[s](a, b)$ asserts the existence of a finite $s$-path from $a$ to $b$, we can express that a structure is finite by writing the formula: $\Phi \equiv \text{Func}[s] \land \exists x \forall y. s_{tc}(x, y)$. Observe that every TC-model that satisfies $\Phi$ is finite. Thus, if we are in a setting – as is frequent in logic – where we may add a new binary relation symbol, $s$, then finiteness is TC-expressible.

**Proposition 3.3.4** Let $\Sigma$ be a finite set of formulas, and $\Gamma$ an r.e., TC-complete axiomatization wrt $\Sigma$ in a language where finiteness is TC-expressible. Then finite TC-validity for $\Sigma$ is decidable.
Proof: Let $\Phi$ be a formula as above that TC-expresses finiteness. Let $\psi$ be any formula. If $\psi$ is not finite TC-valid wrt $\Sigma$, then we can find a finite TC model of $\Sigma$ where $\psi$ is false. If $\psi$ is finite TC-valid, then $\Gamma \cup \Sigma \vdash \Phi \rightarrow \psi$, and we can find this out by systematically generating all proofs from $\Gamma$.

From Proposition 3.3.4 we know that we must restrict our search for cases of TC-completeness to those where finite TC-validity is decidable. In particular, since the finite theory of two functional relations is undecidable, e.g. $\text{IRR}^+04a$, we know that,

**Corollary 3.3.5** There are no r.e. TC-valid axioms for the functional case even if we restrict to at most two binary relation symbols.

### 3.3.2 Nelson’s Axioms

Our idea of considering transitive-closure axioms is similar in spirit to the approach that Nelson takes [Nel83]. To prove some program properties, he introduces a set of reachability axiom schemes for a functional predicate, $f$. By “functional” we mean that $f$ is a partial function:

$$\text{Func}(f) \equiv \forall u, v, w. f(u, v) \land f(u, w) \rightarrow v = w.$$  

We remark that Nelson’s axiom schemes are provable from $T_1$ plus our induction principle. At least two of his schemes may be useful for us to add in our approach. Nelson asked whether his axioms are complete for the functional setting. It follows from Corollary 3.3.5 that the answer is no. We prove below that Nelson’s axioms do not prove $\text{NoExit}$.

Nelson’s basic relation symbols are ternary. For example, he writes, "$u \xrightarrow{f} x \rightarrow v$" to mean that there is an $f$-path from $u$ to $v$ that follows no edges out of $x$. We encode this as, $f_{tc}^x(u, v)$, where, for each parameter $x$ we add a new relation symbol, $f^x$, together with the assertion: $\forall u, v. f^x(u, v) \iff f(u, v) \land (u \neq x)$. Nelson also includes a notation for modifying the partial function $f$. He writes, $f^{(p)}_q$ for the partial function that agrees with $f$ everywhere except on argument $p$ where it has value $q$. Nelson’s eighth axiom scheme asserts a basic consistency property for this notation. In our translation we simply assert that $f^{(p)}_q(u, v) \leftrightarrow (u \neq p \land f(u, v)) \lor (u = p \land v = q)$. When we translate Nelson’s eighth axiom scheme the result is tautological, so we can safely omit it.

Using our translation, Nelson’s axiom schemes are the following.

- **N1** $f_{tc}^x(u, v) \leftrightarrow (u = v) \lor \exists z. (f^x(u, z) \land f_{tc}^z(z, v))$
- **N2** $f_{tc}^x(u, v) \land f_{tc}^x(v, w) \rightarrow f_{tc}^x(u, w)$
- **N3** $f_{tc}^x(u, v) \rightarrow f_{tc}(u, v)$
- **N4** $f_{tc}^y(u, x) \land f_{tc}^z(u, y) \rightarrow f_{tc}^z(u, x)$
- **N5** $f_{tc}(u, x) \rightarrow f_{tc}^y(u, x) \lor f_{tc}^x(u, y)$
- **N6** $f_{tc}^y(u, x) \land f_{tc}^z(u, y) \rightarrow f_{tc}^z(x, y)$
- **N7** $f(x, u) \land f_{tc}(u, v) \rightarrow f_{tc}^x(u, v)$

These axiom schemes can be proved using appropriate instances of $T_1$ and the induction principle. Just as we showed in Proposition 3.2.1 that any finite and acyclic model of $T_1[f]$ is a TC model, we have that,
Proposition 3.3.6 Any finite and functional model of Nelson’s axioms is a TC-model.

Proof: Consider any finite and function model, \( M \). We claim that for each \( f \) and \( x \in |M| \), \((f^x)^M = ((f^x)^M)\). If there is an \( f^x \) path from \( u \) to \( v \), then it follows from repeated uses of N1 that \( f^x \) holds.

If there is no \( f^x \) path from \( u \) to \( v \) and \( u \) is not on an \( f \)-cycle, then using N1 we can follow \( f \)-edges from \( u \) to \( v \) to prove that \( f^x \) does not hold.

If there is no \( f^x \) path from \( u \) to \( v \) and \( u \) is on an \( f \)-cycle containing \( x \), then using N1 we can follow \( f \)-edges from \( u \) to \( x \) to prove that \( f^x \) holds.

Finally, if there is no \( f \) path from \( u \) to \( v \) and \( u \) is on an \( f \)-cycle, suppose for the sake of a contradiction that \( f^x \) holds. Let \( x \) be the predecessor of \( u \) on the cycle. By N7, \( f^x \) must hold. However, this contradicts the previous paragraph.

Axiom schemes N5 and N7 may be useful for us to assert when \( f \) is functional. N5 says that the points reachable from \( u \) are totally ordered in the sense that if \( x \) and \( y \) are both reachable from \( u \), then in the path from \( u \) either \( x \) comes first or \( y \) comes first. N7 says that if there is an edge from \( x \) to \( u \) and a path from \( u \) to \( v \), then there is a path from \( u \) to \( v \) that does not go through \( x \). This implies the useful property that no vertex not on a cycle is reachable from a vertex on the cycle. The interested reader can find a description of the first part of the mark and sweep algorithm which has been proven using the axioms described in this paper and was reported to be beyond the capability of Nelson’s axioms.

We conclude this section by proving the following,

Proposition 3.3.7 Nelson’s axioms do not imply NoExit.

Proof: Consider the structure \( G = (V, f, f_{tc}, f^0_{tc}, f^1_{tc}, f^2_{tc}, \ldots, f^\infty_{tc}, A) \) such that \( V = \mathbb{N} \cup \{\infty\} \), the set of natural numbers plus a point at infinity. Let \( A = \mathbb{N} \), i.e., the color class \( A \) is interpreted as all points except \( \infty \). Define \( f = \{\langle u, u + 1 \rangle | u \in \mathbb{N}\} \), i.e., there is an edge from every natural number to its successor, but \( \infty \) is isolated. However, let \( f_{tc} = \{\langle u, v \rangle | u \leq v\} \), i.e., \( G \) believes that there is a path from each natural number to infinity. Similarly, for each \( k \in V \), \( f^k_{tc} = \{\langle u, v \rangle | u \leq v \land (k < u \lor v \leq k)\} \).

It is easy to check that \( G \) satisfies all of Nelson’s axioms.

The problem is that \( G \models \lnot \text{NoExit}[A, f] \). It follows that Nelson’s axioms do not entail \( \text{NoExit}[A, f] \). This is another proof that they are not TC complete.

3.4 Heuristics for Using the Coloring Axioms

This section presents heuristics for using the coloring axioms. Toward that end, it answers the following questions:

- How can the coloring axioms be used by a theorem prover to prove \( \chi \)? (Section 3.4.2)

- When should a specific instance of a coloring axiom be given to the theorem prover while trying to prove \( \chi \)? (Section 3.4.4)

- What part of the process can be automated? (Section 3.4.5)
Node reverse(Node x)
{
[0] Node y = null;
[1] while (x != null)
{
[2] Node t = x.next;
[3] x.next = y;
[4] y = x;
[5] x = t;
[6] }
[7] return y;
}

Figure 3.1: A simple Java-like implementation of the in-place reversal of a singly linked list.

We first present a running example (more examples are described in Section A.1 and used in later sections to illustrate the heuristics). We then explain how the coloring axioms are useful, describe the search space for useful axioms, give an algorithm for exploring this space, and conclude by discussing a prototype implementation we have developed that proves the example presented and others.

3.4.1 Specification of the Reverse Procedure

The heuristics described in Sections 3.4.2–3.4.4 are illustrated on problems that arise in the verification of partial correctness of a list reversal procedure. Other examples proven using this technique can be found in Appendix A.1.

The procedure reverse, shown in Figure 3.1, performs in-place reversal of a singly linked list, destructively updating the list. The precondition requires that the input list be acyclic and unshared (i.e., each heap node is pointed to by at most one heap node). For simplicity, we assume that there is no garbage. The postcondition ensures that the resulting list is acyclic and unshared. Also, it ensures that the nodes reachable from the formal parameter on entry to reverse are exactly the nodes reachable from the return value of reverse at the exit. Most importantly, it ensures that each edge in the original list is reversed in the returned list.

The specification for reverse is shown in Figure 3.2. We use unary predicates to represent program variables and binary predicates to represent data-structure fields. Figure 3.2(a) defines some shorthands. To specify that a unary predicate \( z \) can point to a single node at a time and that a binary predicate \( f \) of a node can point to at most one node (i.e., \( f \) is a partial function), we use \( \text{unique}[z] \) and \( \text{func}[f] \). To specify that there are no cycles of \( f \)-fields in the graph, we use \( \text{acyclic}[f] \). To specify that the graph does not contain nodes shared by \( f \)-fields, (i.e., nodes with 2 or more incoming \( f \)-fields), we use \( \text{unshared}[f] \). To specify that all nodes in the graph are reachable from \( z_1 \) or \( z_2 \) by following \( f \)-fields, we use \( \text{total}[z_1, z_2, f] \). Another helpful shorthand is \( r_{x,f}(v) \) which specifies that \( v \) is reachable from the node pointed to by \( x \) using \( f \)-edges.

The precondition of the reverse procedure is shown in Figure 3.2(b). We use the predicates \( xe \) and \( ne \) to record the values of the variable \( x \) and the next field at the beginning of the procedure. The precondition requires that the list pointed to by \( x \) be acyclic and unshared. It
also requires that \( \text{unique}[z] \) and \( \text{func}[f] \) hold for all unary predicates \( z \) that represent program variables and all binary predicates \( f \) that represent fields, respectively. For simplicity, we assume that there is no garbage, i.e., all nodes are reachable from \( x \).

The post-condition is shown in Figure 3.2(c). It ensures that the resulting list is acyclic and unshared. Also, it ensures that the nodes reachable from the formal parameter \( x \) on entry to the procedure are exactly the nodes reachable from the return value \( y \) at the exit. Most importantly, we wish to show that each edge in the original list is reversed in the returned list (see Eq. (3.9)).

A loop invariant is given in Figure 3.2(d). It describes the state of the program at the beginning of each loop iteration. Every node is in one of two disjoint lists pointed to by \( x \) and \( y \) (Eq. (3.10)). The lists are acyclic and unshared. Every edge in the list pointed to by \( x \) is exactly an edge in the original list (Eq. (3.12)). Every edge in the list pointed to by \( y \) is the reverse of an edge in the original list (Eq. (3.13)). The only original edge going out of \( y \) is to \( x \) (Eq. (3.14)).

The transformer is given in Figure 3.2(e), using the primed predicates \( n', x', \) and \( y' \) to describe the values of predicates \( n \), \( x \), and \( y \), respectively, at the end of the iteration.

### 3.4.2 Proving Formulas using the Coloring Axioms

All the coloring axioms have the form \( A \equiv P_A \rightarrow C_A \), where \( P_A \) and \( C_A \) are closed formulas. We call \( P_A \) the axiom’s premise and \( C_A \) the axiom’s conclusion. For an axiom to be useful, the theorem prover will have to prove the premise (as a subgoal) and then use the conclusion in the proof of the goal formula \( \chi \). For each of the coloring axioms, we now explain when the premise can be proved, how its conclusion can help, and give an example.

\textbf{NoExit.} The premise \( P_{\text{NoExit}}[C, f] \) states that there are no \( f \)-edges exiting color class \( C \). When \( C \) is a unary predicate appearing in the program, the premise is sometimes a direct result of the loop invariant. Another color that will be used heavily throughout this section is reachability from a unary predicate, i.e., unary reachability, formally defined in Eq. (3.6).

Let us examine two cases. \( P_{\text{NoExit}}[r_{x,f}, f] \) is immediate from the definition of \( r_{x,f} \) and the transitivity of \( f_{tc} \). \( P_{\text{NoExit}}[r_{x,f}, f'] \) actually states that there is no \( f \)-path from \( x \) to an edge for which \( f' \) holds but \( f \) does not, i.e., a change in \( f' \) with respect to \( f \). Thus, we use the absence of \( f \)-paths to prove the absence of \( f' \)-paths. In many cases, the change is an important part of the loop invariant, and paths from and to it are part of the specification.

A sketch of the proof by refutation of \( P_{\text{NoExit}}[r_{x',n}, n'] \) that arises in the reverse example is given in Figure 3.3. The numbers in brackets are the stages of the proof.

1. The negation of the premise expands to:

\[
\exists u_1, u_2, u_3. x'(u_1) \land n_{tc}(u_1, u_2) \land \neg n_{tc}(u_1, u_3) \land n'(u_2, u_3)
\]

2. Since \( u_2 \) is reachable from \( u_1 \) and \( u_3 \) is not, by \( T_2 \), we have \( \neg n(u_2, u_3) \).

3. By the definition of \( n' \) in the transformer, the only edge in which \( n \) differs from \( n' \) is out of \( x \) (one of the clauses generated from Eq. (3.15) is \( \forall v_1, v_2. \neg n'(v_1, v_2) \lor n(v_1, v_2) \lor x(v_1) \)). Thus, \( x(u_2) \) holds.

4. By the definition of \( x' \) it has an incoming \( n \) edge from \( x \). Thus, \( n(u_2, u_1) \) holds.
Figure 3.2: Example specification of reverse procedure: (a) shorthands, (b) precondition pre, (c) postcondition post, (d) loop invariant $\text{LI}[x, y, n]$, (e) transformer $T$ (effect of the loop body).

Figure 3.3: Proving $P_{\text{NoExit}}[r_{x, n}, n']$. 
The list pointed to by \( x \) must be acyclic, whereas we have a cycle between \( u_1 \) and \( u_2 \); i.e., we have a contradiction. Thus, \( P_{\text{NoExit}}[x', n, n'] \) must hold.

\( C_{\text{NoExit}}[C, f] \) states there are no \( f \) paths (\( f_{\text{tc}} \) edges) exiting \( C \). This is useful because proving the absence of paths is the difficult part of proving formulas with TC.

\textbf{GoOut}. The premise \( P_{\text{GoOut}}[A, B, f] \) states that all \( f \) edges going out of color class \( A \), go to \( B \). When \( A \) and \( B \) are unary predicates that appear in the program, again the premise sometimes holds as a direct result of the loop invariant. An interesting special case is when \( B \) is defined as \( \exists w . A(w) \land f(w, v) \). In this case the premise is immediate. Note that in this case the conclusion is provable also from \( T_1 \). However, from experience, the axiom is very useful for improving performance (2 orders of magnitude when proving the acyclic part of reverse’s postcondition).

\( C_{\text{GoOut}}[A, B, f] \) states that all paths out of \( A \) must pass through \( B \). Thus, under the premise \( P_{\text{GoOut}}[A, B, f] \), if we know that there is a path from \( A \) to somewhere outside of \( A \), we know that there is a path to there from \( B \). In case all nodes in \( B \) are reachable from all nodes in \( A \), together with the transitivity of \( f_{\text{tc}} \) this means that the nodes reachable from \( B \) are exactly the nodes outside of \( A \) that are reachable from \( A \).

For example, \( C_{\text{GoOut}}[y', y, n, n'] \) allows us to prove that only the original list pointed to by \( y \) is reachable from \( y' \) (in addition to \( y' \) itself).

\textbf{NewStart}. The premise \( P_{\text{NewStart}}[C, g, h] \) states that all \( g \) edges between nodes in \( C \) are also \( h \) edges. This can mean the iteration has not added edges or has not removed edges according to the selection of \( h \) and \( g \). In some cases, the premise holds as a direct result of the definition of \( C \) and the loop invariant.

\( C_{\text{NewStart}}[C, g, h] \) means that every \( g \) path that is not an \( h \) path must pass outside of \( C \). Together with \( C_{\text{NoExit}}[C, g] \), it proves there are no new paths within \( C \).

For example, in reverse the \textbf{NewStart} scheme can be used as follows. No outgoing edges were added to nodes reachable from \( y \). There are no \( n \) or \( n' \) edges from nodes reachable from \( y \) to nodes not reachable from \( y \). Thus, no paths were added between nodes reachable from \( y \). Since the list pointed to by \( y \) is acyclic before the loop body, we can prove that it is acyclic at the end of the loop body.

We can see that \textbf{NewStart} allows the theorem prover to reason about paths within a color, and the other axioms allow the theorem prover to reason about paths between colors. Together, given enough colors, the theorem prover can often prove all the facts that it needs about paths and thus prove the formula of interest.

### 3.4.3 The Search Space of Possible Axioms

To answer the question of when we should use a specific instance of a coloring axiom when attempting to prove the target formula, we first define the search space in which we are looking for such instances. The axioms can be instantiated with the colors defined by an arbitrary unary formula (one free variable) and one or two binary predicates. First, we limit ourselves to binary unary predicates for which TC was used in the target formula. Now, since it is infeasible to consider all arbitrary unary formulas, we start limiting the set of colors we consider.

The initial set of colors to consider are unary predicates that occur in the formula we want to prove. Interestingly enough, these colors are enough to prove that the postcondition of mark and sweep is implied by the loop invariant, because the only axiom we need is \( \text{NoExit}[\text{marked}, f] \).
An immediate extension that is very effective is forward and backward reachability from unary predicates, as defined in Eq. (3.6) and Eq. (3.7), respectively. Instantiating all possible axioms from the unary predicates appearing in the formula and their unary forward reachability predicates, allows us to prove reverse. For a list of the axioms needed to prove reverse, see Figure 3.4. Other examples are presented in Section A.1. Finally, we consider Boolean combinations of the above colors. Though not used in the examples shown in this chapter, this is needed, for example, in the presence of sharing or when splicing two lists together.

All the colors above are based on the unary predicates that appear in the original formula. To prove the reverse example, we needed $x'$ as part of the initial colors. Table 3.1 gives a heuristic for finding the initial colors we need in cases when they cannot be deduced from the formula, and how it applies to reverse.

An interesting observation is that the initial colors we need can, in many cases, be deduced from the program code. As in the previous section, we have a good way for deducing paths between colors and within colors in which the edges have not changed. The program usually manipulates fields using pointers, and can traverse an edge only in one direction. Thus, the unary predicates that represent the program variables (including the temporary variables) are in many cases what we need as initial colors.

### 3.4.4 Exploring the Search Space

When trying to automate the process of choosing colors, the problem is that the set of possible colors to choose from is doubly-exponential in the number of initial colors; giving all the ax-
3.4. HEURISTICS FOR USING THE COLORING AXIOMS

Figure 3.5: An iterative algorithm for instantiating the axiom schemes. Each iteration consists of three phases that augment the axiom set $\Sigma$.

directly to the theorem prover is infeasible. In this section, we define a heuristic algorithm for exploring a limited number of axioms in a directed way. Pseudocode for this algorithm is shown in Figure 3.5. The operator $\vdash$ is implemented as a call to a theorem prover.

Because the coloring axioms have the form $A \equiv P_A \rightarrow C_A$, the theorem prover must prove $P_A$ or the axiom is of no use. Therefore, the pseudocode works iteratively, trying to prove $P_A$ from the current $\psi \land \Sigma$, and if successful it adds $C_A$ to $\Sigma$.

The algorithm tries colors in increasing levels of complexity. $BC(i, C)$ gives all the Boolean combinations of the predicates in $C$ up to size $i$. After each iteration we try to prove the goal formula. Sometimes we need the conclusion of one axiom to prove the premise of another. The $\text{NoExit}$ axioms are particularly useful for proving $P_{\text{NewStart}}$. Therefore, we need a way to order instantiations so that axioms useful for proving the premises of other axioms are acquired first. The ordering we chose is based on phases: First, try to instantiate axioms from the axiom scheme $\text{GoOut}$. Second, try to instantiate axioms from the axiom scheme $\text{NoExit}$. Finally, try to instantiate axioms from the axiom scheme $\text{NewStart}$. For $P_{\text{NewStart}}[c, f, g]$ to be useful, we need to be able to show that there are either no incoming $f$-paths or no outgoing $f$-paths from $c$. Thus, we only try to instantiate such an axiom when either $P_{\text{NoExit}}[c, f]$ or
$P_{\text{NoExit}}[\neg c, f]$ has been proven.

### 3.4.5 Implementation

The algorithm presented here was implemented using a Perl script and the SPASS theorem prover [WGR96] and used successfully to verify the example programs of Section 3.4.1 and Section A.1.

The method described above can be optimized. For instance, if $C_A$ has already been added to the axioms, we do not try to prove $P_A$ again. These details are important in practice, but have been omitted for brevity.

When trying to prove the different premises, SPASS may fail to terminate if the formula that it is trying to prove is invalid. Thus, we limit the time that SPASS can spend proving each formula. It is possible that we will fail to acquire useful axioms this way.

The coloring axioms are applicable to a wide variety of verification problems. In Section 2.5, we describe the reasoning done by the TVLA system and how it can be simulated using the coloring axioms. Furthermore, the axioms described in this chapter have been used to integrate SPASS as the reasoning engine behind the TVLA system.

One of the main problems in using theorem provers for abstract interpretation is the need for multiple calls to a theorem prover in order to compute the post-image. The next chapter presents a technique that allows the theorem prover to accept multiple conjectures and process them in parallel.
Chapter 4

Labelled Clauses

In this chapter, we add labels to first-order clauses to simultaneously apply superpositions to several proof obligations inside one clause set. From a theoretical perspective, the approach unifies a variety of deduction modes. These include different strategies such as set of support, as well as explicit case analysis, e.g., splitting. From a practical perspective, labelled clauses offer advantages in the case of related proof obligations resulting from multiple conjectures over the same axiom set or from a single conjecture that is a large conjunction. Here, we can share clauses (e.g., the axioms and clauses deduced from them, share Skolem symbols), share deduced clause variants, and transfer lemmas between the different obligations. Motivated by software verification, we have created a prototype implementation of labelled clauses that supports multiple conjectures, and we provide convincing experiments for the benefits.

Outline

In Section 4.1, we present a labelled superposition calculus as an extension of the standard superposition calculus. In Section 4.2, we present several instantiations of the general calculus including one for multiple conjectures. In Section 4.3, we present applications of multiple conjectures in software verification and present experimental results from using our extension of SPASS for handling multiple conjectures.

The tool as well as the input files used for the experiments are available at [LAWRS07a].

4.1 Superposition with Labels

For the presentation of the superposition calculus, we refer to the notation and notions of [Wei01]. We write clauses in implication form $\Gamma \rightarrow \Delta$ where comma to the left means conjunction and comma to the right disjunction. Upper Greek letters denote sequences of atoms ($\Gamma, \Delta$), lower Greek letter substitutions ($\sigma, \tau$), lower Latin characters terms ($s, r, t$) and upper Latin characters atoms ($A, B, E$) where $\approx$ is used as the equality symbol. The notations $s[l]_p$ ($E[l]_p$) expresses that the term $s$ (the atom $E$) contains the term $l$ at position $p$. Replacing a subterm at position $p$ with a term $r$ is denoted by $s[p/r]$ ($E[p/r]$).

We distinguish inference from reduction rules, where the clause(s) below the bar, the conclusions, of an inference rule are added to the current clause set, while the clause(s) below the bar of a reduction rule replace the clause(s) above the bar, the premises. For example,
\[
\begin{array}{c}
\text{J} \quad \Gamma_1 \rightarrow \Delta_1 \quad \Gamma_2 \rightarrow \Delta_2 \\
\Gamma_3 \rightarrow \Delta_3 \\
\text{R} \quad \Gamma'_1 \rightarrow \Delta'_1 \quad \Gamma'_2 \rightarrow \Delta'_2 \\
\Gamma'_3 \rightarrow \Delta'_3
\end{array}
\]

an application of the above inference adds the clause \(\Gamma_3 \rightarrow \Delta_3\) to the current clause set, while
the above reduction replaces the clauses \(\Gamma'_1 \rightarrow \Delta'_1, \Gamma'_2 \rightarrow \Delta'_2\) with the clause \(\Gamma'_3 \rightarrow \Delta'_3\). Note
that reductions can actually be used to delete clauses, if there are no conclusions.

For the introduction and proofs of the properties of our label discipline, sorts, ordering or
selection restrictions are not of importance. Therefore, we leave them out of the presentation
of the superposition inference and reduction rules. They can all be added in a straightforward
way.

To each superposition clause \(\Gamma \rightarrow \Delta\) we add a label \(m\) resulting in 
\(m: \Gamma \rightarrow \Delta\). Then
the standard calculus rules are extended by conditions and operations on the labels. We use a
binary operation \(\circ\) to combine labels for inferences, and a binary operation \(\bullet\) to combine labels
for reductions. Both operations are commutative and associative. We use \(\ominus\) as a special label
to indicate when an inference or a reduction should be blocked.\(^1\) Finally, a preorder, \(\leq\), is used
to define when labels are compatible for clause deletion.

The interpretation of the labels and label operators depends on the instantiation of the
superposition-with-labels calculus. We will give examples in Section 4.2. In particular, the
standard superposition calculus is obtained if all clauses are labelled by the set \(\{1\}\) and \(\circ, \bullet, \leq,\) and \(\ominus\) are instantiated by the standard set operations \(\cap, \cap, \subset,\) and \(\emptyset,\) respectively.

Below we present the extended inference rule superposition right. The missing binary
inference rules superposition left, merging paramodulation, and ordered resolution are defined
accordingly.

**Definition 4.1.1 (Superposition Right)** The inference
\[
\text{J} \quad m_1 : \Gamma_1 \rightarrow \Delta_1, l \approx r \quad m_2 : \Gamma_2 \rightarrow \Delta_2, s[l']_p \approx t \\
(m_1 \circ m_2 : \Gamma_1, \Gamma_2 \rightarrow \Delta_1, \Delta_2, s[p/r] \approx t)\sigma
\]

where

1. \(m_1 \circ m_2 \neq \ominus\)
2. \(\sigma\) is the mgu of \(l'\) and \(l\)
3. \(l'\) is not a variable

and the usual ordering/selection restrictions apply is called a superposition right inference.

For the unary inference rules equality resolution, ordered factoring, and equality factoring
on a clause labelled with \(m\), the resulting clause is labelled with \(m \circ m\), and the rule has an
extra condition that \(m \circ m \neq \ominus\). For inference rules with multiple premises, such as hyper
resolution, the label computations are straightforward extensions of the binary case.

**Definition 4.1.2 (Ordered Hyper Resolution)** The inference
\[
\text{J} \quad m_1 : E_1, \ldots, E_n \rightarrow \Delta \quad m_{2,i} : \rightarrow \Delta_i, E'_i \ (1 \leq i \leq n) \\
(m_1 \circ m_{2,1} \circ \ldots \circ m_{2,n} : \rightarrow \Delta, \Delta_1, \ldots, \Delta_n)\sigma
\]

where

1. \(m_1 \circ m_{2,1} \circ \ldots \circ m_{2,n} \neq \ominus\)
2. \(\sigma\) is the simultaneous mgu of \(E_1, \ldots, E_n, E'_1, \ldots, E'_n\),

and the usual ordering restrictions apply is called an ordered hyper resolution inference.

\(^1\)We require that, for any label \(m, \circ \circ m = \circ\) and \(\ominus \bullet m = \ominus\).
Below we present matching replacement resolution, weak contextual rewriting, and subsumption deletion as examples of how to extend reduction rules by labels. In contrast to the standard superposition calculus, one or more premises of a reduction rule are typically retained. The reason for this is illustrated by the application of labelled clauses to splitting (see Sect. 4.2.2), where the label describes the clause splittings on which a given clause depends. Here it is clear that when a reduction is performed from a clause $C_1$ to a clause $C_2$ that depends on fewer splittings, we must keep $C_1$.

**Definition 4.1.3 (Matching Replacement Resolution)** The reduction

$$
\begin{array}{c}
\rightarrow \\
\Gamma_1 \rightarrow \Delta_1, E_1 \quad \Gamma_2 \rightarrow \Delta_2 \\
\end{array}
$$

$$
\begin{array}{c}
m_1 : \Gamma_1 \rightarrow \Delta_1, E_1 \\
m_2 : \Gamma_2 \rightarrow \Delta_2 \\
m_1 \cdot m_2 : \Gamma_2 \rightarrow \Delta_2 \\
\end{array}
$$

where

1. $m_1 \cdot m_2 \neq \emptyset$
2. $E_1 \sigma = E_2$
3. $(\Gamma_1 \rightarrow \Delta_1) \sigma \text{ subsumes } \Gamma_2 \rightarrow \Delta_2$, where all variables in the co-domain of $\sigma$ are treated as constants for both clauses

is called matching replacement resolution.

This presentation of the matching replacement resolution rule is non-standard, because the parent clause $m_2 : \Gamma_2, E_2 \rightarrow \Delta_2$ is kept. However, in many applications it can then be subsequently deleted by subsumption deletion (see below). For example, in the case of simulating the standard calculus where all labels are identical to $\{1\}$, the clause $\{1\} : \Gamma_2, E_2 \rightarrow \Delta_2$ is always subsumed by $\{1\} \cap \{1\} : \Gamma_2 \rightarrow \Delta_2$, yielding the standard rule.

We present a variant of the rewriting rule; other variants are modified similarly. Two versions of the rule are supplied. The first one is a reduction that can be used when the resulting label is smaller than the parent clause to be deleted. In the second version, the clause is simplified, but the parent clause is not deleted.

**Definition 4.1.4 (Weak Contextual Rewriting)** The reductions

$$
\begin{array}{c}
\rightarrow \\
\Gamma_1 \rightarrow \Delta_1, s \approx t \quad \Gamma_2 \rightarrow \Delta_2, E[\sigma']_p \\
\end{array}
$$

$$
\begin{array}{c}
m_1 : \Gamma_1 \rightarrow \Delta_1, s \approx t \\
m_2 : \Gamma_2 \rightarrow \Delta_2, E[p/t\sigma] \\
m_1 \cdot m_2 : \Gamma_2 \rightarrow \Delta_2, E[p/t\sigma] \\
\end{array}
$$

and

$$
\begin{array}{c}
\rightarrow \\
\Gamma_1 \rightarrow \Delta_1, s \approx t \quad \Gamma_2 \rightarrow \Delta_2, E[\sigma']_p \\
\end{array}
$$

$$
\begin{array}{c}
m_1 : \Gamma_1 \rightarrow \Delta_1, s \approx t \\
m_2 : \Gamma_2 \rightarrow \Delta_2, E[\sigma']_p \\
m_1 \cdot m_2 : \Gamma_2 \rightarrow \Delta_2, E[p/t\sigma] \\
\end{array}
$$

where

1. $m_1 \cdot m_2 \neq \emptyset$
2. $s \sigma = s'$
3. $\Gamma_1 \sigma \subseteq \Gamma_2$, $\Delta_1 \sigma \subseteq \Delta_2$
4. For the first variant, $m_1 \cdot m_2 \leq m_2$
5. For the second variant $m_1 \cdot m_2 \not\leq m_2$

and the usual ordering restrictions apply are called weak contextual rewriting.
The unary simplification rules, such as trivial literal elimination or condensation, are handled similarly to unary inference rules; i.e., given a clause labelled with \( m \), the resulting clause is labelled with \( m \cdot m \) and the rule has an extra condition that \( m \cdot m \neq \emptyset \).

For rules that actually delete clauses, such as tautology deletion and subsumption deletion, we need to guarantee the compatibility of labels, as shown for subsumption deletion below. Tautology deletion is never blocked by the labels because \( \leq \) is reflexive.

**Definition 4.1.5 (Subsumption Deletion)** The reduction

\[
\frac{m_1 : \Gamma_1 \rightarrow \Delta_1 \quad m_2 : \Gamma_2 \rightarrow \Delta_2}{m_1 : \Gamma_1 \rightarrow \Delta_1}
\]

where

1. \( m_2 \leq m_1 \)
2. \( \Gamma_2 \rightarrow \Delta_2 \) is subsumed by \( \Gamma_1 \rightarrow \Delta_1 \)

is called subsumption deletion.

### 4.2 Instantiations

#### 4.2.1 Multiple Labelled Conjectures

The starting point are \( n \) different proof obligations \( \Psi_1, \ldots, \Psi_n \) (dependent or independent) with respect to a theory \( \Phi \). We want to check for which \( i \mid \Phi \models \Psi_i \) holds. Labels are subsets of \( \{1, \ldots, n\} \). All clauses resulting from \( \Phi \) receive the label \( \{1, \ldots, n\} \), and all clauses resulting from \( \Psi_i \) receive the label \( \{i\} \). The label indicates for which proof obligations the clause may be used. The operations \( \circ, \bullet, \leq, \) and \( \emptyset \) are instantiated by the standard set operations \( \cap, \cap, \subseteq, \) and \( \emptyset \), respectively.

**Proposition 4.2.1** From \( \Phi \land \neg \Psi_i \) we can derive \( \rightarrow \square \) by superposition iff we can derive \( \{i\} : \rightarrow \square \) by labelled superposition.

**Proof:** (Sketch) By induction on the length of the superposition derivation doing a case analysis over the different rules. For all labelled inference and simplification rules it holds that the conclusion of a rule is labelled with a number \( i \) iff all premises are labelled with \( i \). If a clause can be removed by labelled subsumption deletion, there exists a more general clause labelled with a superset, i.e., subsumption deletion can be applied to the standard subproofs.

### Refinements

Labelled subsumption deletion is actually weaker than subsumption deletion. For example, in the form shown in Definition 4.1.5 it does not enable subsumption of axiom clauses by conjecture clauses. Furthermore, the calculus considered so far for multiple labelled conjectures does not consider sharing of clauses resulting (from inferences) from different conjectures. Both problems can be overcome by adding the following rules to the calculus.
4.2. Instantiations

**Definition 4.2.2 (Join)** The reduction

\[
R \quad m_1 : \Gamma_1 \rightarrow \Delta_1 \quad m_2 : \Gamma_2 \rightarrow \Delta_2 \\
\quad m_1 \cup m_2 : \Gamma_1 \rightarrow \Delta_1
\]

where
1. \(m_1 \neq \emptyset \) and \(m_2 \neq \emptyset\)
2. \(\Gamma_2 \rightarrow \Delta_2\) and \(\Gamma_1 \rightarrow \Delta_1\) are variants\(^2\)

is called join.

**Definition 4.2.3 (Subsumption Separation)** The reduction

\[
R \quad m_1 : \Gamma_1 \rightarrow \Delta_1 \quad m_2 : \Gamma_2 \rightarrow \Delta_2 \\
\quad m_1 : \Gamma_1 \rightarrow \Delta_1 \\
\quad m_2 \setminus m_1 : \Gamma_2 \rightarrow \Delta_2
\]

where
1. \(m_1 \cap m_2 \neq \emptyset\)
2. \(m_2 \not\subseteq m_1\)
3. \(\Gamma_2 \rightarrow \Delta_2\) is subsumed by \(\Gamma_1 \rightarrow \Delta_1\)

is called subsumption separation.

Subsumption separation can be built into other reduction rules that rely on subsumption. For example, when \(m_2 \not\subseteq m_1\) we can turn matching replacement resolution, Definition 4.1.3, into a reduction, keeping the first parent clause and reducing the second parent clause to \(m_2 \setminus m_1 : \Gamma_2, E_2 \rightarrow \Delta_2\).

Note that using labels allow us to perform reductions of the axiom clauses using conjecture clauses. This would not be possible when using extra predicates for handling multiple conjectures.

When adding the Join rule, the correctness claim needs to be refined as well.

**Proposition 4.2.4** From \(\Phi \land \neg \Psi\) we can derive \(\square\) by superposition iff we can derive \(m : \rightarrow \square\) by labelled superposition for some label \(m\) that contains \(i\).

When deriving an empty clause with a label that contains a certain conjecture, we know that the conjecture is valid given the axiom set. It is sometimes useful to then clausify the conjecture and add the result as new axioms to the axiom set so that it can be used by the other conjectures not yet proven.

**Reusing Skolem Functions.** We can use the labels to devise a better Skolemization process for our setting that results in more clauses being shared between the different proof obligations and thereby avoids duplicate work. Our experimental results is Section 4.3.3 show that this is valuable in practice. The idea is that two Skolem functions of the same arity can be merged if all the clauses they appear in have disjoint labels: any inference between such clauses is blocked, and thus the terms containing the merged functions will never be unified.

\(^2\)Equal with respect to variable renaming.
In general, we would want to merge Skolem functions in a way that allows the Join rule to share the most active clauses. Lacking a way to predict this, we use a heuristic that ensures that at least some clauses will be shared as a result of the process. The heuristic searches for clauses that will be variants after Skolem functions are merged. The heuristic guarantees that the merge is allowed by maintaining for each symbol the set of conjectures it is currently used in.

Example 4.2.5 Let conjecture 1 be \( \forall v . p(v) \lor q(v) \) and conjecture 2 be \( \forall v . p(v) \lor \neg q(v) \). The usual Skolemization of the negated conjectures would result in the following four clauses:

\[
\begin{align*}
\{1\} & : p(c_1) \rightarrow, \{1\} : q(c_1) \rightarrow, \{2\} : p(c_2) \rightarrow, \{2\} : \rightarrow q(c_2). \\
\end{align*}
\]

However, by sharing Skolem constants the following clauses suffice:

\[
\begin{align*}
\{1\}, \{2\} & : p(c_1) \rightarrow, \{1\} : q(c_1) \rightarrow, \{2\} : \rightarrow q(c_1).
\end{align*}
\]

Note that although \( \{1\} : q(c_1) \rightarrow \) and \( \{2\} : \rightarrow q(c_1) \) are resolvable if labels were ignored (resulting in the empty clause), they will never be resolved because their labels are disjoint. As the number and complexity of the conjectures increases, more of the shared structures can be used.

Conjunction Conjectures. When trying to prove a conjecture composed of a large conjunction of formulas, the standard practice of negation and then Skolemization creates large clauses and is typically intractable. Instead, the conjunction can be split into a conjecture per conjunct and processed using the refined calculus above. Now, proving a contradiction for all the labels is equivalent to proving the validity of the original formula.

Proposition 4.2.6 From \( \Phi \land \neg (\bigwedge_{1 \leq i \leq n} \Psi_i) \) we can derive \( \rightarrow \square \) by superposition iff we can derive \( \{1, \ldots, n\} : \rightarrow \square \) by labelled superposition for multiple labelled conjectures.

Theory Consistency. When proving that \( \Phi \rightarrow \Psi \) is valid, it is possible that the reason for the validity is that \( \Phi \) is inconsistent. In the framework of the standard superposition calculus we cannot distinguish between the two cases without inspecting proofs. However, it is easy to support such a consistency check in the context of labelled superposition. We simply add to the label of the axioms a new number 0 resulting in the extended set \( \{0, \ldots, n\} \) for \( n \) conjectures. Then the theory is inconsistent iff we can derive \( m : \rightarrow \square \) for some label \( m \) that contains 0.

Implementation

We have a prototype implementation of the labelled clauses calculus for multiple conjectures within the SPASS theorem prover (available at [LAWRS07a]). Labels were implemented using bit-vectors attached to each clause. The inference and reduction rules were modified to correctly maintain the labels during derivations. The Join rule and Skolem function reuse were also implemented.

The most challenging part in modifying SPASS to support labelled clauses is updating the forward reduction rules, which can now generate many clauses instead of only reducing the given clause. We used a naive approach for implementing these reductions and yet the result is still effective, as can be seen in Section 4.3.3. Backward reduction rules were modified to perform separation to correctly handle, for example, reduction of axiom clauses by conjecture clauses.

We can prevent one conjecture from starving the rest by once in a while selecting a clause from a conjecture that was missing from the labels of the recently selected given clauses. We
have implemented this idea as a runtime option in the new version, but do not yet have experimental results concerning its effectiveness.

4.2.2 Labelled Splitting

Let us consider how labels can be used to model splitting with a single conjecture. For splitting we use a different type of label: sequences of (overlined) clauses with an extra label $\ominus$. We use the labels to record the path in the derivation tree of splits required to generate the clause. We say that $m_1 \leq m_2$ when $m_2$ is a prefix of $m_1$, or $m_1 = \ominus$. The combine operations are simply the greatest lower bound of $\leq$, i.e., $m_1 \circ m_2 = m_1$ if $m_1 \leq m_2$, $m_1 \circ m_2 = m_2$ if $m_2 \leq m_1$, and $m_1 \circ m_2 = \ominus$ otherwise (we define $\bullet$ to be the same as $\circ$). Initially, all clauses are labelled with the empty sequence, denoted by $\epsilon$.

In addition to the labelled superposition rules the extra rules implementing labelled splitting are:

Definition 4.2.7 (Splitting) *The inference*

$$
\frac{}{m : \Gamma_1, \Gamma_2 \rightarrow \Delta_1, \Delta_2}
\frac{}{m, \Gamma_1, \Gamma_2 \rightarrow \Delta_1, \Delta_2 : \Gamma_1 \rightarrow \Delta_1}
\frac{}{m, \Gamma_1, \Gamma_2 \rightarrow \Delta_1, \Delta_2 : \Gamma_2 \rightarrow \Delta_2}
$$

*where*

1. $\text{vars}(\Gamma_1 \rightarrow \Delta_1) \cap \text{vars}(\Gamma_2 \rightarrow \Delta_2) = \emptyset$
2. $\Delta_1 \neq \emptyset$ and $\Delta_2 \neq \emptyset$

*is called splitting.*

Definition 4.2.8 (Branch Closing) *The inference*

$$
\frac{}{m.C \rightarrow \Box}
\frac{}{m.C \rightarrow \Box}
\frac{}{m : \rightarrow \Box}
$$

*is called branch closing.*

Proposition 4.2.9 *From a clause set $N$ we can derive a contradiction by superposition with splitting iff we can derive $\epsilon : \rightarrow \Box$ by labelled superposition with splitting.*

*Proof*: (Sketch) By induction on the length of the superposition derivation, via a case analysis over the different rules. The Splitting rule needs special consideration: we have to show that no inference (reduction) between two clauses $m.C.m_1 : \Gamma_1 \rightarrow \Delta_1$ and $m.C.m_2 : \Gamma_2 \rightarrow \Delta_2$ is possible. This holds by the definitions of the two combination operations ($\circ$ and $\bullet$): combining any two sequences $m.C.m_1$ and $m.C.m_2$ results in $\ominus$. On the other hand, the combination of $m.C$ (or $m.C$) and any prefix of $m$ results in $m.C.m.C$, which corresponds to the standard clause set copy and separation of the standard splitting rule.
Refinements

Once the labels are available and the different branches of the derivation tree spanned by the splitting rule can be investigated simultaneously, it is easy to define and employ the well-known refinements for splitting and tableau proofs. By studying the labels of clauses used in the derivation of an empty clause, refinements like branch condensing (implemented in SPASS, splittings from the most recent backtracking empty clause that did not contribute to the current empty clause can be removed from the label) or the generation of more suitable “backtracking clauses” (see e.g., [NO05]) are straightforward to integrate. The lemma-generation rules invented in the context of tableau [Häb01], for example, the folding up rule can also be integrated.

An obvious refinement of splitting is to add the negation of the first clause part $\Gamma_1 \rightarrow \Delta_1$ to the second part, labelled with the second part (or the other way round). For example, applying this refined version of splitting to the clause $C = m : \rightarrow A, B$ yields $m.C : \rightarrow A, m.\overline{C} : \rightarrow B$, and $m.\overline{C} : A \rightarrow$.

Implementation

In addition to the above mentioned refinements, implementing Splitting via labelled clauses seems to be less effort and as efficient as a standard implementation via a depth-first search and clause copy, as it is done, e.g., in SPASS. For example, branch closing becomes a standard inference/reduction rule application of the calculus while in the context of the depth-search algorithm implemented in SPASS it is a procedure running orthogonal to the standard saturation process, complicating the overall implementation.

4.2.3 Strategies

Labelled clauses can also be used to model well-known strategies and new strategies. For example, the set-of-support strategy forbids inferences between axiom clauses. This can be easily established by the labelled superposition framework using the label set $\{0, 1, \overline{\circ}\}$ and labelling all axiom clauses with 0, all conjecture clauses with 1. We instantiate $m_1 \leq m_2$ to always be true and use the following combination operations:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>$\overline{\circ}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\overline{\circ}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$\overline{\circ}$</td>
</tr>
<tr>
<td>$\overline{\circ}$</td>
<td>$\overline{\circ}$</td>
<td>$\overline{\circ}$</td>
<td>$\overline{\circ}$</td>
</tr>
</tbody>
</table>

Thus any inferences between clauses for the axiom set are blocked, but reductions are allowed, keeping the 0 label.

4.2.4 Slicing

When trying to prove a conjecture in a given time limit, it can be a good heuristic to try different strategies for a fixed period of time. For example, strategies might differ in the selection strategy for negative literals (no selection, always select a negative literal, etc.) or the heuristic to pick the next clauses for inferences (lightest weight, heaviest weight, lowest age). Labelled
4.3. Applications in Software Verification

We simply run the same conjecture simultaneously with different labels for the different strategies. Given n strategies, we give each one a number from \(\{1, \ldots, n\}\). Initially we label all clauses with \(\{1, \ldots, n\}\). We extend the labelled superposition calculus of Section 4.1 to also consider the strategy label, say \(m'\) for inference computation. Then the newly generated clause gets the label \(m_1 \circ m_2 \circ m'\) and we have to check the condition \(m_1 \circ m_2 \circ m' \neq \emptyset\). Nevertheless, common clauses can be shared via the Join rule as in the case of multiple labelled conjectures.

Because the reduction (and simplification) combinator \(\bullet\) differs from the inference combinator, it can be used to implement different strategies concerning reductions and simplifications between clauses generated by different strategies. This provides a high potential for synergy between the different proof attempts. Note that because different strategies may use different orderings, by considering the strategy label at inference rule level, the ordering used by the rule conditions can also be chosen by that label.

4.3 Applications in Software Verification

We present two applications that motivated our interest in labelled clauses. Both concern the application of abstract interpretation in software verification.

4.3.1 Cartesian Abstraction

Predicate abstraction [GS97] abstracts a program into a Boolean program that conservatively simulates (i.e., overapproximates) all potential executions of the original program. Thus, every safety property that holds for the Boolean program is guaranteed to hold for the original program. A predicate \(p\) in predicate abstraction is defined by a closed formula \(\psi_p\) that evaluates to true or false for each concrete program state. An abstract state is a conjunction of (possibly negated) predicates; it represents the set of concrete states that satisfy each (possibly negated) predicate. A set of abstract states corresponds to a disjunction of the individual abstract states. The best approximation of a set of concrete states \(C\) is the strongest set of abstract states such that every \(c \in C\) is satisfied (or, equivalently, the strongest Boolean combination of predicates that holds for every \(c \in C\)).

The operational semantics of a statement \(st\) in a program can be defined using a formula \(\tau_{st}\) over the pre-state and the post-state. A post-state \(S'\) can be the result of a given pre-state \(S\) if \(\tau_{st}\) holds for \(S \cup S'\). The predicate \(p'\) defined using formula \(\psi_p\), but applied to the post-state, is called the primed version of \(p\). The effect of \(st\) on an abstract pre-state \(A\) can be computed using a theorem prover by checking which combinations of primed-predicate formulas are satisfiable when conjoined with \(A \land \tau_{st}\).

A less-costly, but less-precise, variant of predicate abstraction is called Cartesian Abstraction [BPR01]. Instead of using a Boolean combination of predicates to represent a set of states \(C\), this uses a single conjunction of (possibly negated) predicates. Thus, a predicate \(p\) should appear in abstract state \(A_C\) (i) positively when \(\psi_p\) holds for every \(c \in C\); (ii) negatively when \(\psi_p\) does not hold for any \(c \in C\); and (iii) otherwise should not appear.
With Cartesian Abstraction, it is possible to compute the effect of a statement via a set of validity queries to a theorem prover. In each query, the axiom set contains the pre-state $A$, the transformer $\tau_{st}$, and any background theory we have. The conjecture is either a primed predicate or a negated primed predicate. This makes computing transformers for Cartesian Abstraction a good target for our method. All the primed predicates and their negations can be added as conjectures to the same axiom set, and the effect of the statement can be computed using a single call to the theorem prover; any shared structure between the different predicates or similarities among the proofs of the different conjectures will be exploited by the theorem prover.

### 4.3.2 Shape Analysis and Canonical Abstraction

We give a brief informal introduction to Canonical Abstraction [SRW02]. For more information see Section 2.5.

In Canonical Abstraction, the heap is partitioned according to conjunctions of unary predicates. Only universal information about these partitions is saved; that is, if formulas such as $node_1(v)$, $node_2(v)$, etc. are conjunctions of unary predicates that characterize the various subsets that partition the heap-cells, and $p$ is a binary predicate symbol from the vocabulary, we can only record information of the form

$$\forall v_1, v_2. node_i(v_1) \land node_j(v_2) \implies [\neg] p(v_1, v_2). \quad (4.1)$$

To control the abstraction, the designer of an abstraction (or an automatic tool for discovering abstractions [LRS05]) can define auxiliary unary predicates with FO(TC) formulas that are added to the background theory.

Similar to Section 4.3.1, the operational semantics of a statement is defined using a formula over the pre-state and the post-state; to determine the effect of a statement, we need to determine which formulas in form Eq. (4.1) hold for the primed predicates. In this setting, it is again possible to compute the effect of a statement via a set of validity queries, which may be answered en masse by a theorem prover using our method. This allows the theorem prover to exploit similarities among the proofs of the different conjectures.

### 4.3.3 Experiments

We have integrated the prototype implementation of labelled clauses for multiple conjectures in SPASS into TVLA, i.e., the validity queries required to compute the result of applying a transformer are performed by SPASS. In this section, we present a comparison between the performance of the improved version and running the original SPASS version 2.2 on each conjecture sequentially. The results presented here are from the queries generated by the analysis of several heap-manipulating programs, including reversal of a singly-linked list, insert sort of a singly-linked list, insertion into a doubly-linked list, and the mark phase of a simple mark-and-sweep garbage collector.

---

3See Chapter 3 for methods that can be used to handle formulas that involve the TC operator.

4One technical point: case splits are performed externally to the theorem prover (and before the theorem prover is called), by the “focus” operation of [SRW02], hence we do not have to be concerned about disjunctions of the different conjectures.
4.3. Applications in Software Verification

Table 4.1: Comparison between the running times of labelled-clauses SPASS and the original SPASS sequentially.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Sequential</th>
<th></th>
<th></th>
<th>Labelled Clauses</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Derived</td>
<td>3569</td>
<td>408837</td>
<td>83657.1</td>
<td>1408</td>
<td>140679</td>
<td>33088.5</td>
</tr>
<tr>
<td>Kept</td>
<td>3882</td>
<td>70463</td>
<td>22672.1</td>
<td>1366</td>
<td>26238</td>
<td>7357.2</td>
</tr>
<tr>
<td>Time (sec)</td>
<td>1.0</td>
<td>74.0</td>
<td>8.1</td>
<td>0.2</td>
<td>29.0</td>
<td>3.8</td>
</tr>
</tbody>
</table>

To make the comparison fair, we removed three types of queries: (i) any conjecture that is not valid, (ii) queries in which the common axioms are inconsistent, and (iii) queries in which there is only one valid conjecture. When also considering invalid conjectures, the advantage of the labelled-clauses method is even more apparent because the sequential approach has to wait for a timeout for each invalid conjecture. Similarly, when considering queries with an inconsistent axiom set, the improved version can detect that the original axioms are inconsistent, while the sequential approach will have to prove it again and again for each conjecture. Finally, for a single conjecture the labelled-clauses version behaves almost exactly the same as the original version, both in terms of the clauses generated and the time required.

The test set includes 125 queries, with a minimum of 2 conjectures, a maximum of 32 conjectures, and an average of 9.3 conjectures per query. To improve the statistical significance of the results, we took to the test set only queries in which the total time taken by the sequential prover was less than one second. We compare the provers according to three criteria: the number of derived clauses, the number of kept clauses (i.e., which were not forward subsumed), and the time. Table 4.1 has a comparison of the maximum, minimum, and average values for these three criteria for the two provers. The experiments were run on a 2.4GHz E6600 Core 2 Duo processor with 2 GB of memory running Linux.

Figure 4.1 presents a histogram for each criterion showing the values of the labelled-clauses prover as a percentage of the appropriate value for the sequential prover. In most cases we have at least 2-fold improvement in all criteria. The number of derived clauses increases in one case, but only by 6%. There are three examples in which the sequential version was faster; we believe that this is a result of the quality of the current prototype implementation, because the number of derived and kept clauses in these cases is lower in the labelled-clauses version.

Because our conjectures are based on formulas in the normal form shown in Eq. (4.1), the negated conjectures are sets of ground unit clauses. This makes the technique of reusing Skolem constants between conjectures very lucrative. We have checked the improvements brought about by the Join rule and by reusing the Skolem constants. Introducing the Join rule to a vanilla implementation of the labelled-clauses approach makes it run 1.7 times faster. Reusing Skolem constants added, on average, an extra speedup of 1.7, which produced about a 3-fold speedup in total. When considering only cases in which 10 or more conjectures are used, the average speedup is 7-fold.

The use of theorem provers for computing transformers for abstract interpretation is promising. However, even with the techniques presented in this chapter, performance is a major issue. In the second part of the thesis we explore an alternative approach: design abstract domains for shape analysis in which efficient transformers can be computed directly.
Figure 4.1: Histograms for each criterion showing the percentage of the values for the labelled-clauses version compared with the sequential version: (a) derived clauses, (b) kept clauses, (c) running time.
Chapter 5

Specialized Shape Analysis for Commonly Used Data Structures

This chapter addresses the problem of proving safety properties of imperative programs manipulating dynamically allocated data structures using destructive pointer updates. We present a new abstraction for linked data structures whose underlying graphs do not contain cycles. The abstraction is simple and allows us to decide reachability between dynamically allocated heap cells.

We present an efficient algorithm that computes the effect of low level heap mutations in the most precise way. The algorithm does not rely on the usage of a theorem prover. In particular, the worst case complexity of computing a single successor abstract state is $O(V \log V)$ where $V$ is the number of program variables. The overall number of successor abstract states can be exponential in $V$. A prototype of the algorithm was implemented and is shown to be fast.

Our method also handles programs with “simple cycles” such as cyclic singly-linked lists, (cyclic) doubly-linked lists, and trees with parent pointers. Moreover, we allow programs which temporarily violate these restrictions as long as they are restored in loop boundaries.

Outline

- We start with preliminaries and definitions in Section 5.1,
- In Section 5.2 we define the abstraction,
- In Section 5.3 we show how to compute most precise abstract transformers for the abstract domain,
- In Section 5.4 we define information extraction from the domain via formula evaluation,
- In Section 5.5 we explore extensions to the abstract domain,
- Finally, in Section 5.6 we explain the details of the implementation and provide an empirical evaluation.
Chapter 5. Shape Analysis for Commonly Used Data Structures

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E^*$</td>
<td>Reflexive Transitive Closure of $E$</td>
<td>Restriction of first component</td>
</tr>
<tr>
<td>$\text{succ}(X,E)$</td>
<td>${(n,n') \in E \mid n \in X}$</td>
<td>Restriction of second component</td>
</tr>
<tr>
<td>$\text{pred}(X,E)$</td>
<td>${(n,n') \in E \mid n' \in X}$</td>
<td>Relation composition</td>
</tr>
<tr>
<td>$E_1 \circ E_2$</td>
<td>${(n,n'') \mid (n,n') \in E_2, (n',n'') \in E_1}$</td>
<td>Relation image</td>
</tr>
<tr>
<td>$E{X}$</td>
<td>${n' \mid (n,n') \in \text{succ}(X,E)}$</td>
<td>Relation image</td>
</tr>
<tr>
<td>$\text{up}_{b\rightarrow a}$</td>
<td>$\lambda n. \text{if } (n=b) \text{ then } a \text{ else } n$</td>
<td>Updating $b$ to be $a$</td>
</tr>
<tr>
<td>$\text{fld}^C$</td>
<td>$\bigcup_{f \in \text{PRef}} \text{ref}^C(f)$</td>
<td>Edges of $C$</td>
</tr>
<tr>
<td>$\text{disj}(v_1,v_2,v_3)$</td>
<td>$v_1 \neq v_2 \land v_1 \neq v_3 \land v_2 \neq v_3$</td>
<td>The variables are disjoint</td>
</tr>
</tbody>
</table>

Figure 5.1: Notations used in the chapter.

5.1 Preliminaries

We call an allocated object on the heap a heap node. Shape analysis tracks reference program variables and reference fields, i.e., to which heap node each reference variable points to and for each heap node where each of its reference fields point to. In this chapter we assume a fixed set of (reference) program variables denoted by $P\text{Var}$ and a fixed set of reference fields denoted by $P\text{Ref}$.

A state (shape graph [JM81]) is a triple $C \triangleq (U^C, \text{env}^C, \text{ref}^C)$. The universe, $U^C$, is the set of allocated heap nodes; the environment, $\text{env}^C \subseteq P\text{Var} \times U^C$, is a partial function from program variables to the heap nodes that they point to; and $\text{ref}^C : P\text{Ref} \rightarrow \mathcal{P}(U^C \times U^C)$ is a function from each field name $f$ to a relation which pairs each node with the node its $f$ field points to. Since these relations induce a graph on the heap nodes, we will use the term $f$-edge for a pair of nodes in the relation $\text{ref}^C(f)$ and call $f$ its edge type. In languages such as Java where the program cannot use the memory address of an object directly, the specific names of the nodes in $U^C$ are immaterial. Thus, we define equality between states as isomorphism between them.

5.1.1 Notations

Figure 5.1 lists some notation used throughout. We shorten $E\{\{x\}\}$ to $E\{x\}$.

We define $\text{var}(S) \triangleq \text{env}^S\{P\text{Var}\}$ to be the set of nodes in $S$ pointed to by program variables and $\text{shared}(S)$ to be that set of nodes in $S$ that are pointed to by two or more different heap nodes (ignoring self-loops). We say such a node is heap-shared. Formally, $\text{shared}(S) \triangleq \{v \mid (w_1,v) \in \text{fld}^S, (w_2,v) \in \text{fld}^S, \text{disj}(w_1,w_2,v)\}$

5.2 Abstraction

A state, $C$, is concrete if none of its edges are self loops and if each $\text{ref}^C(f)$ is a partial function. The main idea of the abstraction is to keep a set of distinct nodes which are not abstracted and abstract the rest of the graph in such a way that keeps all reachability information for these nodes explicit. The set of distinct nodes we use are those nodes that are either pointed to by variables or heap shared, i.e., $\text{distinct}(S) \triangleq \text{var}(S) \cup \text{shared}(S)$. 
5.2. Abstraction

We contract an edge \((a, b)\) by replacing each occurrence of \(b\) by \(a\), \(\text{contract}(S, a, b) \overset{\text{def}}{=} (U^S - \{b\}, env^S, \lambda f. \{ (up_{b\rightarrow a}(n_1), up_{b\rightarrow a}(n_2)) \mid (n_1, n_2) \in ref^S(f) \})\) (note that \(env^S\) is not updated because we never contract a node pointed to by a variable). We now define a method \(B(S, D)\) that given a state and a set of nodes \(D\) s.t. \(\text{distinct}(S) \subseteq D \subseteq U^S\), returns the abstract state generated by repeatedly applying contraction on all edges that are not incident to nodes in \(D\) until the unique fixed-point is reached. An equivalent way to define \(B(S, D)\) is by collapsing every maximal connected subgraph \(T_n\) of \(S\) that does not contain nodes in \(D\) (the subgraph is a rooted tree) to a single node \(n\) (its root). The edge types of the self-loops of \(n\) are exactly the types of edges within \(T_n\).

We call the function, \(M\), that maps each node to the node it was collapsed into by \(B\) the embedding function (after [SRW02]). When multiple nodes have been embedded into a single node \(n\) (i.e., \(|M^{-1}(n)| > 1\)) we call \(n\) a summary node. Figure 5.2 gives an example of a concrete state \(C_1\) and the result of \(B(C_1, \text{distinct}(C_1))\). We mark summary nodes with a double-circle for emphasis.

The abstraction relation, \(\beta \overset{\text{def}}{=} \{ (S, B(S, \text{distinct}(S))) \mid S \text{ a state} \}\), maps each state, \(S\), to a state in which every edge not incident to a distinct node has been contracted.

5.2.1 Data Structures

We limit the class of data structures handled to graphs with no undirected cycles (i.e., when we remove the direction of the edges we get an undirected forest) and no garbage (i.e. all nodes are reachable from program variables). We call such states admissible states. This class includes linked lists, trees, and trees with limited amount of sharing (i.e., each pair of nodes has at most one simple path between them and each pair of variables meets at most once). Extensions to support cyclic linked lists, doubly linked lists and trees with parent pointers are described in Section 5.5.

We use a standard relational abstract domain with set-union as join (in Section 5.5 we define a more concise partial-join operator). The concretization relation is defined as \(\gamma \overset{\text{def}}{=} \{(S, C)\mid (C, S) \in \beta \text{ and } C \text{ is an admissible concrete state}\}\). We say that an abstract state, \(S\), is feasible if \(\gamma\{S\} \neq \emptyset\), i.e. \(S\) models some admissible concrete state.

5.2.2 Properties of the Abstraction

We start with some important definitions:

![Figure 5.2: (a) A concrete state \(C_1\), (b) \(S_1 = B(C, \text{distinct}(C_1))\)](image)
Theorem 5.2.1  For every feasible abstract state \( S \) the following hold:
1. Every \( f \) edge in \( S \) is an \( f \) may edge.
2. Every non self-loop \( f \) edge is an \( f \) unique may edge.
3. Every \( f \) edge between non-summary nodes is an \( f \) must edge.
4. A node in \( S \) is a summary node iff it has self-loops.
5. For every summary node \( n \) the subgraph induced by \( M^{-1}(n) \) is a tree and has a unique incoming edge which leads to its root.
6. Let \( n_1 \neq n_2 \) where \( n_1 \) has no self-loops or a single self-loop of the same type as its outgoing edge. A path from \( n_1 \) to \( n_2 \) is a must path.

Proof: (sketch)
1. Immediate from definition of contraction.
2. Analysis of possible contractions reveals that the only case in which two edges are merged by a contraction is if an undirected cycle appeared in the original state.
3. Immediate from 4 and the definition of contraction.
4. Contraction always creates a self-loop. Self-loops are preserved by contraction and contraction is the only way to create self-loops.
5. Let \( T_n \) be the subgraph induced by \( M^{-1}(n) \). Since contraction is done on edges, the nodes in \( T_n \) are weakly connected. Shared nodes are never contracted, thus there is no sharing in \( T_n \). Since the original state had no garbage any cycle either has a variable pointing to it, or
5.2. Abstraction

has a shared node. In any case, an entire cycle cannot be contracted to the same summary node. Thus, \( T_n \) is a tree. Furthermore, to avoid sharing and garbage, the one and only incoming edge must be to its root.

6. By 5 and 1, every summary node represents a tree and every edge is a may-edge. Thus, paths between non-summary nodes are must paths. Since a summary node is a tree, all the nodes in it are reachable from the root and so if the target node is a summary node, the path is still a must path. If the source node has a single self-loop it is a singly-linked list. The only outgoing edge from a singly-linked list of the same type as the self-loop is from its last node, thus reachable from all nodes.

The last property is of particular importance since it means that the reachability information in the abstract state is explicit. This property is not standard in shape analysis abstractions (e.g., in TVLA it is not always the case). The reason for the limitation on \( n_1 \) is that if \( n_1 \) has 2 or more self-loops it embeds a tree, thus \( n_2 \) is not reachable from some nodes embedded to \( n_1 \) (e.g. in Figure 5.2 the path in \( S_1 \) from node 3 to node 4 is not a must path, since for example in \( C_1 \) there is no path from node 5 to node 4).

Lemma 5.2.2 defines when an abstract state \( S \) is feasible and Lemma 5.2.3 bounds its size. Note that the set of admissible concrete states is exactly the set of feasible abstract states with no self-loops.

**Lemma 5.2.2 (Feasibility)** Abstract state \( S \) is feasible iff the following hold:

1. There are no edges between two different non-distinct nodes.
2. Distinct nodes are never summary nodes.
3. A node that has two outgoing \( f \) edges has a self-loop of a different edge type.
4. Deleting all self-loops from \( S \) makes it admissible.

**Proof:** (sketch)

(Only If) 1. An edge between two different non-distinct nodes can be contracted, which contradicts that \( S \) is in the image of \( \beta \).

The rest of the properties hold in concrete admissible states and are preserved by contraction.

2. Immediate from definition of contraction.

3. A counterexample would be a node with zero or one self-loops and two outgoing edges of the same type. Since in the original concrete state each edge is a partial function, a node without self-loops cannot have two outgoing edges of the same type. A node with a single self-loop is a linked list, thus the only outgoing edge from it can be from its tail, thus a single edge.

4. It is easy to see that contractions do not introduce garbage or undirected cycles (except for self-loops).

(If) It can be shown that a state that satisfies these properties can always be expanded to a concrete state of finite size.

**Lemma 5.2.3 (MaxSize)** For every feasible abstract state \( S \) we have \( |U^S| \leq \text{MaxSize} \), where \( \text{MaxSize} \) is defined as \( (|PRef| + 1) \times (2 \times |PVar| - 1) \).

**Proof:** Let \( C \) be an admissible concrete state, \( D \) a set s.t. \( \text{var}(C) \subseteq D \subseteq U^C \), and \( S = \beta(C, D) \). \( S \) has the property that every node is either in \( D \) or has a parent in \( D \). Thus, the number of nodes in \( |U^S| \leq |D| \times (|PRef| + 1) \). Since \( C \) has no garbage and no undirected cycles, \( |\text{distinct}(C)| \leq |PVar| \times 2 - 1 \). Thus, if \( (C, S) \in \beta \) then \( |U^S| \leq \text{MaxSize} \).
Chapter 5. Shape Analysis for Commonly Used Data Structures

5.3 Most Precise Abstract Transformers

Concrete Semantics Figure 5.4 defines the concrete semantics for simple atomic statements in Java-like programs. Most preconditions were added to simplify the presentation. In practice we use temporaries to translate each program statement to a sequence of operations while maintaining the preconditions. Some preconditions such as no null-dereference cannot be removed by a sequence of operations. The analysis detects violations of these preconditions and gives a warning.

The gc operation performs garbage collection by removing all nodes not reachable from any variable. Garbage collection can be executed either after every $x = \text{null}$ operation, periodically, or we can run garbage detection instead of garbage collection to detect memory leaks. The semantics of the other operations are straightforward formalizations of standard Java-like operational semantics.

5.3.1 Abstract Transformers

We now show how to compute the most precise abstract transformers (see [CC79]) for our abstraction and concrete semantics defined in Figure 5.4. The most precise abstract transformer of an operation $st$ is defined as $st^{\text{best}} \equiv \beta \circ st \circ \gamma$ (i.e., for each concrete state in $\gamma\{S\}$ apply the concrete semantics and abstract). This definition is not constructive since the number of states in $\gamma\{S\}$ is unbounded and potentially infinite. The main idea is to define a relation $\text{focus}[st]$ whose image is a bounded set of states and if $(S, S') \in \text{focus}[st]$ there is a representative state $C \in \gamma\{S\}$ s.t. $\beta\{st'(S')\} = \beta\{st(C)\}$ and vice versa. Thus, we define the abstract transformer to be $st^{\text{st}} \equiv \beta \circ st \circ \text{focus}[st]$. Note that the transformer defined in the concrete semantics can be applied to abstract states as well.

The focus operation is similar to the one defined in [SRW02], i.e., it is a partial concretization intended to restore enough information to compute the transformer precisely. Let $D(st, C) \equiv \text{distinct}(C) \cup \text{distinct}(st(C))$. We define focus to be:

Definition 5.3.1 $\text{focus}[st] \equiv \{(S, B(C, D(st, C))) \mid C \in \gamma\{S\}\}$

### Table 5.4: The operations supported and their concrete semantics.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Precondition</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$gc$</td>
<td></td>
<td>let $R = fld^+(\text{var}(S))$ in $(R, \text{env}, \lambda f.\text{succ}(R, \text{ref}(f)))$</td>
</tr>
<tr>
<td>$x = \text{null}$</td>
<td>$\text{env}{x} = \emptyset$</td>
<td>$(U, \text{env}, \text{succ}(\text{PVar} \setminus {x}, \text{env}), \text{ref})$</td>
</tr>
<tr>
<td>$x = y$</td>
<td>$\text{env}{x} = \emptyset \wedge \text{ref}(f) \circ \text{env}{x} \subseteq \text{var}(S)$</td>
<td>$(U, \text{env}, \text{ref}[f := \text{succ}(U - \text{env}{x}, \text{ref}(f))])$</td>
</tr>
<tr>
<td>$x = y.f$</td>
<td>$\text{env}{x} \neq \emptyset \wedge \text{ref}(f) \circ \text{env}{x} = \emptyset$</td>
<td>$(U, \text{env}, \text{ref}[f := \text{ref}(f) \cup {(n_x, n_y)}</td>
</tr>
<tr>
<td>$x = \text{malloc}$</td>
<td>$\text{env}{x} = \emptyset \wedge n_{\text{malloc}} \notin U$</td>
<td>$(U \cup {n_{\text{malloc}}}, \text{env} \cup {(x, n_{\text{malloc}})}, \text{ref})$</td>
</tr>
<tr>
<td>$x = \text{null}$</td>
<td>$\text{env}{x} = \emptyset$</td>
<td>$\text{env}{x} = \emptyset$</td>
</tr>
<tr>
<td>$x = y$</td>
<td>$\text{env}{x} = \emptyset$</td>
<td>$\text{env}{x} = \text{env}{y}$</td>
</tr>
</tbody>
</table>

Note that the transformer defined in the concrete semantics can be applied to abstract states as well.
Focus takes all the states in $\gamma\{S\}$ and keeps both the distinct nodes of the state and the nodes that will become distinct after the statement is executed. In Section 5.3.2 we define an algorithm that computes the image of focus.

Lemma 5.3.2 gives some important properties for the interaction of $\beta$ and $st$. Note that the existence of commutative diagrams is not true in general shape abstraction. Theorem 5.3.3 uses Lemma 5.3.2 to prove that $st^\sharp$ is the most precise abstract transformer.

**Lemma 5.3.2** For every $(S, C) \in \gamma$, let $D = D(st, C)$ and $S' = B(C, D)$. Then:

**Idempotence** $\beta\{S'\} = \beta\{S\}$,

**Commutative Diagrams** $B(st(C), D) = B(st(S'), D)$, and

**Equivalence under** $\beta (\beta \circ st)\{C\} = (\beta \circ st)\{S'\}$

**Proof:** (sketch)

**Idempotence** It can be shown that contraction induces a confluent derivation relation commutative in the choice of $D$. Since $B$ can be seen as the fixed-point of that relation, the statement follows.

**Commutative Diagrams** This can be verified by checking the algebraic operations defining the transformer, for each operation in Figure 5.4.

**Equivalence under** $\beta$ By commutative diagrams we have $B(st(C), D) = B(st(S'), D)$. By Idempotence we have $\beta\{B(st(C), D)\} = \beta\{st(C)\}$ and $\beta\{B(st(S'), D)\} = \beta\{st(S')\}$. Thus, $\beta\{st(C)\} = \beta\{st(S')\}$.

**Theorem 5.3.3** $st^\sharp$ is the most precise abstract transformer, i.e., $st^\sharp = st^{best}$.

**Proof:** Let $(S, S^\sharp) \in st^\sharp$. There is $S'$ s.t. $(S, S') \in focus[st]$ and $(S', S^\sharp) \in \beta \circ st$. By Definition 5.3.1 there is a concrete and admissible state $C$ s.t. $(S, C) \in \gamma$ and $S' = B(C, D(st, C))$, and by Lemma 5.3.2 $(C, S^\sharp) \in (\beta \circ st)$ thus $(S, S^\sharp) \in st^{best}$.

Conversely, let $(S, S^\sharp) \in st^{best}$. There is $C$ s.t. $(S, C) \in \gamma$ and $(C, S^\sharp) \in (\beta \circ st)$. Let $S' = B(C, D(st, C))$. By Definition 5.3.1 we have $(S, S') \in focus[st]$, and by Lemma 5.3.2 $(S', S^\sharp) \in \beta \circ st$ thus $(S, S^\sharp) \in st^\sharp$.

### 5.3.2 Algorithms

In order to compute the most precise abstract transformer, $st^\sharp$, we must give efficient algorithms for state equality, focus, and $\beta$. The total complexity of computing the abstract transformer is $O(\text{NS} \times V \times \log V)$ where $\text{NS}$ is the number of successor abstract states (which may be exponential in the number of program variables and reference fields).
Figure 5.5: Abstract state used to demonstrate the exponential number of successor states after a focus operation.

Focus. In Section 5.3.1, we defined focus non-constructively. We now present an algorithm, Focus\((S, st)\), that computes focus\([st] \{ S \}\). The first observation is that for all statements, \(st\), except \(x = y.f\), focus\([st]\) is the identity relation. This is clear for \(x = \text{malloc}\), and true for the rest because \(
\text{distinct}(st(S)) \subseteq \text{distinct}(S)\).

For \(st \triangleq x = y.f\), Focus\((S, st)\) enumerates on all states that can be contracted to \(S\) by a minimal number of contractions and still have \(
\text{distinct}(st(S)) \subseteq \text{distinct}(S)\) as non-summary nodes. Let \(n_f\) be the node pointed to by \(y.f\) in \(S\). If it is a non-summary node Focus\((S, st) = \{ S \}\). Otherwise, let \(G\) be the self-loops of \(n_f\) in \(S\). Let \((S, S') \in \text{focus}[x = y.f]\), \(S'\) can be contracted into \(S\) by at most one contraction for each edge type in \(G\). Let \(N'_f\) be the subgraph of \(S'\) that was contracted into \(n_f\). Since all edges are may-edges, the edges within \(N'_f\) are exactly the self-loops of \(n_f\). Furthermore, since all the edges between different nodes are unique may-edges, the edges between \(N'_f\) and the rest of the graph are exactly the edges between \(n_f\) and the rest of the graph. Finally, since \(S'\) is the result of \(B\) on an admissible concrete state the property that a node that has two outgoing \(g\) edges has a self-loop of different reference field, is maintained. This gives us an enumeration algorithm to compute Focus\((S, st)\).

The number of possible successor states for a single input state is exponential in the number of reference fields and the number of program variables. For example, consider the abstract state in Figure 5.5 and the operation \(\text{focus}[z = x.l]\). The number of output states is exponential in the number of reference fields since a successor for each reference field can either exist or not and any subset of the self loops can exist for each of the successors. The number of output states is exponential in the number of program variables since the outgoing edges to \(y_i\) can be divided in all possible combinations between the different successors.

Lemma 5.3.4 summarizes the properties of Focus\((S, st)\).

**Lemma 5.3.4** \(\text{focus}[x = y.f]\{S\} = \text{Focus}(S, x = y.f)\)

Beta. To compute the image of \(\beta\) we perform two tasks, 1) check that the state is admissible and 2) return a state in which all the possible contractions have been made.

Admissibility. Since an admissible state is one without garbage and with no undirected cycles, the check is done by DFS from all nodes pointed to by variables to make sure that there is no garbage. To compute undirected connectivity, we maintain a Union-Find data structure during the DFS, thus detecting undirected cycles. We start with singleton groups for each node
and for every edge we encounter we union the groups the two incident nodes belong to. Thus the sets maintain weak reachability. If we find the two incident nodes already belong to the same group we found an undirected cycle and we abort. The complexity for this check is \(O(n\alpha(n))\), where \(n\) is the size of the input state and \(\alpha\) is the inverse Ackerman function.

To compute \(\beta\{S\}\) we observe that the edges contracted are exactly the edges between non-distinct nodes. Thus, the algorithm performs two DFS traversals. The first computes \(distinct(S)\) by marking nodes that are either pointed to by variables or have an in-degree greater than one (note that self-loops do not contribute to the in-degree). The second traversal simply contracts every non self-loop edge s.t., both its incident nodes are not distinct. The complexity of this algorithm is \(O(n)\).

**State Equality** We defined state equality as isomorphism between the states. We give an algorithm that computes canonical names for each state. The canonical names of two states are identical iff the two states are isomorphic.

Canonical names are given to nodes by traversing the graph in DFS from program variables (in fixed order) traversing the reference fields in fixed order as well. The name of a node \(n\) is composed of the names of the variables pointing to \(n\), \(n\)’s self loops and for each of \(n\)’s parents, the parent name and the type of the edge leading from the parent to \(n\). To ensure the traversal order is unique, we only leave a node to its children after all its parents have been visited. Hash-cons is used to store the canonical names, allowing for \(O(1)\) amortized time equality checks. The name of a state is the hash-cons of its set of nodes ordered by some fixed order (e.g. memory address of the hash-cons). Thus, the total complexity of the algorithm is \(O(V \log V)\).

### 5.4 Evaluation

We use a subset of first-order logic with transitive closure as a query logic to extract information from states. Let \([\varphi]^S\) denote the Boolean value of formula \(\varphi\) in state \(S\).

**Definition 5.4.1 (Sound)** An evaluation function of a formula is sound iff for every feasible abstract state \(S\), \(\neg[\varphi]^S \implies \forall C \in \gamma\{S\}. \neg[\varphi]^C\)

**Complete** An evaluation function of a formula is complete iff for every feasible abstract state \(S\), \(\neg[\varphi]^S \iff \forall C \in \gamma\{S\}. \neg[\varphi]^C\)

To compute \(assert(\varphi, S)\), i.e., to verify that all the states in \(\gamma\{S\}\) satisfy \(\varphi\), we will apply a sound evaluation function on \(\neg\varphi\) and verify that the result is false.

### 5.4.1 Query Logic

The query logic is first order logic in Negation Normal Form (NNF) over the following vocabulary:

- For every \(x \in PVar\) a unary predicate symbol; \(x(n)\) iff \(x\) points to \(n\)
- For every \(f \in PRef\) a binary predicate symbol; \(f(n_1, n_2)\) iff the \(f\) field of the \(n_1\) points to the \(n_2\)
- Binary predicate symbol TC; \(TC(n_1, n_2)\) iff there is any non-empty path from \(n_1\) to \(n_2\)
• Equality; \( n_1 = n_2 \) iff \( n_1 \) and \( n_2 \) are the same heap node

Examples:

\[
\forall v . \exists w.x(w) \land (v = w \lor TC(w, v)) \tag{5.1}
\]

\[
\forall v, w. \neg y(w) \lor \neg left(v, w) \tag{5.2}
\]

Formula (5.1) states that all the nodes in the heap are either pointed to by \( x \) or reachable from the node pointed to by \( x \). Formula (5.2) states that the any node pointed to by \( y \) has no incoming \( left \) edge.

We will restrict our attention to closed formulas (no free variables). We say that a formula is **guarded** if every quantifier is of the form \( \langle \forall v. x(v) \implies \psi \rangle \) or \( \langle \exists v. x(v) \land \psi \rangle \) where \( x \) is some program variable.

To evaluate formula \( \varphi \) in state \( S \) we translate \( S \) to a standard logical structure \( \hat{S} \) and \( \varphi \) to a FO formula, \( TR(\varphi) \), in the vocabulary of \( \hat{S} \). Let \( [[\varphi]]^S \equiv [[TR(\varphi)]]^{\hat{S}} \) where the right hand side is standard FO Tarskian semantics. Theorem 5.4.2 ensures the soundness of the evaluation and guarantees completeness for the guarded fragment of the query logic.

**Theorem 5.4.2** For every formula \( \varphi \), \( \lambda S. [[TR(\varphi)]]^{\hat{S}} \) is a sound evaluation function. If \( \varphi \) is guarded, it is also a complete evaluation function.

### 5.4.2 Translation

The universe of \( \hat{S} \) is the universe of \( S \). The vocabulary and its interpretation are given in Figure 5.6(a). The translation defines for each edge \( f \) two predicates, \( f^\varphi \) and \( f^\exists \). If \( f^\varphi(n_1, n_2) \) then there is an \( f \) must edge from \( n_1 \) to \( n_2 \). If \( f^\exists(n_1, n_2) \) then there is an \( f \) may edge from \( n_1 \) to \( n_2 \). Similarly we use \( TC^\varphi(n_1, n_2) \) to define a must path from \( n_1 \) to \( n_2 \), and \( TC^\exists(n_1, n_2) \) to define a may path from \( n_1 \) to \( n_2 \). The translation is a formalization of Theorem 5.2.1. Figure 5.6(b) gives the translation of \( S_2 \) defined in Figure 5.3. The translation rules for the literals in the query formula are given in Figure 5.6(c).

**Theorem 5.4.2**

Proof: (sketch)

The evaluation of \( TR(\varphi) \) on \( \hat{S} \) simulates the evaluation of a \( \varphi \) on any concrete state \( C \) s.t. \( (S, C) \in \gamma \). Assume an assignment \( v_i \mapsto n_i \) satisfies a literal \( L(v_1, ..., v_k) \) in \( S' \), we shall see that \( v_i \mapsto M(n_i) \) satisfies \( TR(L)(v_1, ..., v_k) \). Most cases are immediate from the definition of \( \hat{S} \) and the properties of the abstraction (Section 5.2.2). The only case requiring further explanation is \( L \equiv \neg v_1 = v_2 \). Here we may chose \( n_1 \neq n_2 \) s.t. \( M(n_1) = M(n_2) \), but in this case \( sm(M(n_1)) \) thus \( TR(L)(v_1, v_2) \) still evaluates to true. Since an NNF formula has no negation outside of literals this is enough for soundness.

Examples: The translation of (5.1) is \( \forall v. \exists w.x(w) \land (v = w \lor TC^\exists(w, v)) \) which evaluates to true in \( \hat{S}_2 \) as expected. The translation of (5.2) is \( \forall v, w. \neg y(w) \lor \neg left^\exists(v, w) \) unfortunately this formula also evaluates to true. In some cases, including this one, we can overcome this imprecision by an improved formula translation \( TR'(\varphi) \), as described in Appendix 5.4.3.
5.4. Evaluation

<table>
<thead>
<tr>
<th>Vocabulary</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x(n)$</td>
<td>$(x, n) \in env^S$</td>
</tr>
<tr>
<td>$f^3(n_1, n_2)$</td>
<td>$(n_1, n_2) \in ref^S(f)$</td>
</tr>
<tr>
<td>$f^\nu(n_1, n_2)$</td>
<td>$f^3(n_1, n_2) \land \neg sm(n_1) \land \neg sm(n_2)$</td>
</tr>
<tr>
<td>$TC^\exists(n_1, n_2)$</td>
<td>A (possibly empty) directed path from $n_1$ to $n_2$</td>
</tr>
<tr>
<td>$TC^\nu(n_1, n_2)$</td>
<td>$TC^\exists(n_1, n_2), n_1 \neq n_2$ and the path satisfies case 6 of Theorem 5.2.1</td>
</tr>
<tr>
<td>$sm(n)$</td>
<td>$\vee_{f \in \text{Pref}}(n, n) \in ref^S(f)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>(0)</td>
</tr>
<tr>
<td>$y$</td>
<td>(2)</td>
</tr>
<tr>
<td>$left^\exists$</td>
<td>(0, 1), (1, 1), (1, 2)</td>
</tr>
<tr>
<td>$left^\nu$</td>
<td>(0, 1), (1, 2), (1, 2)</td>
</tr>
<tr>
<td>$TC^\exists$</td>
<td>(0, 1), (0, 2), (1, 1), (1, 2)</td>
</tr>
<tr>
<td>$TC^\nu$</td>
<td>(0, 1), (0, 2), (1, 2)</td>
</tr>
<tr>
<td>$sm$</td>
<td>(1)</td>
</tr>
</tbody>
</table>

Figure 5.6: (a) Translation of an abstract state to a logical structure. (b) $\hat{S}_2$ - the translation of $S_2$ from Figure 5.3 (c) Rules for translating a query formula to the vocabulary of $\hat{S}_2$

5.4.3 Improved Translation for Evaluation

To solve the imprecision presented above, we present a more precise formula translation (i.e., the evaluation function for $\varphi$ returns false for more states $S$ in which $\forall C \in \gamma\{S\}. \neg[\llbracket \varphi \rrbracket^C]$). The improved translation is not compatible with the partial join suggested. Thus, it can only be used if the relational join is in play.

We make the following observation: If $(n, m) \in ref^S(f)$ and $n$ is a summary node (respectively $m$) for every $C \in \gamma\{S\}$, $n$ is the contraction of at least two nodes $n_1, n_2$ in $C$, $(n_1, m) \in ref^C(f)$, and $(n_2, m) \not\in ref^C(f)$. Thus, if $v$ and $w$ are universally quantified variables we define $TR'(f(v, w)) \overset{def}{=} f^\nu(v, w)$ and $TR'(-f(v, w)) \overset{def}{=} -f^\exists(v, w)$. The improved translation is more precise in significant cases in practice. For example, the improved translation of formula 5.2 is $\forall v, w. \neg y(w) \lor -left^\exists(v, w)$ which evaluates to false on $\hat{S}_2$ as expected. However, there is still a subtle issue we shall now resolve.

Variable Aliasing

Figure 5.7 demonstrates a problem with the improved translation proposed. Consider the evaluation of formula 5.3

$$\forall v, v_1, v_2. \neg left^\exists(v, v_1) \lor \neg right'(v, v_2)$$  \hspace{1cm} (5.3)

The improved translation will result in the following formula: $\forall v, v_1, v_2. \neg left^\exists(v, v_1) \lor \neg right^\exists(v, v_2)$ The evaluation function of the improved translation for formula 5.3 returns false for $\hat{S}_3$ (consider the assignment $v \mapsto 1, v_1 \mapsto 2, v_2 \mapsto 3$), which is obviously wrong (specifically, the concrete state $C_2 \in \gamma\{S_3\}$ for which the formula evaluates to true). The problem comes from the fact that the improved translation forces us to choose a concrete node in which both literal $\neg left^\exists(v, v_1)$ and $\neg right'(v, v_2)$ are false, which is impossible in $S'_2$. To
Figure 5.7: (a) Abstract state $S_3$, (b) a concrete state $C_2$ s.t. $(S_3, C_2) \in \gamma$ that demonstrate the problem with the improved translation (c) $\hat{S}_3$

solve this problem, whenever we see $\forall v. \varphi(v)$ and $v$ variable appears more than once in $\varphi$, we translate the formula to $\forall v. sm(v) \lor TR'(\varphi(v))$. Thus, if $v$ is a summary node, we know nothing about it, otherwise, the improved translation is used.

Note that this rule needs to be applied only when the improved translation is used, variables that do not occur in literals that have gone through the improved translation need not be touched. There are cases in which occurrences of a variable $v$ do not count for the application of this rule. These are:

- Occurrence of $v$ in equality (since equality already deals with the aliasing problem).
- Occurrence of $v$ in a unary predicate representing a program variable (since either all the concrete nodes represented by an abstract node are pointed to by a variable, or none do)

5.5 Extensions

We have developed several extensions of the technique described here.

5.5.1 Loop Boundaries

Some programs temporarily violate the data structure invariants (including admissibility) and restore all within the boundary of a single loop iteration. We can handle such programs with the same level of precision by only performing $\beta$ on loop boundaries.

5.5.2 Partial Join

Partial Join [MSRF04] replaces union as the join operator of the abstract domain with an operator that merges matching states. We build a variant of the partial join operator by ignoring the self-loops when giving canonical names to states. Matching states are merged by performing union on the self-loops on nodes with the same canonical names. The concretization function is modified to consider that some of the self-loops may not represent concrete edges.
The focus operation needs to be updated according to the changes in the concretization function. There are two changes in the algorithm: 1) There is no need to enumerate the self-loops in the subgraph contracted to the summary node. 2) The case in which the summary node represents a single node needs to be considered.

The experimental results (Section 5.6) show that Partial Join is important for performance, while maintaining precision.

5.5.3 Cycles

The abstract domain can be extended to support cycles in the following limited way. A directed cycle is admissible if there is a path from a variable that contains the entire cycle and all the outgoing edges from all the nodes of this path are of the same edge type (i.e., the cycle is a part of a singly-linked list). A state is admissible if all its undirected cycles are actually admissible directed cycles. All the properties of the abstraction such as the bounded abstract state size remain true for this extended class. Focus and $\beta$ can be easily modified to support these cycles since an entire cycle can never be contracted (since there has to be a node on each cycle that is either pointed to by a variable or heap-shared). The subtleties come from two sources. One is the fact that a self-loop can now represent a concrete self-loop and not a summary node. This can be easily solved by adding an extra bit per node indicating whether it is a summary node or not and maintaining it in all the operations.

The second subtlety is in computation of canonical names, since without breaking the cycles we may never be able to give a name to a node before traversing its children. The solution is to mark the back-edges during the first DFS and ignore them in the second DFS. At the end, we add their names to their incoming nodes.

5.5.4 Parent Pointers

The abstract domain can be extended to allow parent pointers (i.e., doubly linked lists and trees with parent pointers) in the following limited way. Each node can use only a single field as a parent pointer (specified by the user). Parent pointers are not considered for contraction, heap-sharing or garbage (thus every node has to be reachable using non parent-pointer fields). This means that exactly the same nodes will be contracted whether parent pointers exist or not. Either all the nodes contracted to a summary have the same parent pointer (in this case we say that the summary node has that parent pointer) or none of them have it. If two nodes are contracted, all the parent pointers incoming or outgoing from these nodes have to be the inverse of “real” reference fields and the two nodes and the edge between them have to agree on the parent pointer (either none have parent pointers, or all of them have the same parent pointer).

These limitations still allow us to handle doubly-linked lists and trees with parent pointers as long as every node is reachable using “real” reference fields (i.e. there is a pointer from the head of the doubly linked list or from the root of the tree). Specifically we can handle all the doubly-linked list examples of [SRW02].

To support this extension we make the following changes:

Focus The only problem in the current focus is the fact that we can now traverse a parent pointer into a summary node and, in this case, it does not necessarily lead to the root of the sub-graph contracted to the summary node. The parent pointers within the sub-graph are easy
to handle since they are either the inverse of all the reference fields in the sub-graph or none of them.

**Beta** Since the contractions ignore the parent pointers we only need to make sure that the state is admissible. We update the current admissibility check to consider the parent pointer limitation described above.

The algorithm for updating canonical names is simple.

### 5.6 Implementation

We have implemented the abstract transformer detailed above including the extensions of Section 5.5. Focus was implemented only for linked lists and binary trees (i.e., up to two self-loops). The implementation is written in Java and is integrated with the Soot Java Optimization Framework [VRHS+99] as a front end. The empirical results of running our analysis on some examples are given in Figure 5.8. In all cases the analysis also proved absence of memory leaks, acyclicity (where applicable) and absence of null-dereferences. N/A states that the information for the example is not available for that tool and O/S means that it is out of scope for the tool. Max states is the maximum number of states in each program point. The columns marked with "[R]" use the relational join as described in Section 5.3. The columns marked with "[P]" use the partial join extension described in Section 5.5. The TVLA times given for tree manipulating algorithms use partial join as well. The tests were made on an Intel Pentium M, 1.6 GHz with 1.00 GB of RAM.

The “insertSortedTree” and “deleteSortedTree” programs insert and delete element in a sorted tree respectively. The “LDS” program performs the Lindstrom scan [Lin73] of a binary tree. The “reverse” program reverses an acyclic list and “reverseCycle” reverses a cyclic list. The “merge” program merges two sorted lists. The “delete” program deletes an element from a sorted list. The “insertSort” programs does in-place sort of linked lists. The “bubbleSort” and “bubbleSort2” are two variants of an in-place bubble sort for linked lists analyzed by TVLA and [BR05] respectively. We have used TVLA with relational join [MSRF04]. In the tree examples, it is also interesting to compare to the partial join version of TVLA. We shall say more on that in our concluding remarks.

We can see that our analysis is indeed fast and in some cases up to 100 times faster than the other analyses depicted. We should point out that most examples are small, thus the differences in running times can be partially attributed to engineering issues. Checking the properties detailed above for these examples is done automatically by the system. To check other properties we need a way to extract information from the abstract states. This is done by formula evaluation and is detailed in Section 5.4.

One of the main issues with the domain presented in this chapter is that it is ad-hoc, i.e., extending it in any way requires adjusting all the algorithms and reproving soundness and completeness. In the next chapter we present a methodology that allows developing abstract domains for specialized shape analysis problems in a more disciplined and modular way.
Figure 5.8: The empirical results from running the abstract transformer implementation

<table>
<thead>
<tr>
<th>Programs</th>
<th>Time[R]</th>
<th>Max states[R]</th>
<th>Time[P]</th>
<th>Max states[P]</th>
<th>[BR05]</th>
<th>TVLA(^\d)</th>
<th>[MYRS05]</th>
</tr>
</thead>
<tbody>
<tr>
<td>deleteSortedTree</td>
<td>2.359.70</td>
<td>192,355</td>
<td>3.22</td>
<td>520</td>
<td>O/S</td>
<td>47.48</td>
<td>O/S</td>
</tr>
<tr>
<td>insertSortedTree</td>
<td>20.85</td>
<td>9,365</td>
<td>0.55</td>
<td>264</td>
<td>O/S</td>
<td>1.8</td>
<td>O/S</td>
</tr>
<tr>
<td>lindstromScan</td>
<td>1,459.63</td>
<td>79,673</td>
<td>8.36</td>
<td>1337</td>
<td>O/S</td>
<td>65.86</td>
<td>O/S</td>
</tr>
<tr>
<td>insertRedBlack</td>
<td>&gt; 24 hours</td>
<td>38.15</td>
<td>4,853</td>
<td>O/S</td>
<td>N/A</td>
<td>O/S</td>
<td></td>
</tr>
<tr>
<td>reverse</td>
<td>0.05</td>
<td>15</td>
<td>0.11</td>
<td>8</td>
<td>0.1</td>
<td>0.531</td>
<td>5</td>
</tr>
<tr>
<td>reverseCycle</td>
<td>0.24</td>
<td>159</td>
<td>0.26</td>
<td>62</td>
<td>0.1</td>
<td>N/A</td>
<td>2</td>
</tr>
<tr>
<td>merge</td>
<td>0.20</td>
<td>96</td>
<td>0.15</td>
<td>36</td>
<td>17.8</td>
<td>4.006</td>
<td>15</td>
</tr>
<tr>
<td>delete</td>
<td>0.02</td>
<td>20</td>
<td>0.01</td>
<td>12</td>
<td>0.9</td>
<td>1.242</td>
<td>7</td>
</tr>
<tr>
<td>bubbleSort</td>
<td>0.03</td>
<td>36</td>
<td>0.03</td>
<td>21</td>
<td>N/A</td>
<td>11.887</td>
<td>N/A</td>
</tr>
<tr>
<td>bubbleSort2</td>
<td>0.08</td>
<td>76</td>
<td>0.08</td>
<td>33</td>
<td>11.4</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>insertSort</td>
<td>0.06</td>
<td>100</td>
<td>0.05</td>
<td>48</td>
<td>N/A</td>
<td>20.219</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Chapter 6

Constructing Specialized Shape Analyses for Uniform Change

This chapter is concerned with one of the basic problems in abstract interpretation, namely, for a given abstraction and a given set of concrete transformers (that express the concrete semantics of a program), how does one create the associated abstract transformers? We develop a new methodology for addressing this problem, based on a syntactically restricted language for expressing concrete transformers. We use this methodology to produce most precise abstract transformers for abstractions of many important data structures.

Outline

• We start with an informal overview of the problem in Section 6.1,
• In Section 6.2 we give preliminary definitions,
• In Section 6.3 we develop a methodology for developing computable transformers,
• In Section 6.4 we explore various applications of the methodology for specialized shape analysis problems,

6.1 Overview

This section is an informal overview of the methodology presented in this chapter. We use a simple Java procedure that reverses a singly-linked list specified in Figure 3.1 as a running example. We will run reverse on a cyclic singly-linked list. We use a graphical representation of logical structures to depict a store as a graph.

Figure 6.1(a) is an example of a singly linked list with a cycle. Memory cells are represented by the individuals of the structures (the nodes in the graph). Program variables are represented by constants (the text inside the nodes). Pointer fields in a memory cell are represented by binary relations (the edges of the graph, annotated with the relation name). In this case, the next field of the list nodes is represented by the n relation, which is a total function. We can add to the structure auxiliary relations defined using FO(TC) formulas over
Figure 6.1: (a) A concrete structure that represents a singly-linked list with a loop, which is pointed to by $x$ and consists of 6 nodes. (b) The same singly-linked list, this time with auxiliary information. (c) Abstraction of singly-linked lists with loops. (d) & (e) The result of computing formulas (using Kleene semantics) on the abstract structure results in a definite value (i.e., $1$ or $0$). Note there is always a concrete node, null, with a self-loop (for $n$) and no other edges. We do not draw this to save space.

the core relations. For example, in Figure 6.1(b) we use a unary relation $r_{x,n}$ (written below the nodes) to indicate the existence of a path from the node pointed to by $x$ (defined by $r_{x,n}(v) \equiv n^*(x, v)$). The unary relation $c_n$ states that the node is on a cycle of $n\text{ext}$ fields (defined by $c_n(v) \overset{\text{def}}{=} n^+(v, v)$).

We briefly repeat the concepts of three-valued logical shape analysis used in this chapter. In abstract interpretation, we wish to represent a large (possibly infinite) set of stores using a finite set of structures; here this is done by collapsing nodes together into “summary nodes” (drawn as double circles). We use three-valued logic with an additional $\frac{1}{2}$ truth value (for binary relations, this is depicted as a dotted edge) to capture the case in which for some of the nodes represented by the summary node the value is true ($1$) while for others the value is false ($0$). Figure 6.1(c) shows an abstract structure in which constants are untouched and all the nodes with the same values for unary relations are collapsed together. This type of abstraction is called Canonical Abstraction and is guaranteed to result in structures of bounded size for a given vocabulary. The Embedding Theorem of [SRW02] guarantees that if evaluating formulas (using Kleene semantics) on the abstract structure results in a definite value (i.e., $1$ or $0$), evaluating the formula on any concrete structure it represents will yield the same value. Kleene semantics can be understood as considering $\frac{1}{2}$ to be $\{0, 1\}$, $0$ to be $\{0\}$, and $1$ to be $\{1\}$ and evaluating pointwise, e.g., $1 \land \frac{1}{2} = \frac{1}{2}$, but $0 \land \frac{1}{2} = 0$.

Transformers are given for each operation according to the program’s operational semantics. Transformers are specified using guarded commands with formulas in FO(TC) called update formulas. For example, for the operation $t=x.n\text{ext}$ used in line 2 of Figure 3.1, we can use a guard $x \neq \text{null} \land n(x, x_n)$ to (a) ensure that there is no null-dereference, and (b) bind the value of the $n\text{ext}$ field of $x$ to a new (temporary) constant $x_n$. The update for-

---

1For readers familiar with [SRW02], we use tight embedding in this chapter. Thus, each summary node represents at least two nodes.
mulas are: \( t' := x_n, x' := x, n'(v_1, v_2) := n(v_1, v_2), c'_n(v) = c_n(v), r'_{x,n}(v) := r_{x,n}(v), r'_{t,n}(v) := n^*(t', v) \). The most precise abstract transformer would return a set of abstract structures that captures as tightly as possible (for the abstraction in use) the result of applying the transformer on all the concrete structures represented by the original abstract structure. This kind of abstract transformer is called the most precise abstract transformer [CC79] and can be theoretically computed by finding all concrete structures represented by an abstract structure (a.k.a. concretization), computing the transformer on each of them, and abstracting the results. However, because the number of concrete structures represented by an abstract structure is unbounded (and potentially infinite), this is not an algorithm. Figure 6.1(d) and Figure 6.1(e) show the result for \( t=x\ .\ next \) on the structure in Figure 6.1(c). The structure in Figure 6.1(d) represents the case in which the list before the cycle is of length 2, and the structure in Figure 6.1(e) represents the case where it is of length 3 or more. Note that simply evaluating the update formulas on the structure in Figure 6.1(c) would not have given us this precise result.

We seek a way to compute the same result as the most precise abstract transformer without resorting to full concretization. One of the key principles of our methodology is to find a partial concretization that 1) is computable, 2) returns a finite set of abstract structures that represents the same concrete structures, and 3) for each of these structures the most precise abstract transformer can be computed by simply evaluating the update formulas. We call the operation of finding such a partial concretization Focus after a similar operation in [SRW02]. Focus replaces each structure with a set of structures, representing the same concrete structures, in which the partitioning of the concrete nodes into summary nodes is more fine-grained. This can be achieved by bifurcating summary nodes into two groups: nodes for which an atomic formula holds, and nodes for which it does not hold. We call such a formula a focus formula. For example, Figure 6.2(a) and (b) show the result of Focus for the focus formula \( n(x, v) \) on the structure in Figure 6.1(c). The second and third nodes in the lists of Figure 6.2(a) and (b) are the result of bifurcating the second node in Figure 6.1(c) according to the focus formula. For the second node, the formula holds, and for the third node the formula does not hold. As we can see, this process can result in multiple structures; Figure 6.2(a) corresponds to the case in which the original summary node represents two concrete nodes and in Figure 6.2(b) the case in which the summary node represents three or more concrete nodes. We can see that in both cases, the second node has been materialized out of the original summary node.

To automate the Focus operation, we propose an algorithm that can compute the partial concretization for a set of focus formulas: the first phase does not understand the intended meaning of the relations; the second phase applies a “feasibility check” supplied by the developer of the abstraction. An algorithm for feasibility checking should return true iff an abstract structure represents at least one concrete structure. Figure 6.2(c) and (d) show structures arising in the Focus process that are infeasible. Structure 6.2(c) is infeasible because the second node must represent at least two nodes and the first node must have a direct edge to both of them, which contradicts that \( n \) is a function. Structure 6.2(d) is infeasible because the self-loop on the second node means that it must both have a self-loop and not have a self-loop. In Section 6.4, we provide algorithms for checking feasibility for several abstractions of commonly used data structures. Note that even if we cannot check feasibility for some abstraction (or have only a sound approximation), the resulting transformer is a sound approximation of the most precise abstract transformer.

The problem with finding the right focus formulas and using Focus for the transformer given for \( t=x\ .\ next \) is that for the computation of \( r'_{t,n} \) we require that the evaluation of
Figure 6.2: Some of the structures arising in the process of Focus for the operation \( t = x . \text{next} \) on the structure in Figure 6.1(c).

\( n^*(t', v) \) return precise results — in particular; for any element in the cycle, it should return 1. However, this means that all the edges until the cycle must be 1, which means we need to consider all possible lengths for the segment of the list before the cycle. This is not possible. To solve this problem, we need to somehow limit the update formulas. This leads to our second principle, \textbf{monadic-uniform update formulas}.

The update formula for \( r_{t,n}' \) can be rewritten as \( r_{t,n}'(v) := r_{x,n}(v) \land (c_n(x) \lor x \neq v) \). If \( x \) is on a cycle, \( t \) must be on the same cycle; thus, whatever was reachable from \( x \) is now also reachable from \( t \). Otherwise, the only node that was reachable from \( x \) and is not reachable from \( t \) is \( x \) itself. Evaluating this updated transformer on the structures in Figure 6.2(a) and (b) results in the structures in Figure 6.1(d) and Figure 6.1(e). Thus, focusing on \( n(x, v) \) was enough. This is not a coincidence. We show that if we limit the update formulas to a certain syntactic class (which we call monadic-uniform), we can automatically find the focus formulas needed for the Focus operation, and the result of Focus is guaranteed to be bounded (a function of the size of the original structure).

The process of finding monadic-uniform update formulas is not trivial, especially when trying to update reachability. Fortunately, we can use existing results from the dynamic descriptive complexity [Imm99, Hes03] and database [DS95] communities on maintaining reachability when edges are added or removed. A key step in finding such monadic-uniform update formulas is the addition of auxiliary relations, which together with the other relations can be maintained by monadic-uniform update formulas. In Section 6.4, we provide monadic-uniform transformers for the abstractions used for many of the analyses done successfully with TVLA.

Our methodology can be summarized as follows:

1. Find an abstraction that captures the properties you want to verify. Describe it within the framework of parameterized shape analysis of [SRW02].

2. Insure that all update formulas are monadic-uniform, adding extra auxiliary relations as needed.

3. Optionally, develop a feasibility check for abstract structures of this (possibly augmented) vocabulary; or, settle for a sound approximation of the most precise abstract transformer.

The chapter presents the necessary algorithms for binding these ingredients together to compute most precise abstract transformers.
6.2 Preliminaries

We represent stores as logical structures. This allows us to use logical formulas to define the semantics of statements and abstractions of stores. To simplify the presentation, we describe everything in the context of a specific vocabulary. It should be clear from the description that the formulas are schematic and can be instantiated to the specific program fields and variables.

See Definition C.1.1 for a formal definition of the syntax of FO(TC) formulas. We use the shorthand (when \( \varphi_1 \implies \psi_1, \ldots, \varphi_k \implies \psi_k, \text{default} \implies \psi )\) for a sequential case split; i.e., formally it is: \( \ldots \lor (\neg \varphi_1 \land \ldots \lor \neg \varphi_{k-1} \land \varphi_k \land \psi_1) \lor \ldots \lor (\neg \varphi_1 \land \ldots \lor \neg \varphi_k \land \psi) \).

A 2-valued logical structure is a triple \( S = \langle U^S, R^S, C^S \rangle \) of a universe \( U^S \) of individuals, a map \( R^S \) of relation symbols to truth-valued functions, and a map \( C^S \) of constant symbols to individuals. See Definition C.1.2 [LASIR06] for a formal definition.

6.2.1 Programming-Language Statements

Formulas are used to update the store in a standard way as follows:

**Definition 6.2.1 (Store Updates)** An update formula of a relation \( r \) of arity \( k \) has the form: \( r'(v_1, \ldots, v_k) := \varphi_r(v_1, \ldots, v_k) \), where \( \varphi_r(v_1, \ldots, v_k) \) is a formula with free variables \( v_1, v_2, \ldots, v_k \). An update formula of a constant \( c \) has the form:

\[
\begin{align*}
\varphi_c(v) & := (\text{when } \varphi_1 \implies s_1, \ldots, \text{when } \varphi_k \implies s_k, \text{default} \implies s_{k+1}), \\
\end{align*}
\]

where the \( \varphi_i \) are closed formulas and the \( s_i \) are constant symbols. This is a shorthand for the following formula with one free variable: \( \varphi_c(v) \overset{\text{def}}{=} (\ldots, \text{when } \varphi_i \implies v = s_i, \ldots, \text{default} \implies v = s_{k+1}) \) For the special case in which \( k=0 \) we simply write \( \varphi' := s_1 \).

Every statement \( s_t \) in the programming language is associated with a transformer \( \tau_{s_t} \), which consists of a guard formula, \( \text{guard}_{\tau_{s_t}} \), and a set of update formulas for each relation and constant symbol in the vocabulary. If the guard formula has free variables, the update formulas can refer to them as constants.

Given a 2-valued logical structure, \( S = \langle U, R, C \rangle \), the expansion of \( S \) for \( \tau \) is the set \( \text{expand}_{\tau}(S) \) of all the structures \( S' = \langle U, R, C' \rangle \) s.t., \( C' \) is identical to \( C \) except it gives an interpretation to all the free variables of \( \text{guard}_{\tau} \). We say \( S' \) is expanded for \( \tau \).

The application of the transformer \( \tau \) on a structure \( S' \in \text{expand}\_\tau(S) \) is the 2-valued structure \( \tau(S') \overset{\text{def}}{=} \langle U, R'', C'' \rangle \), where for every relation symbol \( r \), \( R''(r)(\overline{u}) = [\varphi_r(\overline{u})]^S' \), and for every constant symbol \( c \), let \( u_c \in U \) be the unique element for which \( S', u_c \models \varphi_c \), we have \( C''(c) = u_c \). Note that \( C'' \) gives an interpretation only to the original constants and not to the free variables of \( \text{guard}_{\tau} \). The meaning of the transformer \( \tau \) on \( S \) is the set \( [\tau](S) \overset{\text{def}}{=} \{ \tau(S') \mid S' \in \text{expand}(S) \land S' \models \text{guard}_{\tau} \} \).

The free variables in the guard formula allow for the introduction of non-determinism. These free variables are considered as additional constants by the update formulas. The syntactic form of the update formulas for constants guarantees that for each constant symbol \( c \) there is only one \( u_c \) for which \( S', u_c \models \varphi_c \). Thus, once the free variables have been assigned, the computation of the transformer is deterministic.

For simplicity, we do not support operations that change the universe. However, because we allow infinite universes, we can easily model the allocation and deallocation of individuals using a designated relation that holds only for allocated individuals (or, if the operational semantics allows, by using a free list).
Table 6.1 lists the transformers that define the operational semantics of the five kinds of Java-like statements. Here \( x, t, \) and \( y \) are constants that denote the target of pointer variables \( x, t, \) and \( y, \) respectively. \( sel \) is a binary relation that models the pointer field \( sel. \) We do not specify update-formulas for relations and constants with unchanged values. The guard formulas for statements that access \( sel \) ensure that no null-dereference has occurred. In case of a field traversal, the guard formula also selects the target of the field using the free variable \( t_{sel}. \) Note that program conditions are simply modeled by guard formulas.

**Integrity Constraints:** We allow restriction of the potential stores that may arise in the program by a finite set of closed formulas called integrity constraints and denoted by \( \Sigma. \) We assume that the meaning of every transformer \( \tau \) maintains the integrity constraints, i.e., if \( S \models \Sigma, S' \in [\tau]^S \) a 2-valued structure, then \( S' \models \Sigma. \)

In the case of pointer fields, we require that every field be a total function. Thus, in particular, the pointer field(s) of \( null \) points to \( null. \)

**Auxiliary Information:** The most interesting integrity constraints occur as a result of extra relations whose values are derived from other relations. Formally, an auxiliary relation \( r \) of arity \( k \) is defined via a defining formula \( \varphi_r \) with \( k \) free variables. This results in the integrity constraint \( \forall v_1, \ldots, v_k : r(v_1, \ldots, v_k) \iff \varphi_r. \) Thus, every statement must maintain this invariant. Auxiliary information allows us to reduce the complexity of update formulas. Furthermore, it is often the information maintained by auxiliary relations that enables us to compute most precise abstract transformers.

Section 6.1 introduced two types of auxiliary relations, \( r_{x,n} \) for reachability from a program variable, and \( c_n \) for cyclicity. The interaction between them is used to define a monadic-uniform update formula for traversal of an edge.

### 6.2.2 Monadic-Uniform Updates

In this section, we restrict the way the semantics of statements are allowed to be defined to use only formulas of a certain syntactic class. The new stores can differ from the original store in many values but the change should be uniform in the sense defined below. We begin by defining atomic formulas that are essentially unary.

**Definition 6.2.2** We say that an atomic formula is monadic if it is of the form \( r(c_1, \ldots, c_i, v, c_{i+1}, \ldots, c_{k-1}) \) where \( r \) is \( k \)-ary relation and \( c_1, \ldots, c_{k-1} \) are constant symbols.
An FO(TC) formula \( \varphi \) is **monadic** if all of the atomic formulas appearing in \( \varphi \) are monadic or ground.

The following formulas are monadic: \( r(v, c), v = c, r(v) \), \( \forall v. r(v, c) \). The following formulas have variables in more than one position, and thus are not monadic: \( r(v, v), r(v_1, v_2), v_1 = v_2 \). Note that although \( r(v, v) \) uses a single variable, it is not monadic.

Next, we define monadic update formulas, which are a restricted case of update formulas in which a tuple is classified by monadic formulas, and for each class, the value of an existing relation is copied.

**Definition 6.2.3 (Monadic-Uniform Updates)** A **monadic-uniform formula** \( \varphi(v_1, \ldots, v_k) \) is syntactically equivalent to \( (\ldots, \text{when } \varphi_1 \implies \psi_1, \ldots, \text{default } \implies \psi_i) \) where the \( \varphi_i \) are monadic FO(TC) formulas with free variables \( v_1, v_2, \ldots, v_k \), and the \( \psi_i \) are restricted to either 1, 0, or a literal with distinct variables.

A **monadic-uniform transformer** is a transformer in which all the update formulas and the guard formula are monadic uniform.

All the transformers of Table 6.1 are constructed to be monadic-uniform transformers (see Section 6.4). Monadic-uniform formulas disallow direct interaction between non-monadic relations, e.g., \( r(v_1, v_2) \land q(v_1, v_2) \) is not monadic-uniform. \( r(v, v) \) is not monadic-uniform because it is equivalent to \( r(v_1, v_2) \land v_1 = v_2 \) and captures the interaction between \( r \) and equality.

### 6.2.3 Canonical Abstraction

In this section, we use 3-valued logic to conservatively represent sets of stores. As explained in Chapter 2, we define a lattice of static information where lattice elements are sets of 3-valued structures. A 3-valued structure is similar to a 2-valued structure, except \( S^3 \) has the same value in all of the embedded concrete structures.

**Definition 6.2.4** A **tight embedding function** is a surjective function \( f: U^S \rightarrow U^{S'} \) such that, for every \( c \in C \), \( C^{S'}(c) = f(C^S(c)) \) and for every relation \( r \in R \) of arity \( k \), \( R^{S'}(r)(u'_{1}, \ldots, u'_{k}) = \bigcup_{f(u_i)=u'_i, 1 \leq i \leq k} R^S(r)(u_1, \ldots, u_k) \). We say that \( S' = f(S) \) and that \( S' \) is a **tight embedding** of \( S \).

Note that if \( C^S(c) = u \) and \( u \) is a summary node, only one of the nodes mapped to \( u \) equals \( c \), not all of them. Thus, \( [u = u]^{S'} = \frac{1}{2} \).

**Canonical embedding**, denoted by \( \beta \), is the embedding obtained by using unary relation symbols to distinguish between individuals, i.e., two concrete individuals \( u_1, u_2 \in U^S \) are mapped to the same individual if and only if they agree on the values of unary relation symbols. For each constant, \( c \), there is an implied unary relation, \( P_c \), true just of \( c \).

According to the **embedding theorem** [SRW02], every formula with a definite value in a structure has the same value in all of the embedded concrete structures.

Canonical abstraction allows us to define the set of stores represented by a 3-valued structure.

---

2In the rest of the chapter, whenever we refer to embedding, we mean tight embedding and use the term tight embedding only for emphasis.
Definition 6.2.5 For a 3-valued structure $S$, $\gamma(S)$ denotes the set of 2-valued structures that $S$ represents, i.e., $\gamma(S) = \{S^2 | \Sigma \models \beta(S^2) = S\}$. We say that a structure $S$ is feasible if $\gamma(S) \neq \emptyset$.

The complexity of checking feasibility of a structure comes from the need to satisfy the integrity constraints and because of interactions between auxiliary relations and core relations.

### 6.3 Methodology for Developing Computable Transformers

A shape-analysis problem is characterized by a triple of the class of allowed structures, the initial abstraction, and the set of possible atomic operations.

The running example (see Figure 3.1) is an instance of the following shape-analysis problem: The class of allowed structures is (possibly cyclic) singly-linked lists. The initial abstraction tracks: pointed to by a program variable (by representing program variables as logical constants), the next field (by maintaining a binary relation $n$), reachability from program variables (by unary relations of the form $r_{x,n}(v)$, which indicate that $v$ is reachable from program variable $x$ using the next field), and cyclicity (by a unary relation $c_n(v)$, which indicates that $v$ is part of a cycle).

The first step in developing computable most precise abstract transformers for a shape-analysis problem is to find monadic-uniform transformers for all the operations required. A key step in finding such update formulas is the introduction of additional auxiliary relations that, together with the original relations, can be maintained in a monadic-uniform way.

The main difficulty in maintaining the relations used in the shape-analysis problem for the running example is the maintenance of reachability. Fortunately, we can use (with a small modification to make it monadic-uniform) the DynQF update formulas for transitive closure given by Hesse in [Hes03]. We introduce three auxiliary binary relations. The relation $p_n(v_1, v_2)$ maintains the reflexive transitive closure of the $n$ relation (i.e., existence of a path between $v_1$ and $v_2$ using the next field). The relation $cut_n(v_1, v_2)$ holds for exactly one edge in each cycle (enforced using appropriate integrity constraints). The relation $pc_n(v_1, v_2)$ (called PathCut by Hesse) maintains the reflexive transitive closure of the un-cut edges. Together, these relations allow us to create monadic-uniform transformers for all the needed operations (see [Hes03] and Section 6.4 for more details).

Imperative programs lead to monadic-uniform transformers because they can only change information directly pointed to by variables. The difficulty comes from relations such as reachability in which a local update can cause widespread change. We take advantage of the specific structure of the graphs in each case to build a monadic-uniform transformer for them.

The final step in our methodology is to develop an algorithm for checking the feasibility of an abstract structure of the chosen vocabulary. Here we need to take into account the integrity constraints, including the set of allowed structures and the meaning for all the auxiliary relations.

In Section 6.4, we show that, to check feasibility of an abstract structure that can arise in the shape-analysis problem defined above, we can compute a candidate concrete structure s.t. the abstract structure is feasible iff the concrete structure is consistent (i.e., satisfies the integrity constraints) and its $\beta$ is the original structure. The size of the candidate structure is linear in the size of the original abstract structure. Thus, we can check its feasibility in time polynomial in the size of the original abstract structure.
The rest of the section describes how to compute most precise abstract transformers for a given shape-analysis problem that has monadic-uniform transformers and a decidable feasibility-checking problem. Proofs can be found in Appendix C.1.

First, we define the concept of a **focused** structure for a monadic-uniform transformer. For such structures and transformers, the transformer preserves embedding (see Lemma 6.3.2).

**Definition 6.3.1** We say that \( S \) is focused for a \( \tau \) (denoted by focused\(_{\tau}(S)\)) when (1) \( S \) is expanded for \( \tau \), (2) all the monadic atomic formulas that appear in any update formula of \( \tau \) or in guard\(_{\tau}\), evaluate to definite truth values in \( S \), and (3) all the constants interpreted by \( C^S \) are mapped to concrete nodes.

We define \( \beta_{\tau} \) to be a canonical embedding function that honors all new constants and monadic atomic formulas appearing in transformer \( \tau \). \( \gamma_{\tau} \) is defined analogously to \( \gamma \) but in relation to \( \beta_{\tau} \).

The structures in Figure 6.2(a) and (b) are focused for \( \tau = x\.next \) if we map \( x_n \) to any concrete node (only when \( x_n \) is mapped to the second node of the list will the guard formula hold). For Figure 6.1(c), when trying to interpret \( x_n \) in a way that will satisfy the guard formula, the only node worth considering is the second node of the list. There are two reasons why such a structure is not focused. First, the second node is a summary node, thus a constant cannot be mapped to it. Second, \( n(x,x_n) \), which appears in the guard formula, evaluates to \( \frac{1}{2} \). Note that the fact that the structures in Figure 6.2(a) and (b) are focused does not mean that all the update formulas evaluate to definite values for all the nodes, e.g., the \( n \) relation has several indefinite tuples in resulting structure Figure 6.1(e).

For structures that are focused for a transformer \( \tau \), we use the canonical embedding function \( \beta_{\tau} \), and when referring to the feasibility of a focused structure, we mean non-emptiness of \( \gamma_{\tau} \).

**Lemma 6.3.2** Let \( \tau \) be a monadic-uniform transformer, \( S \) be a structure s.t. focused\(_{\tau}(S) \) holds, \( C \) be a concrete structure, and \( f \) be an embedding function s.t. \( f(C) = S \). The following properties hold: (1) \( f(\tau(C)) = \tau(S) \), (2) \( [\text{guard}_{\tau}]^{C} = [\text{guard}_{\tau}]^{S} \), (3) for every unary relation \( r \) and node \( u \) we have \( [r(u)]^{\tau(C)} = [r(f(u))]^{\tau(S)} \), and (4) for every constant \( c \), \( \tau(S) \) maps \( c \) to a concrete node.

When embedding is preserved, all unary relations are definite, and all the constants are mapped to non-summary nodes, \( \beta \) will return the same value for both updated structures. Corollary 6.3.3 entails that a monadic-uniform transformer is actually the most precise abstract transformer for focused abstract structures.

**Corollary 6.3.3** Let \( \tau \) be a monadic-uniform transformer. If focused\(_{\tau}(S) \) and \( f(C) = S \) then \( \beta(\tau(C)) = \beta(\tau(S)) \)

Corollary 6.3.3 suggests a way to compute the most precise abstract transformer: Given an abstract structure, find a set of feasible focused structures that represent the same concrete structures. Definition 6.3.4 makes this notion formal.
Definition 6.3.4 \( \text{focus}_\tau \) is an operation that given a feasible structure \( S \) returns a finite set of structures \( FS \) s.t. \( \bigcup_{S' \in \gamma(S)} \text{expand}_\tau(S') = \bigcup_{F \in FS} \gamma_\tau(F) \) and for every \( F \in FS \), \( F \) is feasible and focused \( \text{focus}_\tau(F) \).

We now sketch the algorithm that computes \( \text{focus}_\tau \). The algorithm systematically replaces each \( \frac{1}{2} \) value for monadic formulas by 0 or 1, duplicating structures as necessary. There may be a large but bounded number of such structures. Each candidate structure is checked for feasibility and discarded if infeasible.

Algorithm 6.3.5 Given \( \tau, S \), compute \( \text{focus}_\tau(S) \).

0. \( FS = FS_{\text{orig}} = \text{expand}_\tau(S) \) \hspace{1cm} // the current set of structures
1. for each \( A(v) \) from \( MA \) and \( F \) from \( FS \) do 
   2. for each node \( b \in U^F \) s.t. \( [A(b)]^F = \frac{1}{2} \) do 
      3. Remove \( F \) from \( FS \) and replace by \( F_{u_1u_2} : u_j \in \{s,c\} \)
         s.t. \( b \) is split into \( b_0, b_1 \), \( [A(b_i)]^{F_{u_1u_2}} = i \), and,
         \( b_i \) is a summary node in \( F_{u_1u_2} \) iff \( u_j = s \).
   4. for each structure \( F \), new tuple created, \( \bar{t} \), and relation \( R \) s.t. \( [R(\bar{t})]^F = \frac{1}{2} \),
      add structures \( F_i : i \in \{0, 1\} \) s.t. \( [R(\bar{t})]^{F_i} = i \) and \( \beta(F_i) \in FS_{\text{orig}} \)
5. for each structure \( F \), if \( \gamma_\tau(F) = \emptyset \), remove \( F \) from \( FS \)
6. return(\( FS \))

Focus can yield a double-exponential number of structures. The maximum number of individuals in a single structure can be exponential in the number of predicates and the number of possible structures is exponential in the number of nodes. From our experience with TVLA, the first blowup — the maximal number of individuals — rarely happens in practice. However, in contrast to TVLA, the use of tight embedding suggests that the second blowup may indeed occur in practice. We are working on ways to remedy the situation, e.g., by moving to non-tight embedding (see [SRW02]).

From the correctness of Algorithm 6.3.5, our main theorem follows:

Theorem 6.3.6 If \( S \) is feasible then we can automatically compute the most precise abstract transformer: \( bt_\tau(S) \equiv \{ \beta(\tau(S')) \mid S' \in \text{focus}_\tau(S) \land [\text{guard}_\tau]^S = 1 \} \)

Note that if there is no feasibility check, the methodology still guarantees that we obtain a most precise abstract transformer, but with respect to a \( \gamma \) that does not force the concrete structures to adhere to the integrity constraints. However, when using this \( \gamma \), the abstraction is not likely to be strong enough to establish the properties that we desire.

### 6.4 Applications

This section describes several applications of the methodology described in Section 6.3 for computing transformers for different shape-analysis problems. For each problem, we specify
the class of allowed structures, the relations we maintain, and, when known, an algorithm for checking feasibility. Further details can be found in Appendix C.2.

Table 6.2 summarizes the different shape-analysis problems described in this section and the type of feasibility checks we have for them. For all of these problems, we show monadic-uniform transformers for field manipulations. SLL/DLL stands for Singly/Doubly Linked Lists, and NUC for No Undirected Cycles. PVar stands for Program Variables. A description of each class of structures and the meaning of each relation is given in the appropriate subsection below. Note that for every vocabulary we require a new feasibility-checking algorithm.

Dong and Su [DS95] show how to update reachability in a general acyclic graph using first-order logic. However, their formulas are not monadic-uniform and it is unclear whether it is possible to make them monadic-uniform.

Direct means there is a direct algorithm to check feasibility of an abstract structure. MSO means we can reduce the feasibility check to a satisfiability check of an MSO formula on trees. Open means we are still working on checking feasibility for this problem. We believe that checking feasibility is decidable for all of these problems.

**Singly-Linked Lists:** The first class of allowed structures we examine is acyclic singly linked lists. The vocabulary includes constants that represent program variables, a functional binary relation \( n \) that represents the next field, a unary relation \( r_{x,n} \) for each program variable \( x \) that represents reachability from \( x \) (a.k.a., unary reachability), and a binary relation \( p_n \) (path of \( n \)) that represents reachability between any two elements. The guard formulas are used to detect null dereferences or the formation of garbage or cycles. Monadic-uniform update formulas can be easily written for all the needed operations.

Table 6.3 lists the transformers for the field-manipulating operations. Update formulas for unchanged relations are omitted. The update formulas for reachability follow the ones described in [Hes03]. For traversal of a field, we use the free variable \( y_n \) of the guard formula to capture the target of the next field for \( y \) (\( x_n \) is used similarly in the removal of an edge).

To check feasibility of a focused abstract structure, we build a single candidate concrete structure s.t. the original structure is feasible iff it is the result of applying \( \beta \) on the candidate structure and the candidate structure satisfies the integrity constraints.

<table>
<thead>
<tr>
<th>Structures</th>
<th>Vocabulary</th>
<th>Feasibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acyclic SLL</td>
<td>( p_n, n, \text{PVar} )</td>
<td>Direct</td>
</tr>
<tr>
<td>Acyclic SLL</td>
<td>( r_{x,n}, n, \text{PVar, Colors} )</td>
<td>Direct</td>
</tr>
<tr>
<td>Cyclic SLL</td>
<td>( p_n, pcn, n, \text{PVar} )</td>
<td>Direct</td>
</tr>
<tr>
<td>Cyclic SLL</td>
<td>( r_{x,n}, r_{cx,n}, n, \text{PVar, Colors} )</td>
<td>Direct</td>
</tr>
<tr>
<td>DLL</td>
<td>( p_f, p_b, cf_b, cb_f, \text{PVar, Colors} )</td>
<td>Direct/Open</td>
</tr>
<tr>
<td>Ordered SLL</td>
<td>( r_{x,n}, r_{cx,n}, n, \text{dl}<em>{e}, \text{PVar, inOrd}</em>{n,dl_e}, \text{inROrd}_{n,dl_e} )</td>
<td>Open</td>
</tr>
<tr>
<td>Trees</td>
<td>( p, l, r, \text{PVar} )</td>
<td>Direct</td>
</tr>
<tr>
<td>Trees</td>
<td>( p, l, r, \text{PVar, Colors} )</td>
<td>MSO</td>
</tr>
<tr>
<td>NUC</td>
<td>( p, l, r, s_{x,y}, \text{PVar} )</td>
<td>Direct</td>
</tr>
<tr>
<td>NUC</td>
<td>( p, l, r, s_{x,y}, \text{PVar, Colors} )</td>
<td>MSO</td>
</tr>
<tr>
<td>Shared Trees</td>
<td>( p, l, r, \text{PVar} )</td>
<td>Open</td>
</tr>
</tbody>
</table>

Table 6.2: Summary of the shape-analysis problems and their feasibility-check status.
Update Formula for removal of an edge is not monadic as a basis for monadic-uniform update formulas. Fortunately, we can easily rewrite that formula to be monadic-uniform.

To analyze programs that manipulate cyclic singly-linked lists, we use the following vocabulary: constants represent program variables; two functional binary relations $l$ and $r$ represent

<table>
<thead>
<tr>
<th>Relation</th>
<th>Update Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = y.n$</td>
<td>$n(y, y_n) \land y \neq \text{null} \land (x = \text{null} \lor \vee_{z \neq x} r_{z,n}(x))$</td>
</tr>
<tr>
<td>$r'_{x,n}(v)$</td>
<td>$r_{y,n}(v) \land y \neq v$</td>
</tr>
<tr>
<td>$x.n\text{ext} = \text{null}$</td>
<td>$n(x,x_n) \land x \neq \text{null} \land (x_n = \text{null} \lor \vee_{z,x} r_{z,n}(x))$</td>
</tr>
<tr>
<td>$n'(v_1,v_2)$</td>
<td>(when $v_1 = x \Rightarrow v_2 = \text{null}, \text{default} \Rightarrow n(v_1,v_2))$</td>
</tr>
<tr>
<td>$p'_n(v_1,v_2)$</td>
<td>$p_n(v_1,v_2) \land \neg(p_n(v_1,x) \land p_n(x_n,v_2))$</td>
</tr>
<tr>
<td>$r'_z,n(v)$</td>
<td>$r_{z,n}(v) \land \neg(r_{z,n}(x) \land r_{x,n}(v) \land x \neq v)$</td>
</tr>
<tr>
<td>$x.n\text{ext} = y$</td>
<td>$x \neq \text{null} \land \neg r_{y,n}(x) \land n(x,\text{null})$</td>
</tr>
<tr>
<td>$n'(v_1,v_2)$</td>
<td>(when $v_1 = x \Rightarrow v_2 = y, \text{default} \Rightarrow n(v_1,v_2))$</td>
</tr>
<tr>
<td>$p'_n(v_1,v_2)$</td>
<td>$p_n(v_1,v_2) \lor (p_n(v_1,x) \land p_n(y,v_2))$</td>
</tr>
<tr>
<td>$r'_z,n(v)$</td>
<td>$r_{z,n}(v) \lor (r_{z,n}(x) \land r_{y,n}(v))$</td>
</tr>
</tbody>
</table>

Table 6.3: Monadic-uniform transformers for acyclic singly-linked lists.

Algorithm 6.4.1 (Checking Feasibility)

Replace every summary node with two concrete nodes connected by an edge, all incoming edges to the summary node go to the first concrete node, all outgoing edges from the summary nodes start from the second node. Each edge in the abstract structure is translated into a single edge in the concrete structure. We then simply compute $\beta$ on this structure and return true if it equals the original structure and satisfies the integrity constraints (i.e., $n$ is a total function).

Cyclicity: To handle cyclicity, we use the ideas from [Hes03], which allow for quantifier-free update of reachability in singly-linked lists. The update of [Hes03] is based on the addition of a binary relation, called PathCut, as an auxiliary relation. For every cycle, we call the last edge added to the cycle (i.e., the edge that closed the cycle) a cut edge. PathCut indicates reachability over $n$ minus the cut edges. When the cycle is broken, its cut edge is readded to PathCut. The update formula suggested by [Hes03] for removal of an edge is not monadic-uniform. Fortunately, we can easily rewrite that formula to be monadic-uniform.

To analyze programs that manipulate cyclic singly-linked lists, we use a vocabulary similar to that of acyclic singly-linked lists. The additional relations needed to allow updates to be monadic-uniform (and ease feasibility checking) are: $cut_n$ is a binary relation representing the cut edges, $pc_n$ is a binary relation representing PathCut, $rc_{z,n}(v)$ is a unary relation indicating $v$ is reachable from program variable $x$ using $pc_n$, and $c_n(v)$ is unary relation indicating that $v$ is on a cycle. The resulting abstraction is similar in the distinctions it makes to that of [MYRS05]. Because $cut_n$ is needed only to update itself, and the feasibility check can recover the cut edges from $pc_n$, we can remove $cut_n$ and still compute the most precise abstract transformer.

We use the DynQF updates by [Hes03] as a basis for monadic-uniform update formulas. Feasibility checking can be done using the same ideas as for acyclic lists with the necessary changes to support the cut edges.

Trees: To analyze trees using monadic-uniform transformers, we use the following vocabulary: constants represent program variables; two functional binary relations $l$ and $r$ represent
the left and right fields respectively; two new constants $x_l$ and $x_r$ for each program variable $x$ indicate the target of its left and right fields, respectively; a binary relation $p$ represents reachability (existence of a path) between any two elements (using any fields); unary relation $r_{x,sel}$ for each program variable $x$ represents reachability from the $sel$ field of $x$. The guard formulas verify that each operation maintains treeness.

The key to updating reachability in this case is the observation that between every two nodes there is at most one path. Thus, the paths that should be removed when removing an edge from $x$ to $x_l$ are exactly the ones that would have been added if this edge had been added.

We can either check feasibility by reduction to satisfiability of an MSO formula (similar to the $\hat{\gamma}$ of [YRS04]) on trees or we can check it directly (with lower complexity) by building a single candidate concrete structure in a way similar to singly-linked lists.

**No Undirected Cycles:** In [LAIS06], we introduced a class of structures whose underlying undirected graphs are acyclic (a.k.a. No Undirected Cycles). There we show an abstraction for handling this class of structures and algorithms for computing most precise abstract transformers for this abstraction. Structures with No Undirected Cycles are acyclic and have the interesting property that each pair of program variables can meet only once (i.e., there is a single shared node reachable from both variables s.t. none of the nodes pointing to that node are reachable from both variables). Furthermore, between any two nodes there is at most one path.

We now define an abstraction similar to [LAIS06] and apply our methodology. The vocabulary used for trees in extended with the following constants: For each pair of distinct program variables $x$ and $y$, we add $s_{x,y}$, which is the unique node in which $x$ and $y$ meet and create sharing (or $null$ if no such node exists). These are used in the guard formulas to detect formation of undirected cycles. We also maintain unary reachability from these constants. We can write a monadic-uniform guard formula using transitive closure that detects the formation of undirected cycles. We can check feasibility of such structures using methods similar to the ones using for trees.

While we can define an isomorphic abstraction to the one defined in [LAIS06], note that the abstract transformers defined there are hand-coded to handle the particular abstraction and are thus significantly more efficient than the ones that will be generated by this chapter’s methodology. Furthermore, we prove in [LAIS06] upper bounds on the size and number of abstract states. While these upper bounds apply to this isomorphic version as well, we do not know of a simple way to derive them directly from the properties of the predicates.

**Shared Trees:** Shared trees are graphs in which between any two nodes there is at most one (possibly empty) path. A way to visualize shared trees is that from every node looking down the graph you see a tree. Shared trees arise in applicative data structures (e.g., see [Mye84, Oka98]) and in operating systems and databases performing shadow paging (e.g., see [Bro89]).

We use the same vocabulary as in the case of trees. Updating reachability for this class of structures is done in the same way as in trees, because between any two nodes there is at most one path. Detecting when the shared-trees property has been violated is done by a guard formula when adding an edge. Again, the formula is monadic-uniform but not quantifier-free.

We are working on checking feasibility for shared trees in this vocabulary and believe it is decidable. Because shared trees have unbounded tree width, a direct translation into satisfiability of an MSO formula will not yield decidability.

**Uninterpreted Unary Relations:** Sets and Boolean fields can be added to any of the above shape-analysis problems by introducing uninterpreted unary relations (a.k.a. colors). We allow
addition and removal of an element from a set, query for existence of an element in a set, and selection of an arbitrary element from a set. The additional update formulas needed are trivial. Selection is done by using a guard formula with a free variable. The difficulty in checking feasibility when adding colors to a vocabulary, in contrast to the original feasibility-checking problem, comes from the fact that the colors can make distinctions that the original abstraction could not. The binary relations between the now-separate nodes need to be taken into account.

Checking feasibility for singly-linked lists can be done by first checking feasibility ignoring the colors, and then reducing the feasibility for each segment of the list to the Directed Chinese Postman Problem [EJ73], which can be solved in polynomial time. Checking feasibility for trees and structures with No Undirected Cycles, can be done by reduction to MSO.

Other cases: The relations required for analyzing doubly linked lists and ordered lists can also be maintained using monadic-uniform transformers.

We do not have a general feasibility check for any structure over the vocabulary of doubly-linked lists. However, we do know how to check feasibility for all the structures arising in most programs that manipulate doubly-linked lists (e.g., all the example programs of TVLA) because all such structures are only ever small perturbations of well-formed doubly linked lists.

In the third and final part of the thesis we explore ways to increase the applicability of shape analysis algorithms by allowing existing abstract domains to be extended to track sizes of data structures and their correlations.
Chapter 7

A Combination Framework for Tracking Partition Sizes

In this chapter, we describe an abstract interpretation based framework for tracking correlations between sizes of memory partitions. Instances of this framework can prove traditional properties such as memory safety and program termination but can also establish upper bounds on usage of dynamically allocated memory. Our framework also stands out in its ability to prove properties of programs manipulating both linked objects and arrays which is considered a difficult task.

Technically, we define an abstract domain that is parameterized by an abstract domain for tracking memory partitions (sets of memory locations) and by a numerical abstract domain for tracking relationships between cardinalities of the partitions. We describe algorithms to construct the transfer functions for the abstract domain in terms of the corresponding transfer functions of the parameterized abstract domains.

A prototype of the framework was implemented and used to prove interesting properties of realistic programs, including programs that could not have been automatically analyzed before.

Outline

We start by describing interesting applications of our framework, namely termination analysis, memory bounds analysis, and memory safety and functional correctness (Section 7.1). We continue with the technical part of the chapter which includes requirements from the set domain (Section 7.2), definition of the combined domain (Section 7.3), and algorithms for computing the abstract interpretation required operation for the combined domain (Section 7.4). Our framework enables verification of desirable properties of real-world examples, which to our knowledge, have not been analyzed before. We present experimental results illustrating the feasibility of our approach (Section 7.5.1). We also present a case-study regarding how the choice of the constituent set domain and the constituent numerical domain affects the precision of the set cardinality domain (Section 7.5).
7.1 Applications

Our work has several applications that are mentioned below, and we present experimental results for each of these applications in Section 7.5.1.

Proving Memory Safety as well as Data-structure Invariants. Often some numeric program variables are related to size of data-structures, and are used to iterate over data-structures. Our analysis can automatically track these relationships between program variables and size of data-structures. These relationships are important to prove memory safety. A common pattern in C where these relationships arise is when lists are converted into arrays and are then iterated over in the same loop that iterates over the corresponding array without having a null dereference check. These relationships are also important to prove data-structure invariants. This happens frequently in object-oriented code wherein base class libraries maintain length of data-structures like queues or lists. In Section 7.1.1, we present a procedure from Microsoft product code that illustrates the importance of tracking relationships between numeric variables and sizes of data-structures for proving both memory safety and data-structure invariants. We do not know of any existing technique that can automatically verify the correctness of assertions in this code.

Bounding Memory Allocation. This involves bounding the sizes of the partitions corresponding to the allocation statements in the program. This is especially important in embedded systems, where we would like to prove statically that the amount of memory that the system is shipped with is sufficient to execute desired applications. We present examples of bounding memory allocation in terms of sizes of input data-structures for deep copy routines over a variety of data-structures in Section 7.5.

Proving Termination. The oldest trick for proving termination of loops has been that of finding a ranking function [Tur89]. A ranking function for a loop is a function whose value decreases in each iteration and is bounded below by some finite quantity. There has recently been a lot of work on discovering fancy forms of numerical ranking functions (lexicographic polyranking functions [BMS05], disjunctively well-founded linear ranking functions [PR04]). However, for several programs based on iteration over data-structures, the ranking function is actually related to the cardinality of some partition of the data-structure. Our technique can find such ranking functions and can in fact even prove a bound on the loop iterations by instrumenting a counter variable in the loop and discovering invariants that relate the counter variable and sizes of partitions. We illustrate this by means of the BubbleSort example in Section 7.1.2. We do not know of any existing technique that can prove even termination of this example automatically.

7.1.1 String Buffer Example

This example illustrates the use of our analysis for proving memory safety as well as establishing data-structure invariants.

Consider the string buffer data-structure StringBuffer described in Figure 7.1(a), as taken from Microsoft product code. A string buffer is implemented as a list of chunks (in reverse order, so that appends are fast). A chunk consists of a character array content whose
typedef struct {
    int len; int size;
    char* content;
    StringBuffer* previous;
} * StringBuffer;

Remove(StringBuffer *x, int startIndex, int count) {
    Assume(startIndex ≥ 0 ∧ count > 0);
    n := 0;
    for (y := x; y ≠ null; y := y -> previous)
        n := n + (y -> len);
    if (n < startIndex + count) return;
    endIndex := startIndex + count;
    y := x; m := n - (y -> len);
    for (; m > endIndex; m := m - (y -> len))
        Assert(y ≠ null); y := y -> previous;
    endChunk := y; endChunkOff := m;
    for (; m > startIndex; m := m - (y -> len))
        Assert(y ≠ null); y := y -> previous;
    startChunk := y; startChunkOff := m;
    if (startChunk ≠ endChunk)
        endChunk -> previous := startChunk;
    startChunk -> len := startIndex - startChunkOff;
    tmp := endIndex - endChunkOff;
    else
        tmp := endIndex - startChunkOff;
    endChunk -> len := (endChunk -> len) - tmp;
    for (i := 0; i < endChunk -> len; i := i + 1)
        i' := i + tmp;
    endChunk -> content[i] := endChunk -> content[i'];
    n' := 0;
    for (y := x; y ≠ null; y := y -> previous)
        n' := n' + (y -> len);
    n' := n' + (y -> len);
    Assert(n' = n - count);
}

Figure 7.1: Remove method of StringBuffer data-structure (adapted slightly from Microsoft product code). The method has 2 memory safety assertions and one assertion that relates the sizes of the string buffer at the entry and exit: (a) StringBuffer data-structure; (b) Method to remove count elements from string buffer x starting at location startIndex.
Interesting Invariants at program point $\pi$

1. $n = |R_1(x)| - |R_1(y)|$
2. $n = |R_1(x)|$
3. $m = |R_1(y)| - (y \rightarrow \text{len}) \land m > 0$
4. $m = |R_1(y)| - (y \rightarrow \text{len}) \land m > 0$
5. $\text{startChunkOff} = |R_1(\text{startChunk})| - (\text{startChunk} \rightarrow \text{len})$
6. $\text{endChunkOff} = |R_1(\text{endChunk})| - (\text{endChunk} \rightarrow \text{len})$
7. $|R_1(x)| = n - (\text{endChunkOff} - \text{startChunkOff} - (\text{startChunk} \rightarrow \text{len}))$
8. $(\text{startChunk} \rightarrow \text{len}) = \text{startIndex} - \text{startChunkOff}$
9. $|R_1(x)| = n - (\text{endChunkOff} - \text{startChunkOff} - (\text{startChunk} \rightarrow \text{len}))$
10. $\text{startIndex} = \text{endIndex} - \text{endChunkOff}$
11. $\text{startIndex} = \text{startIndex} - \text{startChunkOff}$
12. $\text{startChunkOff} = |R_1(\text{startChunk})| - (\text{startChunk} \rightarrow \text{len})$
13. $\text{endChunkOff} = |R_1(\text{endChunk})| - (\text{endChunk} \rightarrow \text{len})$
14. $|R_1(x)| = n - (\text{endChunkOff} - \text{startChunkOff} - (\text{startChunk} \rightarrow \text{len}))$

Figure 7.2: Remove method of StringBuffer data-structure: (a) Interesting invariants at various program points in the Remove method that are necessary to prove the given assertions; The invariants given hold right before $\pi$ is executed. (b) An example of a string buffer $x$ before and after the remove method.
7.1. Applications

The total size is \texttt{size} and its \texttt{len} field denotes the total number of valid characters in the array. This program contains the following features that make the task of analysis/verification challenging: (i) The usage of dynamic memory and pointers with destructive pointer mutations and (ii) The usage of arrays and arithmetic. These features are common in C. Moreover, Java ADT implementations, such as hash-maps, raise similar challenges.

The Remove method over string buffer (Figure 7.1(b)) takes as input a non-negative start index \texttt{startIndex} and a positive integer \texttt{count} and deletes \texttt{count} characters starting from \texttt{startIndex}. The first loop (Lines 3-4) counts the total length of characters inside string buffer and stores it into the variable \texttt{n}. The second loop (Lines 8-9) finds the first chunk \texttt{endChunk} from which characters are to be removed, while the third for loop (Lines 11-12) finds the last chunk \texttt{startChunk} from which characters are to be removed. Both these loops have a memory safety assertion at lines 9 and 12 respectively. The loop condition for each of these loops indicates that \texttt{m} is positive; it does not explicitly indicate that \texttt{y} \neq \texttt{null}. However, the assertions hold because there is a relationship between \texttt{m} and \texttt{y}, namely that the total number of characters in the string buffer before \texttt{y} is equal to \texttt{m}. Hence, if \texttt{m} > 0, it implies that \texttt{y} \neq \texttt{null}. The framework presented in this chapter can be used to automatically discover such relationships between numerical program variables and sizes of appropriate partitions of data-structures. For this purpose, we require a set domain whose base-set constructor can represent the set of all characters in a chunk \texttt{x} and the chunks before it (referred to as \texttt{R}_i(\texttt{x}) in Figure 7.2(a)). When coupled with a relational numerical domain that can represent linear inequalities between numerical variables, our combination framework yields a set cardinality analysis that can discover the required invariants (shown in Figure 7.2(a)).

The next loop (Lines 21–23) slides down an appropriate number of characters in \texttt{endChunk}. The last loop (Lines 25–26) counts the total number of characters in the string buffer and stores it in the variable \texttt{n’}. Line 27 then asserts that \texttt{n’} (whose value is the total number of characters in the string buffer at the end of the Remove method) is less than \texttt{n} (whose value is the total number of characters in the string buffer at the beginning of the Remove method) by an amount equal to \texttt{count}. The assertion holds because lines 14–23 remove \texttt{count} characters from the string buffer by destructively updating the data-structure and adjusting the value of then \texttt{len} field of appropriate chunks. The approach presented in this chapter can automatically discover such relationships that relate the sizes of various partitions of data-structures. These relationships along with the required invariants at other program points are shown in Figure 7.2(a).

Figure 7.2(b) describes the effect of the Remove method over an example string buffer \texttt{x}. The filled part (both dark filled part and lightly filled part) of each chunk represents the original characters in the string buffer \texttt{x}. The solid filled part represents the characters to be removed. The solid filled part is identified by the second loop (Lines 8–9) and the third loop (Lines 11–12).

7.1.2 BubbleSort Example

This example illustrates the use of our analysis for proving termination as well as computing a bound on the number of loop iterations.

Consider the BubbleSort procedure shown in Figure 7.3 that sorts an input array \texttt{A} of length \texttt{n}. Ignore lines 2 and 4 that update a counter variable \texttt{c}. The algorithm works by repeatedly iterating through the array to be sorted, comparing two items at a time and swapping them
BubbleSort(int* A, int n)
[1] change := true;
[2] c := 0;
[3] while (change) {
[4]   c := c + 1;
[5]   change := false;
[6]   for (j := 0; j < n - 1; j := j + 1) {
[8]       Swap(A[j], A[j + 1]);
[9]       change := true;
[10]     }
[12] }

Figure 7.3: BubbleSort Routine.

if they are in the wrong order (Line 8). The iteration through the array (Loop in lines 6-11) is repeated until no swaps are needed (indicated by the change Boolean variable), which indicates that the array is sorted.

Notice that establishing a bound on the number of iterations of the outer while-loop of this procedure is non-trivial; it is not immediately clear why the outer while loop even terminates. However, note that in each iteration of the while loop, at least one new element “bubbles” up to its correct position in the array (i.e., it is less or equal to all of its successors). Hence, the outer while loop terminates in at most n steps. The set cardinality analysis that we introduce in this chapter can automatically establish this fact by computing a relationship between an instrumented loop counter c (to count the number of loop iterations of the outer while loop) and the number of elements that have been put in the correct position. In particular, the set cardinality analysis computes the invariant that c is less than or equal to the size of the set of the array indices that hold elements at their correct position (provided the set cardinality analysis is constructed from a set analysis whose base-set constructor can represent such a set).

### 7.2 Set Domain

In this section, we formalize the notion of a set abstract domain. In particular, we describe the interface that a set domain should support in order for it to be combinable with a numerical domain in our combination framework.

A set domain $\mathcal{P}$ consists of set-domain elements that are related by some partial order $\preceq_{\mathcal{P}}$. A set-domain element $P$ of a set domain should expose some bounded collection of (interpreted) base-sets, referred to as $\text{BaseSets}(P)$. Examples for base-sets are $R_1(z)$, where $z$ is a program variable, as used in Section 7.1.1. The numerical domain tracks relationships of the cardinalities of the base-sets in $\text{BaseSets}(P)$. 

A set domain exports all the standard operations needed to perform abstract interpretation (namely Join, Widen, Eliminate, PostPredicate, as defined in Section 7.4). Besides these operations, a set domain also needs to support the following operations in order for it to be combinable with a numerical abstract domain to enable tracking of numerical properties over cardinalities of base-sets.

7.2.1 Witness Operator

The set domain \( \mathcal{P} \) exports an operation \( \text{Witness}_\mathcal{P} \) that takes as input a collection \( S \) of base-sets and a set-domain element \( P \). Intuitively, \( \text{Witness}(S, P) \) returns the interpretation of base-sets in \( S \) in terms of the base-sets that occur in \( P \) using the information from \( P \). This is needed because base-sets are interpreted. Thus, even if the base-sets in \( S \) are semantically identical to some base-sets in \( \text{BaseSets}(P) \) (a common case in our setting), there is no way to infer this equality without the help of the set domain. Without \( \text{Witness} \), the numerical domain cannot infer anything about cardinalities of base-sets in \( S \) (even when it has information about cardinalities of base-sets that occur in \( P \)).

The result of \( \text{Witness}(S, P) \) is in the form of normalized set-inclusion relationships. A normalized set-inclusion relationship (implied by \( P \)) between a collection \( T \) of base-sets is a relationship of the form

\[
\bigcup_{i \in I} p_i \subseteq \bigcup_{j \in J} p'_j
\]

where \( p_i, p'_j \in T \) (for all \( i \in I, j \in J \)) and \( P \) implies that \( p_i \)'s are all mutually disjoint (i.e., under any concretization of \( P \), the interpretations of \( p_i \)'s are mutually disjoint). Furthermore, \( I \) is maximal and \( J \) is minimal. In practice, most of the relationships we get are of the form \( p = \bigcup p_i \) where the \( p_i \)'s are mutually disjoint.

**Example 7.2.1** Consider a simple set domain whose base sets are of the form \( q_{nk} \) where \( n \) is a positive integer and \( k \) is a natural. We define \( x \in q_{nk} \) iff \( (x \mod n) \equiv k \). Let \( \text{BaseSets}(P) = \{q_{30}, q_{20}\} \) and \( S = \{q_{60}, q_{63}, q_{40}\} \). We have

\[
\text{Witness}(S, P) = \{ q_{60} \cup q_{63} \subseteq q_{30}, q_{30} \subseteq q_{60} \cup q_{63}, q_{60} \subseteq q_{20}, q_{40} \subseteq q_{20} \}
\]

Note that although \( q_{60} \cup q_{40} \subseteq q_{20} \), this relation is not part of the witness since \( q_{60} \) and \( q_{40} \) are not disjoint.

The advantage of normalized set-inclusion relationships is that they can be easily translated into numerical relationship over cardinalities of base-sets, which is something that a numerical domain can understand. They will thus be used to relate the cardinalities of the base-sets before and after an abstract operation (see the \( \mathcal{P}2\mathcal{N} \) operation in Section 7.3).

The soundness of the combination framework only requires that the base-sets of the left side of a normalized set-inclusion relationship \( p_i \) be all mutually disjoint in \( P \). The soundness does not require \( I \) to be maximal or \( J \) to be minimal, and neither does it require the \( \text{Witness} \) operator to return all such relationships. However, a more precise collection of such relationships leads to a more precise combined domain.
7.2.2 Generate Operator

The set domain also has an interface \( \text{Generate}_{P} \) to generate information about the cardinality of any base-set in relation to any constant. The function \( \text{Generate}_{P} \) takes a set-domain element \( P \), and returns a collection of inequalities of the form \( |p| \leq c \) or \( |p| \geq c \) (where \( p \in \text{BaseSets}(P) \) and \( c \) is some non-negative integer constant) that are implied by \( P \). In case the set-domain element is inconsistent (i.e., does not represent any concrete element), \( \text{Generate}_{P} \) simply returns false.

7.2.3 Examples of Set Domains

In this section, we show that several popular heap/shape analysis domains can be viewed as set domains. In particular, we show that for each of these domains we can easily implement the required operations to be used in the combination framework.

In all three cases the domain is a powerset domain over some base domain. The combination with the numerical domain is performed at the level of the base domain. The construction of the powerset domain is done on top of the combined domain.

Canonical Abstraction

Canonical Abstraction [SRW02] is a powerful domain for shape analysis (also see Chapter 2). The domain is a powerset of abstract shape graphs. Each abstract shape graph is based on equivalence classes of memory locations based on the unary predicates that hold for them. These equivalence classes are called abstract nodes. Canonical abstraction maintains the invariant that each abstract node \( \eta \) must represent at least one memory location. Canonical abstraction can maintain binary information that holds universally on abstract nodes, i.e., formulas of the form \( \forall v_1, v_2. \eta_1(v_1) \land \eta_2(v_2) \rightarrow p(v_1, v_2) \) where \( p \) is a binary predicate, \( v_i \) ranges over memory locations and \( \eta_i(v_i) \) holds when \( v_i \) is in the equivalence class defined by \( \eta_i \).

Specifically, canonical abstraction can express the notion of an abstract node \( \eta \) that represents exactly one memory location using the formula \( \text{unit}(\eta) = \forall v_1, v_2. \eta(v_1) \land \eta(v_2) \rightarrow v_1 = v_2. \)

Required Operations

The base-sets of an abstract shape graph are its abstract nodes. Thus,

\[
\text{BaseSets}(P) \equiv \{ \eta \mid \eta \in P \}
\]

The \( \text{Witness}_{CA} \) operation is straightforward as the abstract nodes are based on equivalence classes. Thus, abstract nodes have canonical names, which allow to easily relate abstract nodes from different abstract shape graphs. Furthermore, because the base-sets are equivalence classes of unary predicates, they are necessarily disjoint.

The cardinality constraints arise from the non-emptiness requirement and the definition of \( \text{unit} \), i.e.,

\[
\text{Generate}_{CA}(P) \equiv \bigwedge \{ \{ |\eta| \geq 1 \mid \eta \in \text{BaseSets}(P) \} \cup \{ |\eta| = 1 \mid \eta \in \text{BaseSets}(P), \text{unit}(\eta) \in P \} \}
\]

Finally, the canonical abstraction domain can easily interpret cardinality constraints of the form \( |\eta| = 1 \) by asserting the formula defining \( \text{unit}(\eta) \). Note that although this was not
part of the standard interface for the canonical abstraction domain, there is an existing mechanism to easily support this constraint. We use this choice of set domain in our experiments in Section 7.5.1.

### Boolean Heaps

The Boolean Heaps Domain [PW05] is a powerset domain over Boolean Heaps. As in canonical abstraction, each Boolean heap is formed of equivalence classes of unary predicates. However, there is no requirement for non-emptiness of abstract nodes and no extra binary information. Boolean heaps support predicates of the form \( v = t \) where \( t \) is a ground term. If such a unary predicate holds on an abstract node, this node represents at most one memory location. Thus, we shall define \( \text{unique}(\eta) \) to hold for any \( \eta \) in which there is a predicate of the form \( v = t \).

**Required Operations**

The base-sets of a Boolean heap are its abstract nodes. Thus,

\[
\text{BaseSets}(P) \overset{\text{def}}{=} \{ \eta \mid \eta \in P \}
\]

The Witness\(_{BH}\) operation is very similar to that of canonical abstraction as both abstractions are based on equivalence classes of unary predicates. Because abstract nodes are equivalence classes of unary predicates, the resulting base-sets are necessarily disjoint.

Boolean heaps have the notion of a unique base-set but no requirement for non-emptiness. Thus, \( \text{Generate}_{BH} \) is defined by:

\[
\text{Generate}_{BH}(P) \overset{\text{def}}{=} \bigwedge (\{ |\eta| \geq 0 \mid \eta \in \text{BaseSets}(P) \} \cup \{ |\eta| \leq 1 \mid \eta \in \text{BaseSets}(P), \text{unique}(\eta) \})
\]

Given a cardinality constraint of the form \( |\eta| = 0 \), the Boolean heap can soundly remove \( \eta \) from the Boolean heap, thus reducing its size and complexity.

### Separation Domain

We use the separation domain of Distefano et al [DOY06] as a representative of Separation Logic [Rey02] based abstract domains. The separation domain is a powerset domain of symbolic heaps. A symbolic heap \( P \) is a separation logic formula of the form: \( \exists x'_1, \ldots, x'_n. (\bigwedge_{\pi \in \Pi_P} \pi) \land (\star Q) \)

\( \Pi_P \) is a set of equalities and dis-equalities between pointer variables (and possibly nil). \( \Sigma_P \) can contain either

- **junk** — any non-empty memory,
- \( x \mapsto y \) — a memory that contains a single address \( x \) whose content is \( y \), or
- \( ls(x, y) \) — a non-empty singly-linked list starting at address \( x \) and whose last element points to the address \( y \).\(^1\)

\(^1\)ls\((x, y)\) is defined recursively as \( x \neq y \land (x \mapsto y \lor \exists z. x \mapsto z \star ls(z, y)) \)
The formula $\varphi_1 \star \varphi_2$ holds on memories that can be decomposed into two disjoint parts s.t. $\varphi_1$ holds on one of them and $\varphi_2$ holds on the other.

**Required Operations** We use the members of $\Sigma_P$ (a.k.a. star conjuncts) as base-sets, i.e.,

$$\text{BaseSets}(P) \overset{\text{def}}{=} \Sigma_P$$

The $\text{Witness}_{SL}$ operation is computed by finding which star conjuncts are subsets of other star conjuncts. Such algorithms already exist in the domain for performing containment checks. The locality of most operations (i.e., the fact that they modify only small parts of the memory each time) means that most of the star conjuncts will appear verbatim. This allows for efficient implementation of the $\text{Witness}_{SL}$ operation. Furthermore, the base-sets are star conjuncts, thus all base-sets of an abstract element are disjoint.

Because all star conjuncts represent non-empty sets and a star conjunct of the form $x \mapsto y$ is of cardinality 1, Generate$_{SL}$ is defined by:

$$\text{Generate}_{SL}(P) \overset{\text{def}}{=} \bigwedge \left( \{|\eta| \geq 1 | \eta \in \text{BaseSets}(P)\} \cup \{|\eta| = 1 | \eta \in \text{BaseSets}(P), \eta \equiv x \mapsto y\} \right)$$

Finally, if the numerical domain discovers that $|ls(x, y)| = 1$ then $ls(x, y)$ can be strengthened to $x \mapsto y$.

We use the syntax of separation logic in our examples\(^2\).

### 7.3 Set Cardinality Domain

In this section, we define the notion of a set cardinality domain that is obtained by a combination of a set domain and a numerical domain. Given a set domain $\mathcal{P}$ and a numerical domain $\mathcal{N}$, we define their combination to be the set cardinality domain $\mathcal{P} \uplus \mathcal{N}$ whose elements are pairs $(P, N)$, where $P$ is some element that belongs to the set domain $\mathcal{P}$ and $N$ is some element that belongs to the numerical domain $\mathcal{N}$. Furthermore, $N$ uses some special variables, each of which denotes the cardinality of some base-set from $\text{BaseSets}(P)$. We denote the special variable corresponding to a base-set $p$ by $\tilde{p}$. In addition, we use in our examples the shorthand

\(^2\)Note that the join algorithm we use is a refined version of the one in [DOY06] and uses the ideas in [MSRF04].
We use the notation $\text{SpecialVars}(N)$ to denote the set of all special variables (i.e., non-program variables that corresponds to the cardinality of some base-set) that occur in $N$. For notational convenience, we overload the notation $\text{SpecialVars}(P)$ to denote the set of all special variables that denote the cardinality of some base-set in $P$, i.e.,

$$\text{SpecialVars}(P) = \{ \widetilde{p} \mid p \in \text{BaseSets}(P) \}$$

Hence, every element $(P, N)$ that belongs to the domain $\mathcal{P} \bowtie N$ has the property that $\text{SpecialVars}(N) \subseteq \text{SpecialVars}(P)$. Also, without loss of any generality, we assume that for any two distinct elements $(P_1, N_1)$ and $(P_2, N_2)$, $\text{SpecialVars}(P_1) \cap \text{SpecialVars}(P_2) = \emptyset$ since we can always rename the special variables that denote the cardinality of some base-sets.

The pre-order $\preceq_{\mathcal{P} \bowtie \mathcal{N}}$ between two elements of the combined domain is defined as follows:

$$(P_2, N_2) \preceq_{\mathcal{P} \bowtie \mathcal{N}} (P_1, N_1) \overset{\text{def}}{=} P_3 \preceq_{\mathcal{P}} P_1 \land N_4 \preceq_{\mathcal{N}} N_1$$

where $(P_3, N_3) = \text{Saturate}(P_2, N_2)$

$N_4 = \text{P2N}(P_3, P_1, N_3)$

The pre-order above has two important components $\text{Saturate}$ and $\text{P2N}$, which are described below.

**Saturate** The $\text{Saturate}$ function takes as input an element $(P, N)$ from the combined domain and returns another element $(P', N')$ from the combined domain. $P'$ and $N'$ are obtained from $P$ and $N$ respectively by repeated sharing of information about relationships of base-set sizes with integral constants using the $\text{Generate}_P$ interface exported by the set domain (as described in Section 7.2) and the $\text{Generate}_N$ interface that can be provided by a numerical domain as follows:

$$\text{Generate}_N(N) \overset{\text{def}}{=} \bigwedge_{x \in U} \text{Eliminate}_N(N, \mathcal{V}(N) - \{x\})$$

where $U = \text{SpecialVars}(N)$

The function $\text{Eliminate}_N(N, V)$ eliminates all variables in set $V$ from the abstract element $N$. The function $\text{Eliminate}_N$ is part of the standard abstract interpretation interface that the numerical domain $N$ comes equipped with. $\mathcal{V}(N)$ denotes the set of all variables that occur in $N$.

The $\text{Saturate}$ function above is inspired by the Nelson-Oppen methodology of combining decision procedures for disjoint theories [NO78], where elements from different theories share variable equalities (since that is the only information that can be understood by elements from both theories) until no more equalities can be shared. In our case, the information that can be understood by the set-domain element as well as the numerical element involve relating the size of any base-set with a constant. The Nelson-Oppen decision procedure terminates because the number of independent variable equalities that can be shared is bounded above by the number of variables. In our case, the number of relationships that can be shared is potentially unbounded since there is no limit on the size of the constants. To address this issue, we allow for sharing of only those relationships that involve constants up to a bounded size, say 2. This
is because, in practice, all instances of set domains can usually only make use of the information whether the size of a base-set is 0, 1, or more than 1. Bounding the size of constants that can be shared during the saturation process guarantees efficient termination of the Saturate function.

We show below an example, where the Saturate function leads to repeated sharing of information between a set-domain element \( P \) and a numerical element \( N \).

**Example 7.3.1** Consider a program that traverses two linked lists of the same length. One of the lists is pointed to by \( x \) and the other by \( y \). Traversing the first list using the statement \( z = x \rightarrow \text{next} \) will cause the set domain to perform a case split on whether \( x \) is a singleton list or not. The case in which \( x \) is a singleton yields the element \((P, N)\) where \( P \) is \([x \mapsto \text{nil}]^A \ast [\text{ls}(y, \text{nil})]^B \wedge z = \text{nil} \) and \( N \) is \( A = B \). Calling Generate\(_{P}(P)\) results in \( A = 1 \wedge B \geq 1 \), which is used to strengthen \( N \) yielding \( A = 1 \wedge A = B \). Finally, Generate\(_{N}(A = 1 \wedge A = B)\) is \( A = 1 \wedge B = 1 \), i.e., \( x \mapsto \text{nil} \) is \( 1 \wedge |\text{ls}(y, \text{nil})| = 1 \). When used to strengthen \( P \) this yields \( x \mapsto \text{nil} \ast y \mapsto \text{nil} \wedge z = \text{nil} \). Thus, using the cardinality information we have discovered that the second list is a singleton as well.

Saturating the input abstract elements is the first step in all the abstract domain operations described in Section 7.4. However, in the examples there we use saturated elements as inputs to be able to concentrate on other interesting aspects of the algorithms.

**P2N Operator** Note that the base-sets in \( P_3 \) may have different names than those in \( P_1 \), yet the base-sets in \( P_3 \) might be related to those in \( P_1 \) since these are interpreted base-sets. The function \( \text{P2N}(P_3, P_1, N) \) performs the role of relating the sizes of the base-sets in \( P_1 \) and \( P_3 \) using the Witness operator, translating this into a numerical relationship, and communicating this information to \( N \).

Given any two set-domain elements \( P, P' \), and a numerical element \( N \), the function \( \text{P2N}(P, P', N) \) yields a numerical element that is more precise than \( N \) and incorporates information about numerical relationships between sizes of base-sets in \( P' \) and those in \( P \).

\[
\text{P2N}(P, P', N) \overset{\text{def}}{=} \text{PostPredicate}_{N}(N, S)
\]

where \( S \) is the conjunction of linear inequalities, one corresponding to each normalized set-inclusion relationship in \( \text{WS} = \text{Witness(BaseSets}(P'), P) \).

\[
S = \{ \sum_i \tilde{p}_i \leq \sum_j \tilde{p}'_j \mid (\bigcup_i p_i \subseteq \bigcup_j p'_j) \in \text{WS} \}
\]

The function \( \text{PostPredicate}_{N}(N, \text{pred}) \) strengthens \( N \) by assuming the predicate \( \text{pred} \).

We require that the \( \text{P2N} \) operator satisfies the following transitivity property.

**Property 7.3.2** For any set-domain elements \( P, P', P'' \) and any numerical element \( N \), if \( P \preceq_{P} P' \preceq_{P} P'' \), then \( \exists V : \text{P2N}(P', P'', \text{P2N}(P, P', N)) = \text{P2N}(P, P'', N) \), where \( V = \text{SpecialVars}(P') \).

The operator \( \text{P2N} \) is used extensively in the implementation of the abstract domain operations for the combined domain to relate base-sets coming from different set-domain elements, and strengthen the given numerical element with this information.
Theorem 7.3.3 The relation $\preceq_{\mathcal{P}\times\mathcal{N}}$ defined above is, in fact, a pre-order.

The proof of Theorem 7.3.3 is given in Appendix D.1. To recap, the pre-order saturates the left element to share information between the set domain and the numerical domain. It then uses $\mathcal{P}\times\mathcal{N}$ to relate the base sets of the left element to those of the right element.

The pre-order, as stated above, also defines the logic or formalizes the reasoning power of our combination framework. In that regard it is related to decision procedures that have been described for logics that combine sets, and numerical properties of their cardinalities. However, the two main differences are: (a) Our combination framework is modular wherein any set analysis can be combined with any numerical analysis, and (b) more importantly, we show how to perform abstract interpretation of a program over such a logic. Performing abstract interpretation requires many more transfer functions besides a decision procedure (such as Join, Widen, Eliminate, etc).

7.4 Abstract Interpreter for the Set Cardinality Domain

Let $\mathcal{P}$ and $\mathcal{N}$ be any set domain and numerical domain respectively. In this section, we show how to efficiently combine the abstract interpreters that operate over the abstract domains $\mathcal{P}$ and $\mathcal{N}$ to obtain an abstract interpreter that operates over the set cardinality domain $\mathcal{P}\times\mathcal{N}$. Our combination methodology yields the most precise abstract interpreter for the set cardinality domain $\mathcal{P}\times\mathcal{N}$ relative to the pre-order defined in Section 7.3. The key idea of our combination methodology is to combine the corresponding transfer functions of the abstract interpreters that operate over the domains $\mathcal{P}$ and $\mathcal{N}$ to yield the transfer functions of the abstract interpreter that operates over the domain $\mathcal{P}\times\mathcal{N}$.

An abstract interpreter performs a forward analysis on the program computing invariants (which are elements of the underlying abstract domain over which the analysis is being performed) at each program point. The invariants are computed at each program point from the invariants at the preceding program points in an iterative manner using appropriate transfer functions. We first describe our program model in Section 7.4.1. In the subsequent sections, we describe the construction of these transfer functions for the set cardinality domain $\mathcal{P}\times\mathcal{N}$ in terms of the transfer functions for the individual domains $\mathcal{P}$ and $\mathcal{N}$.

7.4.1 Program Model

We assume that each procedure in a program is abstracted using the flowchart nodes shown in Figure 7.4.

We allow for assume and assert program statements of the form $\text{assume}(\text{pred})$ and $\text{assert}(\text{pred})$, where $\text{pred}$ is a predicate that is either understood by the set domain $\mathcal{P}$, or it is a linear inequality predicate. The linear inequality predicate can be over program variables and over special variables that denote the cardinality of some base-set that can be specified using some base-set constructors exported by the set domain $\mathcal{P}$.

Since we allow for assume statements, without loss of generality, we can treat all conditionals in the program as non-deterministic (i.e., control can flow to either branch irrespective of the program state before the conditional). A join node has two incoming edges. Note that a join node with more than two incoming edges can be reduced to multiple join nodes each with two incoming edges.
We now describe the transfer functions for each of the flowchart nodes.

### 7.4.2 Join Node

The abstract element \( E \) after a join node (Figure 7.4(a)) is obtained by computing the join of the elements \( E_1 \) and \( E_2 \) before the join node using the join operator.

\[
E = \text{Join}_\mathcal{D}(E_1, E_2)
\]

The join operator \( \text{Join}_\mathcal{D} \) for a domain \( \mathcal{D} \) takes as input two elements \( E_1 \) and \( E_2 \) from domain \( \mathcal{D} \) and computes an optimal upper bound of \( E_1 \) and \( E_2 \) with respect to the pre-order \( \preceq_\mathcal{D} \). The following definition makes this more precise.

**Definition 7.4.1 (Join Operator \( \text{Join}_\mathcal{D} \))** Let \( E = \text{Join}_\mathcal{D}(E_1, E_2) \). Then,

- **Soundness**: \( E_1 \preceq_\mathcal{D} E \) and \( E_2 \preceq_\mathcal{D} E \).

- **Completeness**: If \( E' \) is such that \( E_1 \preceq_\mathcal{D} E' \) and \( E_2 \preceq_\mathcal{D} E' \) and \( E \preceq_\mathcal{D} E' \), then \( E \preceq_\mathcal{D} E' \).

Figure 7.5 shows how to implement the join operator \( \text{Join}_{\mathcal{P} \times \mathcal{N}} \) for the set cardinality domain \( \mathcal{P} \times \mathcal{N} \) using the join operators \( \text{Join}_\mathcal{P} \) and \( \text{Join}_\mathcal{N} \) for the set and numerical domains in a modular fashion. The implementation also makes use of the eliminate operator \( \text{Eliminate}_\mathcal{N} \) for the numerical domain, which is described in Section 7.4.3.

**Example 7.4.2** We explain the implementation of the \( \text{Join}_{\mathcal{P} \times \mathcal{N}} \) operator by considering the example in Figure 7.5(b). This is a simplified example of a situation that occurs during an in-place reversal of a linked list. The first input \( (P_1, N_1) \) represents two disjoint lists, the list
Eliminate\(_{\mathcal{P}\bowtie\mathcal{N}}\)((P,N),\ell) = \\
\text{Inputs:} \\
P = [ls(x,z)]^A \bowtie [ls(z,\text{nil})]^B \\
N = A \geq 1 \land B \geq 1 \land A = n \land B = k \\
\ell = z \\

\text{Trace of } Eliminate\(_{\mathcal{P}\bowtie\mathcal{N}}\)((P,N),\ell): \\
P_1 = [ls(x,\text{nil})]^C \\
N_1 = A + B = C \land A \geq 1 \land B \geq 1 \land A = n \land B = k \\
V = \{z,A,B\} \\
N_2 = C = n + k \land n \geq 1 \land k \geq 1 \\

(a) Algorithm 

The first step in joining the input elements is to saturate both of them. Remember that in all the examples, we use saturated elements to be able to concentrate on the important issues. Thus, the saturate operation has nothing to add.

Now, the join of the set domain is performed yielding \(P\), which represents two disjoint lists pointed to by \(x\) and \(y\). The \(\mathcal{P}\bowtie\mathcal{N}\) operator is used to strengthen the numerical element by relating the cardinalities of the base-sets of \(P\) to the cardinalities of the base-sets in the original elements. This is done using the Witness operation, which interprets the base-sets of \(P\) using the base-sets of the original elements\(^3\). For \(N_1\) this means that \(A = E\) and \(B = F\) and for \(N_2\) this means that \(C = E\) and \(D = F\). Note that without \(\mathcal{P}\bowtie\mathcal{N}\) there will be no relation between the base-sets of the two inputs and thus the numerical join would simply return \(n \geq 2\). Next, the numerical join is performed. The join loses the fact that one of the lists was a singleton, but retains the important information that the sum of the lengths of the lists is \(n\).

Finally, the numerical variables corresponding to the original base-sets are eliminated to ensure that all the special variables in the numerical element come from the set-domain element. In case of polyhedra join, this has no effect as any information on the original special variables relates to only one of the inputs and is thus lost in the join.

Theorem 7.4.3 The join operator described in Figure 7.5(a) satisfies both the soundness and the completeness property stated in Definition 7.4.1 (provided the join operators for the base domains, Join\(_\mathcal{P}\) and Join\(_\mathcal{N}\), satisfy these properties, and the eliminate operator for the numerical domain, Eliminate\(_N\), satisfies the respective soundness and completeness properties stated in Definition 7.4.4 on Page 113).

\(^3\)Because any relation added by \(\mathcal{P}\bowtie\mathcal{N}\) is based on the Witness operation and the original elements are saturated, no new relationships will be added among the original base-sets.
Proof: Consider the pseudo-code shown in Figure 7.5(a). First note that the \((P, N)\) is an element of the set cardinality domain \(\mathcal{P} \times \mathcal{N}\) since \(\text{SpecialVars}(N) \subseteq \text{SpecialVars}(P)\). (This is ensured by Line 8.)

We now prove soundness. Since \(P := \text{Join}_\mathcal{P}(P_1', P_2')\), it follows from the soundness of \(\text{Join}_\mathcal{P}\) that \(P_1' \preceq_\mathcal{P} P\) and \(P_2' \preceq_\mathcal{P} P\). Similarly, since \(N := \text{Join}_\mathcal{N}(N_1'', N_2'')\), it follows from the soundness of \(\text{Join}_\mathcal{N}\) that \(N_1'' \preceq_\mathcal{N} N\) and \(N_2'' \preceq_\mathcal{N} N\). It follows from the soundness of \(\text{Eliminate}_\mathcal{N}\) that \(N \preceq_\mathcal{N} N'\), and hence \(N_1'' \preceq_\mathcal{N} N'\) and \(N_2'' \preceq_\mathcal{N} N'\). It now follows from the definition of \(\preceq_{\text{Join}\mathcal{P}}\) that \((P_1, N_1) \preceq_{\text{Join}\mathcal{P}} (P, N')\) and \((P_2, N_2) \preceq_{\text{Join}\mathcal{P}} (P, N')\).

We now prove completeness. Suppose \((P_3, N_3)\) is such that \((P_1, N_1) \preceq_{\text{Join}\mathcal{P}} (P_3, N_3)\) and \((P_2, N_2) \preceq_{\text{Join}\mathcal{P}} (P_3, N_3)\) and \((P_3, N_3) \preceq_{\text{Join}\mathcal{P}} (P, N)\). Then, it follows from the definition of \(\preceq_{\text{Join}\mathcal{P}}\) and Lemma D.1.1 (Page 171) that \(P_1' \preceq_\mathcal{P} P_3' \wedge N_1'' \preceq_\mathcal{N} N_3, P_2' \preceq_\mathcal{P} P_3' \wedge N_2'' \preceq_\mathcal{N} N_3, \) and \(P_3' \preceq_\mathcal{P} P \wedge N_3'' \preceq_\mathcal{N} N, \) where \(N_1'' = P_2N(P_1', P_3, N_1'), N_2'' = P_2N(P_2', P_3, N_2'), N_3'' = P_2N(P_3', P, N_3'),\) and \((P_3', N_3') = \text{Saturate}(P_3, N_3).\) Since \(P := \text{Join}_\mathcal{P}(P_1', P_2')\), it follows from the completeness of \(\text{Join}_\mathcal{P}\) that \(P \preceq_\mathcal{P} P_3'\), and hence \(P \preceq_\mathcal{P} P_3\) (since clearly \(P_3' \preceq_\mathcal{P} P_3\)). Let \((P', N'') = \text{Saturate}(P, N')\). It now suffices to show that \(P_2N(P', P_3, N'') \preceq_\mathcal{N} N_3.\)

\[
\begin{align*}
P_2N(P', P_3, N) & = P_2N(P', P_3, \text{Join}_\mathcal{N}(N_1'', N_2'')) \\
& \preceq_\mathcal{N} \text{Join}_\mathcal{N}(P_2N(P', P_3, N_1'), P_2N(P', P_3, N_2')) & (7.1) \\
& = \text{Join}_\mathcal{N}(P_2N(P', P_3, P_2N(P', P, N_1')), P_2N(P', P_3, P_2N(P', P, N_2'))) \\
& \preceq_\mathcal{N} \text{Join}_\mathcal{N}(P_2N(P', P_3, N_1'), P_2N(P', P_3, N_2')) & (7.2) \\
& \preceq_\mathcal{N} \text{Join}_\mathcal{N}(N_3, N_3) = N_3 & (7.3)
\end{align*}
\]

We first explain why Eq. 7.1 holds. First, note that \(P'\) and \(P_3'\) refer to the same abstract element (as \(P_3' \preceq_\mathcal{P} P' \wedge P' \preceq_\mathcal{P} P_3'\)), and hence \(\text{BaseSets}(P') = \text{BaseSets}(P_3') = \text{BaseSets}(P_3).\) Let \(\sigma\) denote the bijective mapping from the special variables in \(\text{SpecialVars}(P_3)\) to those in \(\text{SpecialVars}(P')\) that correspond to the same base- set. For any numerical element \(N_4, P_2N(P', P_3, N_4)\) only adds the variable equalities to \(N_4\) that correspond to \(\sigma(v) = \sigma(u)\) for all \(u \in \text{SpecialVars}(P_3).\) Now, note that \(\text{Join}_\mathcal{N}(P_2N(P', P_3, N_1'), P_2N(P', P_3, N_2')) = \text{Join}_\mathcal{N}(N_1'', N_2'')\) since without loss of any generality, we can assume that the special variables in \(\text{SpecialVars}(P_3)\) do not occur in \(N_1''\) and \(N_2''\) (since we can always rename them). Hence, \(P_2N(P', P_3, \text{Join}_\mathcal{N}(N_1'', N_2'')) = P_2N(P', P_3, \text{Join}_\mathcal{N}(P_2N(P', P_3, N_1''), P_2N(P', P_3, N_2''))) \preceq_\mathcal{N} \text{Join}_\mathcal{N}(P_2N(P', P_3, N_1''), P_2N(P', P_3, N_2')).\) The last deduction above follows from the fact that for any numerical element \(N_5, \) we have \(\text{PostPredicate}_\mathcal{N}(N_5[x/y], x = y) \preceq_\mathcal{N} N_5\) (provided \(\text{PostPredicate}_\mathcal{N}\) is complete).

Eq. 7.2 follows from the following property of \(P_2N\) operator (which in turn is a corollary of Property 7.3.2 (Page 108)): For any set-domain elements \(P_4, P_5, P_6\) and any numerical element \(N, \) if \(P_4 \preceq_\mathcal{P} P_5 \preceq_\mathcal{P} P_6,\) then \(P_2N(P_5, P_6, P_2N(P_4, P_5, N)) \preceq_\mathcal{N} P_2N(P_4, P_6, N).\) Also, note that \(P_2N(P_1', P_1') = P_2N(P_1', P_1') \text{ and } P_2N(P_2', P_2') = P_2N(P_2', P_2')\) since \(\text{BaseSets}(P) = \text{BaseSets}(P).\)

Eq. 7.3 follows from the completeness of \(\text{Join}_\mathcal{N}\) operator.
7.4. ABSTRACT INTERPRETER FOR THE SET CARDINALITY DOMAIN

Now, observe that without loss of generality, we can assume that \( \text{SpecialVars}(N_1) \) and \( \text{SpecialVars}(N_2) \) are disjoint from \( \text{SpecialVars}(N_3) \) (since we can always rename the special variables that correspond to cardinalities of base-sets). It now follows from the completeness of \( \text{Eliminate}_N \) operator that \( \mathbb{P}2N(P, P_3, N') = N_3 \) since \( N' = \text{Eliminate}_N(N_2, V) \) and \( N_3 \) does not involve any variables in the set \( V \).

### 7.4.3 Assignment Node

The abstract element \( E' \) after an assignment node \( \ell := e \) (Figure 7.4(b)) is the strongest post-condition of the element \( E \) before the assignment node with respect to the assignment \( \ell := e \). It is computed by using an existential quantification operator \( \text{Eliminate}_D \) as described below.

\[
E' = \text{Eliminate}_D(E_1, x')
\]

where \( E_1 = \text{PostPredicate}_D(E[x'/x], x = e[x'/x]) \)

The post-predicate operator \( \text{PostPredicate}_D \) is defined in Section 7.4.4. The existential quantification operator \( \text{Eliminate}_D \) for any domain \( D \) takes as input an element \( E \) from \( D \) and an lvalue \( \ell \), and produces the least element that is above \( E \) and does not get affected by any change to \( \ell \).

**Definition 7.4.4 (Eliminate Operator \( \text{Eliminate}_D \))** Let

\[
E' = \text{Eliminate}_D(E, \ell).
\]

Then,

- **Soundness**: \( E \preceq_D E' \) and \( E' \) does not get affected by any change to \( \ell \).

- **Completeness**: If \( E'' \) is such that \( E \preceq_D E'' \) and \( E'' \) does not get affected by any change to \( \ell \), then \( E' \preceq_D E'' \).

Figure 7.6 shows how to implement the eliminate operator \( \text{Eliminate}_{\mathbb{P} \times \mathbb{N}} \) for the set cardinality domain \( \mathbb{P} \times \mathbb{N} \) using the eliminate operators \( \text{Eliminate}_\mathbb{P} \) and \( \text{Eliminate}_\mathbb{N} \) for the set and numerical domains in a modular fashion.

**Example 7.4.5** We demonstrate the implementation of the operator \( \text{Eliminate}_{\mathbb{P} \times \mathbb{N}} \) by the example in Figure 7.6(b). The example comes from appending two linked lists. The input is a list pointed to by \( x \) whose original length is \( n \) and a list pointed to by \( z \) whose length is \( k \). The second list has been appended to the first list. Now, we wish to existentially eliminate \( z \).

First, we eliminate \( z \) from the set domain, yielding \( P_1 \), a list pointed to by \( x \) losing the information on where \( z \) pointed to. Next, we use \( \mathbb{P}2N \) to express the cardinalities of the base-sets of \( P \) in terms of the cardinalities of the base-sets of \( P_1 \). In this case, \( A + B = C \), i.e., the sum of the lengths of the two parts of the list in \( P \) is equal to the length of the list in \( P_1 \). Next we eliminate \( z \) and variables corresponding to the original base-sets from the numerical element. This loses the original partition of the list and retains the important information that the length of the list is \( n + k \).

**Theorem 7.4.6** The Eliminate operator described in Figure 7.6(a) satisfies both the soundness and the completeness property stated in Definition 7.4.4 (provided the eliminate operator for the base domains, \( \text{Eliminate}_\mathbb{P} \) and \( \text{Eliminate}_\mathbb{N} \), satisfy these properties).
Proof: Consider the pseudo-code shown in Figure 7.6(a). Note that \((P_1, N_2)\) is an element of \(\mathcal{P} \bowtie \mathcal{N}\) since \(\text{SpecialVars}(N_2) \subseteq \text{SpecialVars}(P_1)\). (This is ensured by Line 5.)

We first prove soundness. It follows from the soundness of \(\text{Eliminate}_\mathcal{P}\) that \(P' \leq_\mathcal{P} P_1\) and change in \(\ell\) does not affect \(P_1\). It follows from the soundness of \(\text{Eliminate}_\mathcal{N}\) that \(N_1 \leq_\mathcal{N} N_2\) and change in \(\ell\) does not affect \(N_2\). Hence, change in \(\ell\) does not affect \((P_1, N_2)\), and \((P, N) \leq_{\mathcal{P} \bowtie \mathcal{N}} (P_1, N_2)\).

We now prove completeness. Let \((P', N'_2)\) be such that \((P, N) \leq_{\mathcal{P} \bowtie \mathcal{N}} (P'_1, N'_2)\) and \((P_1, N'_2)\) does not get affected by change in \(\ell\).

- From the above assumptions, we conclude that \(P'_1\) does not get affected by change in \(\ell\), and that \(P' \leq_\mathcal{P} P'_1\). It now follows from completeness of \(\text{Eliminate}_\mathcal{P}\) that \(P_1 \leq_\mathcal{P} P'_1\).

- From the above assumptions, we conclude that 
  \[
  \mathcal{P}2\mathcal{N}(P'_1, P'_1, N_1) = \mathcal{P}2\mathcal{N}(P_1, P'_1, \mathcal{P}2\mathcal{N}(P', P_1, N'))
  \leq_\mathcal{N} \mathcal{P}2\mathcal{N}(P', P'_1, N')
  \leq_\mathcal{N} N'_2
  \]
  (7.4)

Eq. 7.4 follows from the following property of \(\mathcal{P}2\mathcal{N}\) operator (which in turn is a corollary of Property 7.3.2 (Page 108)): For any set-domain elements \(P_4, P_5, P_6\) and any numerical element \(N\), if \(P_4 \leq_\mathcal{P} P_5 \leq_\mathcal{P} P_6\), then \(\mathcal{P}2\mathcal{N}(P_4, P_5, \mathcal{P}2\mathcal{N}(P_4, P_5, N)) \leq_\mathcal{N} \mathcal{P}2\mathcal{N}(P_4, P_5, N)\).

It follows from completeness of \(\text{Eliminate}_\mathcal{N}\) operator that \(\text{Eliminate}_\mathcal{N}(\mathcal{P}2\mathcal{N}(P_1, P'_1, N_1), V) \leq_\mathcal{N} \text{Eliminate}_\mathcal{N}(N'_2, V)\). Now, observe that without loss of generality, we can assume that \(\text{SpecialVars}(P)\) and \(\text{SpecialVars}(P'_1)\) are disjoint (since we can always rename the special variables that correspond to cardinalities of base-sets). Since \(P_1, P'_1, N'_2\) do not get affected by any change in \(\ell\) values in \(V\), it follows that \(\mathcal{P}2\mathcal{N}(P_1, P'_1, \text{Eliminate}_\mathcal{N}(N_1, V)) \leq_\mathcal{N} N'_2\), i.e., 

\[\mathcal{P}2\mathcal{N}(P_1, P'_1, N_2) \leq_\mathcal{N} N'_2\]

This implies that \((P_1, N_2) \leq_{\mathcal{P} \bowtie \mathcal{N}} (P'_1, N'_2)\).

### 7.4.4 Assume Node

The abstract element \(E'\) after an assume node \(\text{Assume}(\text{pred})\) (Figure 7.4(c)) is obtained by using the post-predicate operator described below.

\[E' = \text{PostPredicate}_\mathcal{D}(E, \text{pred})\]

The post-predicate operator \(\text{PostPredicate}_\mathcal{D}\) for a domain \(\mathcal{D}\) takes as input an abstract element \(E\) from domain \(\mathcal{D}\) and a predicate \(\text{pred}\) and returns the most precise abstract element \(E'\) such that \(\gamma_\mathcal{D}(E') \supseteq \gamma_\mathcal{D}(E) \cap \gamma(\text{pred})\), where \(\gamma\) is the concretization operation. The following makes this more precise.

**Definition 7.4.7 (Post-predicate Operator \(\text{PostPredicate}_\mathcal{D}\))**

Let \(E' = \text{PostPredicate}_\mathcal{D}(E, \text{pred})\). Let \(\gamma_\mathcal{D}\) denote the concretization function for domain \(\mathcal{D}\). Then,
Example 7.4.8 We demonstrate the PostPredicate\textsubscript{\(\mathcal{P}\bowtie\mathcal{N}\)} operator by the example in Figure 7.7(b). We return to the example of a list of length \(n\) pointed to by \(x\) to which a list of length \(k\) pointed to by \(z\) has been appended. We wish to assume the predicate \(m = [ls(x, \text{nil})]\), i.e., that \(m\) is the length of the entire list pointed to by \(x\). Note that the predicate refers to a base-set that \(P\) can interpret but is not in BaseSets\((P)\). The variable that represents the length of the list pointed to by \(x\) is \(C\). First we assume that \(m = C\). Next, \(\mathcal{P}\bowtie\mathcal{N}\) is used to interpret \(C\) in terms of the cardinalities of the base-sets in \(P\). In this case, \(C = A + B\), i.e., the sum of the lengths of the two parts of the list. Finally, we eliminate the special variables that do not correspond to the base-sets in BaseSets\((P)\). In this case, \(C\) is eliminated, retaining the important information that \(m = n + k\).

Theorem 7.4.9 The post-predicate operator described in Figure 7.7(a) satisfies the soundness property stated in Definition 7.4.4 (provided the post-predicate operators for the base domains, PostPredicate\textsubscript{\(\mathcal{P}\)} and PostPredicate\textsubscript{\(\mathcal{N}\)}, satisfy the same soundness property, and the eliminate operator for the numerical domain, Eliminate\textsubscript{\(\mathcal{N}\)}, satisfies the respective soundness property stated in Definition 7.4.4).

The proof of Theorem 7.4.9 follows simply from the observation that the concretization function for the combined domain is the intersection of the concretization functions of the set domain and the numerical domain. However, note that the post-predicate operator described in
Figure 7.7(a) does not necessarily satisfy the completeness property stated in Definition 7.4.7 because the pre-order for the combined domain \( \preceq_{P \times N} \) only accounts for a limited (not necessarily complete) sharing of information between the set domain and the numerical domain. In other words, our pre-order is not the best partial-order that corresponds to the concretization function for the combined domain.

### 7.4.5 Fixed-point Computation

In presence of loops, the abstract interpreter goes around each loop until a fixed point is reached. A fixed point is said to be reached when the abstract elements \( E_1, E_2 \) over domain \( D \) at any program point inside the loop in two successive iterations of that loop represent the same set of concrete elements, i.e., \( E_1 \preceq_D E_2 \) and \( E_2 \preceq_D E_1 \).

If the domains \( P \) or \( N \) have infinite chains, then fixed point for a loop may not be reached in a finite number of steps. In that case, a widening operation may be used to over-approximate the analysis results at loop heads.

A widening operator for a domain \( D \) takes as input two elements from \( D \) and produces an upper bound of those elements (which may not necessarily be the least upper bound). A widening operator has the property that it guarantees fixed point computation across loops terminates in a finite number of steps even for infinite height domains.

A widen operator \( \text{Widen}_D \) for a domain \( D \) takes as input two elements \( E_1 \) and \( E_2 \) from domain \( D \) and returns an element \( E \) with the following property:

**Definition 7.4.10 (Widening Operator \( \text{Widen}_D \))**

Let \( E = \text{Widen}_D(E_1, E_2) \). Then,

- **Soundness**: \( E_1 \preceq_D E_2 \) and \( E_2 \preceq_D E \).
- **Convergence**: The sequence of widen operations converges in a bounded number of steps, i.e., for any strictly increasing sequence \( E_0, E_1, \ldots \) (such that \( E_i \preceq_D E_{i+1} \) but \( E_{i+1} \not\preceq_D E_i \) for all \( i \)), if we define \( E_0' := E_0 \), \( E_1' := \text{Widen}_D(E_0', E_1) \), \( E_2' := \text{Widen}_D(E_1', E_2), \ldots \), then there exists \( i \geq 0 \) such that \( E'_j \preceq_D E'_i \) and \( E_i' \preceq_D E'_j \) for all \( j > i \).

Figure 7.8 shows how to implement the widen operator \( \text{Widen}_{P \times N} \) for the set cardinality domain \( P \times N \) using the widen operators \( \text{Widen}_P \) and \( \text{Widen}_N \) for the set and numerical domains in a modular fashion.

**Example 7.4.11** We demonstrate the \( \text{Widen}_{P \times N} \) operator by the example in Figure 7.8(b). This example is taken from a program that creates a list of length \( n \). After the first iteration, the list pointed to by \( x \) is a singleton and the loop counter \( i = 1 \). After the second iteration, the length of the list pointed to by \( x \) is between 1 and 2 and \( i \) equals the length of the list. Note that using Join, the constant 2 will keep increasing without ever converging. The \( \text{Widen} \) operator for the set domain is equivalent to Join for this set domain and returns a non-empty list pointed to by \( x \). The \( P2N \) operator expresses the cardinalities of the original base-sets in terms of this new list. Eliminating the special variables for the original base-sets yields numerical elements in which all the information is in terms of the new base-set. This elimination is
7.4. Abstract Interpreter for the Set Cardinality Domain

Widen\textsubscript{\mathcal{P}\mathcal{N}}((P_1, N_1), (P_2, N_2)) =
(P'_1, N'_1) := (P_1, N_1);
(P'_2, N'_2) := Saturate(P_2, N_2);
\begin{align*}
P & := \text{Widen}_\mathcal{P}(P'_1, P'_2); \\
N'' & := \mathcal{P}_2\mathcal{N}(P'_1, P, N'_1); \\
N'_2'' & := \mathcal{P}_2\mathcal{N}(P'_2, P, N'_2); \\
N'' & := \text{Eliminate}_\mathcal{N}(N''_1, \\
& \quad \text{SpecialVars}(N'_1) - \\
& \quad \text{SpecialVars}(P)); \\
N''_2 & := \text{Eliminate}_\mathcal{N}(N''_2, \\
& \quad \text{SpecialVars}(N'_2) - \\
& \quad \text{SpecialVars}(P)); \\
N & := \text{Widen}_\mathcal{N}(N''_1, N''_2); \\
\text{Output} & (P, N);
\end{align*}

(a) Algorithm

(\text{b) Example

\begin{align*}
\text{Inputs:} \\
P_1 & = [x \mapsto \text{nil}]^4 \\
N_1 & = A = 1 \land i = 1 \land i \leq n \\
P_2 & = [\text{ls}(x, \text{nil})]^B \\
N_2 & = B \geq 1 \land B \leq 2 \land i = B \land i \leq n
\end{align*}

\begin{align*}
\text{Trace of Widen}_\mathcal{P}\mathcal{N} & ((P_1, N_1), (P_2, N_2)); \\
P & = [\text{ls}(x, \text{nil})]^C \\
N_{i''} & = C = A \land B = 1 \land i = 1 \land i \leq n \\
N_{i''} & = C = B \land B \geq 1 \land B \leq 2 \land i = B \land i \leq n \\
N_{i''} & = C = 1 \land i = C \land i \leq n \\
N_{i''} & = C \geq 1 \land C \leq 2 \land i = C \land i \leq n \\
N & = C \geq 1 \land i = C \land i \leq n
\end{align*}

Figure 7.8: This figure describes the algorithm for widening for set cardinality domain $\mathcal{P} \times \mathcal{N}$ in terms of the widening algorithms for the domains $\mathcal{P}$ and $\mathcal{N}$ along with an example.

\textbf{Implies}_\mathcal{P}\mathcal{N}((P, N), \text{pred}) =
(P_1, N_1) := Saturate(P, N);
\text{if} \text{pred} \text{is an arithmetic predicate:}
\begin{align*}
V & := \text{SpecialVars} (\text{pred}) - \\
& \quad \text{SpecialVars}(P); \\
N_2 & := \mathcal{P}_2\mathcal{N}(P, V, N_1); \\
\text{return Implies}_\mathcal{N}(N_2, \text{pred});
\end{align*}
\text{else} // \text{pred is set predicate:}
\begin{align*}
\text{return Implies}_\mathcal{P}(P_1, \text{pred});
\end{align*}

(a) Algorithm

(\text{b) Example

\begin{align*}
\text{Inputs:} \\
P & = [\text{ls}(x, z)]^A \ast [\text{ls}(z, \text{nil})]^B \\
N & = A \geq 1 \land B \geq 1 \land A = n \land B = k \\
\text{pred} & = [[\text{ls}(x, \text{nil})]^C] = n + k
\end{align*}

\begin{align*}
\text{Trace of Implies}_\mathcal{P}\mathcal{N}((P, N), \text{pred}); \\
V & = \{C\} \\
N_2 & = A + B = C \land A \geq 1 \land B \geq 1 \\
& \quad A = n \land B = k
\end{align*}

Figure 7.9: This figure describes the algorithm for the implication checking transfer function $\text{Implies}_\mathcal{P}\mathcal{N}$ in terms of the implication checking transfer functions for the domains $\mathcal{P}$ and $\mathcal{N}$ along with an example.
necessary to ensure the widening operator precondition that $N''_2$ is weaker than $N''_1$. Finally, the numerical widen operator loses the possible range of the length of the list, but retains the important information that the length of the list equals the iteration number and that $i \leq n$, which will allow us to prove that after the loop terminates the length of the list is $n$.

The $\text{Widen}$ operation described in Figure 7.8 clearly satisfies the soundness property described above. It also satisfies the convergence property; as stated in the theorem below.

**Theorem 7.4.12** The $\text{Widen}$ operation described in Figure 7.8 satisfies the convergence property stated in Definition 7.4.10 (provided the $\text{Widen}$ operations for the base domains, $\text{Widen}_P$ and $\text{Widen}_N$, satisfy Definition 7.4.10).

**Proof:** Let $(P_1, N_1), (P_2, N_2), \ldots$, be any chain of elements in the set cardinality domain that arise at a given program point during successive loop iterations. Let $(Q_1, M_1) = (P_1, N_1)$ and $(Q_{i+1}, M_{i+1}) = \text{Widen}_{\mathbb{P} \times \mathbb{N}}((Q_i, M_i), (P_i, N_i))$. Then, according to Lemma D.2.4 in Appendix D.2, we have (1) $Q_i \preceq P_i$ for all $i < j$, and (2) if $Q_j \preceq P_i$ for some $i < j$, then $M_k \preceq M_{k+1} \sigma$ and $M_{k+1} \sigma \not\preceq M_k$ for all $i \leq k < j$ (otherwise the fixed-point computation converges), where $\sigma$ is some bijective variable renaming. Lemma D.2.1 (in Appendix D.2) bounds the number of times $Q_i$ can strictly increase. Lemma D.2.3 (in Appendix D.2) bounds the number of times $M_i$ can strictly increase in case $Q_i$ is stationary up to variable renaming. Hence, the length of the chain is bounded above by the product of the number of times $Q_i$ can strictly increase and the number of times $M_i$ can strictly increase up to variable renaming. Hence, the result.

### 7.4.6 Assert Node

After fixed-point has been reached, the results of the abstract interpreter can be used to validate given assertions in the program. Let $E$ be the abstract element computed by the abstract interpreter immediately before an assert node $\text{Assert}(\text{pred})$ after fixed-point computation. The assertion $\text{pred}$ can be validated by using the $\text{Implies}_P$ operator that takes as input an abstract element $E$ and a predicate $\text{pred}$ and checks whether or not $E$ implies $\text{pred}$.

Figure 7.9 shows how to implement the operator $\text{Implies}_{\mathbb{P} \times \mathbb{N}}$ in terms of the operators $\text{Implies}_P$ and $\text{Implies}_N$, along with an example.

### 7.5 A Case Study

The choice of which set analysis and which numerical analysis to combine depends on what data-structures and what properties of those data-structures we want to analyze. Different data-structures typically require different base-set constructors, while different operations on the same data-structure typically require different numerical domains. We illustrate this by two sets of examples.

Table 7.1 shows the loop invariants required to establish relationships between the size of the output data-structure and the size of the input data-structure for $\text{Copy}$ function. These examples require different set domains, but the same numerical domain, namely Karr’s linear equalities domain, works for all of these examples. The base-set constructors given here are
Table 7.1: This table describes the loop invariants (and hence illustrates the choice of the set analysis domain) required to analyze the copy routine of various data-structures. The property discovered is the relationship between the size of the output data-structure with the size of the input data-structure. (a) Acyclic List, (b) Cyclic List, (c) Tree, (d) n-ary Tree, (e) List of Lists, (f) List of Arrays (e.g., StringBuffer), (g) Array of Lists (e.g., Hashtable)
Table 7.2: This table describes the loop invariants (and hence illustrates the choice of the numerical analysis domain) required to analyze various routines of acyclic list data-structure. The property discovered is the relationship between the size of the output data-structure with the size of the input data-structure.

used to informally demonstrate the type of invariants the set domain should be able to represent. The exact way that these base-sets are defined changes according to the set domain used.

Table 7.2 shows the loop invariants required to establish relationships between the sizes of the output data-structure and the size of the input data-structure on various functions for an acyclic list (see Table 7.1 for the definition of $R(x,y)$). These examples require different numerical domains, but the same set analysis domain, namely one that provides a “reachability via next link” base-set constructor, works for all of these examples.

It is also interesting to note that in order to validate a given program, one can either choose a more precise set domain or a more precise numerical domain. For example, consider proving the data-structure invariant for the list copy example. The inductive invariant required to prove that the size of the copied list is the same as the size of the input list can either be expressed as $|R(x',\text{null})| = |R(x,y)|$ (see Copy example in Table 7.2) or $|R(x')| = |R(x)| - |R(y)|$ (see Acyclic List example in Table 7.1). The former is an element in the set cardinality domain built from a relatively more precise set domain (i.e., one that supports $R(x,y)$ as a base-set constructor as opposed to simply supporting $R(x)$), but a relatively less precise numerical domain (i.e., one that supports variable equalities, as opposed to arbitrary linear equalities).

The above observations are not supposed to imply that for each program, we need to work with a different combination of a set domain and a numerical domain. Certain set domains and certain numerical domains are more precise than several others; but precision comes at the cost of efficiency. Hence, we need to estimate the least precise set domain and the least precise numerical domain that would be good enough to reason about desired properties of desired programs.

### 7.5.1 Experimental Results

We have implemented an instance of the combination framework by combining the TVLA system [LAS00] with the Polyhedra abstract domain [CH78] as implemented by the PPL library [BHZ08] using some extra widening heuristics. We chose these domains as they can together handle all of the benchmarks described below. Using less precise domains it would be possible to prove some of these benchmarks with better efficiency. Table 7.3 summarizes the results of running the tool on a set of benchmarks. In all cases, the properties specified were proven without false alarms. The benchmarks were run on a 2.4GHz E6600 Core 2 Duo pro-
cessor with 2 GB of memory running Linux. For each program we give the time, the overhead factor over running TVLA without cardinality support, and the total number of abstract shape graphs generated in the analysis. Note that the overhead is rather low with an average of 60%. In some cases, the analysis with cardinality is even faster, as it can prune some of the search space using the more precise domain. In some of the examples, running without cardinality support yields memory safety false alarms.

The benchmarks are divided into the five categories detailed below.

**StringBuffer**  We analyzed the two most challenging methods from among the methods supported by the String Buffer class (as implemented in Microsoft product code): SBRemove (see Figure 7.1) and SBToString. SBToString converts a StringBuffer to a single string by allocating an array of the appropriate size and copying the characters in the correct order. We prove memory safety and data structure invariants on both examples. In SBRemove, we prove that the number of characters in the resulting StringBuffer is in sync with the number of characters removed. In SBToString, we prove that the size of the resulting string equals the number of characters in the original StringBuffer. These examples combine recursively defined data structures with arrays and demonstrate how the domain can track non-trivial relationships between the heap and the numerical variables in the programs.

**Termination**  We prove termination of two non-trivial examples: BubbleSort (see Figure 7.3) and Mark. The Mark example performs a DFS scan of a graph marking nodes as they are visited and using a stack for pending nodes. The scan terminates when the pending stack is empty. Proving termination in this case is non-trivial as the pending stack can grow as well as shrink in each iteration. Our technique is able to prove termination of this example by establishing the inductive invariant that the instrumented loop counter is bounded above by the cardinality of the set of nodes whose visited flag has been set to true.

**Linked List Examples**  This includes all examples in Table 7.2. We prove memory safety and data-structure invariants for all examples. In Reverse we prove that the length of the reversed list equals the length of the original list. In filter we prove that the length of the list is less or equal to the length of the original list. In Merge we prove that the length of the resulting list is the sum of lengths of the original lists. In MergeNoDups we prove that the length of the resulting list is less or equal to the sum of lengths of the original lists.

**Data Structure Copy**  This includes all examples in Table 7.1. These examples illustrate the power of our technique for proving bounds on memory allocation in terms of inputs. We prove that the size of the copied data-structure is equal to the size of the original input data-structure. Note that since a deep copy is performed, the relationship between the memory locations of the original data structure and the copied one cannot be expressed using set comparison operators (like set equality or set inclusion).

**JDK Collections Library**  We have used the tool to analyze most functions of the LinkedList and HashMap classes of JDK 5.0 [Ann]. The LinkedList class implements a circular doubly-linked list and the HashMap class implements an array of disjoint singly-linked lists. These functions, listed in Table 7.3, include the ones used to add a single element, add multiple
### Table 7.3: Experimental results for the set cardinality benchmarks

<table>
<thead>
<tr>
<th>Category</th>
<th>Program</th>
<th>Time (secs)</th>
<th>Overhead</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>String Buffer</td>
<td>SBRemove</td>
<td>295.21</td>
<td>2.83</td>
<td>50,615</td>
</tr>
<tr>
<td></td>
<td>SBTosString</td>
<td>79.53</td>
<td>3.15</td>
<td>10,176</td>
</tr>
<tr>
<td>Termination</td>
<td>BubbleSort</td>
<td>3.57</td>
<td>0.54</td>
<td>886</td>
</tr>
<tr>
<td></td>
<td>Mark</td>
<td>2.44</td>
<td>3.02</td>
<td>1,530</td>
</tr>
<tr>
<td>Linked List</td>
<td>Reverse</td>
<td>0.34</td>
<td>1.64</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>Filter</td>
<td>0.76</td>
<td>0.54</td>
<td>238</td>
</tr>
<tr>
<td></td>
<td>Merge</td>
<td>1.08</td>
<td>1.88</td>
<td>341</td>
</tr>
<tr>
<td></td>
<td>MergeNoDups</td>
<td>4.06</td>
<td>2.53</td>
<td>1,838</td>
</tr>
<tr>
<td>Data Structure</td>
<td>AcyclicListCopy</td>
<td>0.39</td>
<td>1.44</td>
<td>74</td>
</tr>
<tr>
<td>Copy</td>
<td>CyclicListCopy</td>
<td>4.54</td>
<td>1.20</td>
<td>155</td>
</tr>
<tr>
<td></td>
<td>TreeCopy</td>
<td>4.15</td>
<td>1.45</td>
<td>642</td>
</tr>
<tr>
<td></td>
<td>NaryTreeCopy</td>
<td>138.20</td>
<td>N/A</td>
<td>5,439</td>
</tr>
<tr>
<td></td>
<td>ListOfListsCopy</td>
<td>39.95</td>
<td>1.44</td>
<td>5,353</td>
</tr>
<tr>
<td></td>
<td>ListOfArraysCopy</td>
<td>12.67</td>
<td>1.02</td>
<td>2,260</td>
</tr>
<tr>
<td></td>
<td>ArrayOfListsCopy</td>
<td>7.99</td>
<td>0.30</td>
<td>1,628</td>
</tr>
<tr>
<td>JDK Collections</td>
<td>LLAdd</td>
<td>1.45</td>
<td>2.23</td>
<td>17</td>
</tr>
<tr>
<td>Library</td>
<td>LLAddAll</td>
<td>10.93</td>
<td>0.02</td>
<td>215</td>
</tr>
<tr>
<td></td>
<td>LLRemove</td>
<td>2.51</td>
<td>1.20</td>
<td>173</td>
</tr>
<tr>
<td></td>
<td>HMPut</td>
<td>9.45</td>
<td>1.02</td>
<td>3,132</td>
</tr>
<tr>
<td></td>
<td>HMPutAll</td>
<td>111.84</td>
<td>2.59</td>
<td>22,431</td>
</tr>
<tr>
<td></td>
<td>HMRemove</td>
<td>2.13</td>
<td>1.92</td>
<td>725</td>
</tr>
</tbody>
</table>
Chapter 8
Related Work

The related work discussion is divided by the parts and chapters of the thesis.

8.1 Using Automated Theorem Provers for Shape Analysis

8.1.1 Partial Axiomatization of Transitive Closure

**Shape Analysis.** This work was motivated by our experience with TVLA. The TVLA system is more automatic than the methods described in this chapter since it does not rely on user-supplied loop invariants. However, the techniques presented in this chapter are potentially more precise due to the use of full first-order reasoning. It can be shown that the **NoExit** scheme allows us to infer reachability at least as precisely as evaluation rules for 3-valued logic with Kleene semantics. In the future, we hope to develop an efficient non-interactive theorem prover that enjoys the benefits of both approaches. An interesting observation is that the colors needed in our examples to prove the formula are the same unary predicates used by TVLA to define its abstraction. This similarity may, in the future, help us find better ways to automatically instantiate the required axioms. In particular, inductive logic programming has recently been used to learn formulas to use in TVLA abstractions [LRS05], which holds out the possibility of applying similar methods to further automate the approach of this chapter.

**Decidable Logics.** Decidable logics can be employed to define properties of linked data structures: Weak monadic second-order logic has been used in [EMS00, MS01] to define properties of heap-allocated data structures, and to conduct Hoare-style verification using programmer-supplied loop invariants in the PALE system [MS01]. A decidable logic called $L_r$ (for “logic of reachability expressions”) was defined in [BRS99]. $L_r$ is rich enough to express the shape descriptors studied in [SRW98] and the path matrices introduced in [Hen90]. More recent decidable logics include Logic of Reachable Patterns [YRS+06] and a decision procedure for linked data structures that can handle singly linked lists [BR06].

This chapter does not develop decision procedures, but instead suggests methods that can be used in conjunction with existing theorem provers. Thus, the techniques are incomplete and the theorem provers need not terminate. However, our initial experience is that the extra flexibility gained by the use of first-order logic with transitive closure is promising. For example, we can prove the correctness of imperative destructive list-reversal specified in a natural way and the correctness of mark and sweep garbage collectors, which are beyond the scope of Mona and
Indeed, in [IRR+04b], we have tried to simulate existing data structures using decidable logics and realized that this can be tricky because the programmer may need to prove a specific simulation invariant for a given program. Giving an inaccurate simulation invariant causes the simulation to be unsound. One of the advantages of the technique described in this chapter is that soundness is guaranteed no matter which axioms are instantiated. Moreover, the simulation requirements are not necessarily expressible in the decidable logic.

Other First-Order Axiomatizations of Linked Data Structures. The closest approach to ours that we are aware of was taken by Nelson as we describe in Appendix 3.3. This also has some follow-up work by Leino and Joshi [Lei98]. Our impression from their write-up is that Leino and Joshi’s work can be pushed forward by using our coloring axioms.

A more recent work by Lahiri and Qadeer [LQ06] uses first-order axiomatization. This work can be seen as a specialization of ours to the case of (cyclic) singly linked lists.

Dynamic Maintenance of Transitive Closure. Another orthogonal but promising approach to transitive closure is to maintain reachability relations incrementally as we make unit changes in the data structure. It is known that in many cases, reachability can be maintained by first-order formulas [DS95, PI97] and even sometimes by quantifier-free formulas [Hes03]. Furthermore, in these cases, it is often possible to automatically derive the first-order update formulas using finite differencing [RSL03].

8.1.2 Labelled Clauses

For the general methodology of labelled clauses there is a huge literature related to our approach in the sense that the work can be reformulated and explored via a labelled-clause discipline. A discussion of this general aspect is beyond the scope of this chapter. Therefore, we concentrate on work related to our approach of proving multiple conjectures via labelled clauses.

Similar techniques for enhancing a theorem prover (for classical logic) to work with multiple goals have been developed by A. Voronkov, and are being incorporated into Vampire [Vor07]. However, his approach is based on an extension of splitting, as described in [RV01].

A logical candidate for proving conjectures in parallel is to use additional predicates [Gre69]. In our setting, this would mean replacing each negated conjecture \( \neg \varphi_i \) with the disjunction \( b_i \lor \neg \varphi_i \) where \( b_i \) is a fresh propositional variable (i.e., nullary relation). Now if the prover deduces the unit clause \( b_i \), the conjecture \( \varphi_i \) is valid. The main problem with this approach is that it will also explore disjunctions between different \( b_i \)'s, i.e., between different conjectures, which unnecessarily increases the search space by an exponential factor. Furthermore, the literals added to the clauses block any reductions from (derived) conjecture clauses in the axioms’ clauses.

Symbolic decision procedures [LBC05] are a technique for finding all possible disjunctions of conjectures for a given axiom set. The technique is limited to specific theories, such as uninterpreted functions and difference logic. Furthermore, because it is described in the context of the Nelson-Oppen method [NO78] for combining decision procedures, its usability in the case of quantifiers is limited.

In the Boolean satisfiability community there is a related idea of incremental solvers (see e.g., [WKS01]). There, it is possible to add and remove clauses from the theorem prover without restarting it. On the other hand, labelled clauses allow us to attempt to prove multiple
conjectures simultaneously.

## 8.2 Specialized Shape Analysis

### 8.2.1 Specialized Shape Analysis for Commonly Used Data Structures

Shape and heap analysis is a subject of active research with many interesting algorithms including [JM81, SRW02, LYY05]. The TVLA system generalizes these algorithms and can be utilized to implement our algorithm. Indeed, in this chapter we followed the line of research similar to the one in [Hen90, LYY05, LQ06, MYRS05] of developing a specialized shape analysis for commonly used data structures. We are very pleased with the ability of our method to compute the most precise abstract transformers in an efficient way. In contrast, TVLA can spend a lot of time in order to determine if an abstract state is feasible. Indeed it can spend an exponential time even when there are no resultant abstract states. The abstraction in this chapter is tailored for an interesting set of properties. A mechanism to support other properties (such as TVLA’s Instrumentation Predicates) remains an interesting open problem.

Connection analysis [GH98] keeps reachability information between program variables. Our work is more precise as it can perform strong updates for heap manipulation. Grammar based abstraction [LYY05] uses a restricted grammar to annotate summary nodes with their possible shapes. The abstractions are incomparable since the grammar based abstraction can express invariants (such as binomial heap) that cannot be expressed in our abstraction. On the other hand, the grammar based abstraction can deal with only a limited amount of sharing. For example, it cannot represent a tree with parent pointers and a pointer arbitrarily deep into the tree.

The shape analysis of [DOY06] is very similar to [MYRS05] both in the properties of the abstraction and in the programs handled.

### Decision Procedures for Linked Data Structures

An orthogonal line of research is the development of decision procedures and theorem provers which support transitive closure [BPZ05, IRR+04b, LAIR+05, BR05]. Such techniques can be utilized with arbitrary abstractions.

In this chapter, we developed direct methods for a specific abstraction. We are encouraged by the fact that our asymptotic complexity is lower than the above mentioned procedures by orders of magnitudes. Moreover, our implementation is also faster by a factor of 100 than the one reported in [BR05]. The MONA System [HJJ+95] can be used to implement the operations in this chapter. However, it has non-elementary complexity and is in our experience infeasible for program with trees.

### 8.2.2 Constructing Specialized Shape Analyses for Uniform Change

#### Specialized Shape Analyzers

Developing specialized shape analysis for commonly used data structures is an active line of research [Hen90, MYRS05, LAIS06, DOY06]. We are encouraged by the fact that we are able

---

1Our method also allows trees which are beyond the scope of [BR05].
to express all of the above-cited work using our methodology. Moreover, our methodology supports shared trees and the addition of arbitrary colors, which are beyond the scope of existing methods. It should be noted that our current algorithms are more costly. In particular, the ad-hoc algorithm in [LAIS06] runs in time essentially linear in the output, which is hard to beat. In the future, we plan to reduce the costs of creating the transformers by: (i) focusing only the necessary parts, (ii) developing more efficient focus algorithms, and (iii) using incrementality to reduce the cost of feasibility checks.

The TVLA System

The results in this chapter are inspired by the TVLA system. The TVLA system does not require that update formulas be monadic-uniform. It also allows arbitrary classes of graphs to be used. Also, [RSL03] includes an algorithm for automatically generating update formulas for auxiliary information, which is fully integrated into the system. (§5.4.1 of [Log06] describes the application of that machinery for an abstraction similar to the one described for cyclic singly-linked lists.) However, the TVLA system does not guarantee that the transformers are the most precise abstract transformers. Moreover, the system can issue a runtime exception in certain cases when an operation may lead to an infinite number of structures. In this chapter, we build specialized shape analyses that can handle many of cases for which TVLA was used. For most of these cases, we can now compute the most precise abstract transformer. In the future, it may be possible to combine methods like the ones in [RSL03] with our method. For example, there may be a way to generate monadic-uniform update formulas in certain cases.

The focus operation in TVLA differs from the one in this chapter in several key aspects including: (i) it requires the user to specify which formulas to focus on, and (ii) it may yield an infinite number of structures. In contrast, in this chapter we show that for every monadic-uniform update, there is a computable set of focused structures that lead to most precise abstract transformers. Our results also shed light on the cases when the updates in TVLA are precise.

Procedures and Libraries

In this chapter, we focused on handling programs without procedures and libraries. It is possible to handle procedures and libraries by tabulation of input/output relations between abstract values (e.g., see [CC78,RSY05]). It may be also possible to handle specific libraries by allowing monadic-uniform specifications of auxiliary relations that describe an abstraction of the effect on the client module.

Employing Theorem Provers and Decision Procedures

Theorem provers and decision procedures can be employed to prove properties of programs that manipulate the heap (e.g., see [Nel83,MS01,LQ06,LAIR+05,RBH06]). Moreover, they can be used to fully automate the process of generating transformers (e.g., see [BR01,HJMS03,RSY04,YRS04]).

Results from dynamic descriptive complexity and the methodology of this chapter improve the aforementioned results in various ways. For instance, in contrast to the method of Lahiri and Qadeer [LQ06], which requires user intervention, our method handles programs that manipulate cyclic lists in a totally automatic way.
In essence, the introduction of transformers that use only monadic-uniform update formulas can be seen as a way to replace a characterization of mutations of data structures with a characterization in terms of invariants. That is, two-vocabulary structures (which describe the state before and after the transition) are a natural way to express mutations, whereas standard one-vocabulary structures express invariants. In some cases, the switch from two-vocabulary to one-vocabulary structures results in an order-of-magnitude complexity improvement. In other cases, where decision procedures are not known for—or known not to exist for—two-vocabulary structures, the reduction to one-vocabulary structures restores the possibility of employing decision procedures:

- With two-vocabulary structures, it is easy to see that monadic second-order logic is undecidable even on linked lists. (The intuitive reason is that two functions, plus a few unary relations, can be used to encode a grid.) However, monadic second-order logic on trees is decidable \[Rab69\], and thus can be used to perform the feasibility checks on one-vocabulary structures that are needed when our method is employed.

- Rakamaric et al. \[RBH06\] gave a complete decision procedure for checking feasibility of a given (one-vocabulary) abstract state, but left open the question of how to handle transformers in the most-precise way. Our methodology solves this problem: the DynQF updates for singly linked lists of Hesse \[Hes03\] can be used to recast the problematic transformers using only one-vocabulary formulas, and hence the most precise abstract transformer is computable as explained in Section 1.2.2.

8.3 Combining Shape Analysis Domains

8.3.1 A Combination Framework for Tracking Partition Sizes

Combining Abstractions

The seminal paper by Cousot and Cousot in \[CC79\] introduces different methods for combining abstract domains including reduced product, which can be used to explain our domain construction (see \[Deu90\] for further elaboration on domain constructors). However, the problem of developing an effective procedure for computing abstract transformers for reduced products has been addressed only in specific settings. Gulwani and Tiwari gave algorithms for constructing the transfer functions for reduced products for a special case of abstract domains (called logical abstract domains) with the further restriction that the abstract domains being combined should be over convex theories with disjoint signatures \[GT06\]. Their methodology is not applicable in our setting since the abstract domains that we consider in this chapter, namely set domains and numerical domains, do not fit the required restrictions: the set domain is not convex, and furthermore, the set domain and the numerical domain both share the cardinality function symbol. Our work thus extends the line of work on constructive synthesis of abstract transformers for reduced product domains (from the abstract transformers of individual domains) for an important class of domains.

*:Combining Heap and Numerical Abstractions

The idea to combine numeric and pointer analysis for establishing properties of memory was pioneered by Alain Deutsch \[Deu92,Deu94\]. Deutsch’s abstraction deals with may-aliases
in a rather precise way but loses most of the information when the program performs destructive memory updates.

In [HP99] a type and effect system is suggested for a variant of ML that allows to bound the size of memory used by the program with applications to embedded code. There, the type system allows verifying bounds on memory usage while our analysis can be used to infer the bound. Furthermore, their type system is for a functional language while our analysis is appropriate for an imperative language with destructive pointer updates.

In [HJ03] linear typing and linear programming based inference are used to statically infer linear bounds on heap space usage of first-order functional programs running under a special memory mechanism. In contrast, our method handles imperative programs which use destructive updates.

In [YKB02] an algorithm for inferring sizes of singly-linked lists was presented. This algorithm uses the fact that the number of uninterrupted list segments in singly-linked lists is bounded. This limits the applicability of the method for showing specific properties of singly-linked lists. Similar restrictions apply to [BBH+06, MBCC07].

A general method for combining numeric domains and canonical abstraction was presented in [GDD+04]. Their method is orthogonal to ours, as it addresses the problem of abstracting values of numerical fields. On the other hand, our work is concerned with cardinalities of memory partitions. Combining the methods can be very useful and is the subject of future work.

Rugina [Rug04] presents a static analysis that can infer quantitative properties (namely height and skewness) of tree-like heaps. Rugina does not address the issue of sizes of data structures and is limited to tree-like heaps. On the other hand, Rugina can handle properties such as height, which are beyond the scope of this chapter.

In [CDOY06] a method is presented for analyzing a memory allocator by interpreting memory segments as both raw buffers and structured data. However, their method presents a limited way of treating sizes of chunks of memory since they are limited to contiguous chunks of memory and cannot handle sizes of recursive data structures.

In [GRS05], a specialized canonical abstraction was applied to analyze properties of arrays. Arrays are partitioned into the parts before, at, and after a given index. This gives a way to track sizes of specific partitions. However, it does so only in the special case of arrays. Furthermore, it cannot track sizes of partitions other than the ones formed by index variables. Specifically, their method would not be able to handle examples such as StringBuffer remove.

Reducing Pointer Programs to Integer Programs

In [DRS03, BBH+06, MBCC07] it was proposed to conduct pointer analysis in a pre-pass and then to convert the program into an integer program to allow integer analysis to check the desired properties. This “reduction based approach” allows using different integer analyzers on the resulting program. Furthermore, for proving simple properties of singly-linked lists it was shown in [BBH+06], that there is no loss of precision. However, it may lose precision in cases where the heap and numerics interact in complicated ways. Also, the reduction may be too expensive. Our transformers avoid these issues by iterating between the two abstractions and allowing information flow in both directions. Furthermore, our framework allows for an arbitrary set domain (it is not restricted to domains that can represent only singly-linked lists). Finally, proving soundness in our case is simpler.
8.3. Combining Shape Analysis Domains

Decision Procedures for Reasoning about Heap and Arithmetic

One of the challenging problems in the area of theorem proving and decision procedures is to develop methods for reasoning about arithmetic and quantification.

In [KR07] an algorithm for combining Boolean algebra and quantifier free Presburger arithmetic is presented. Their approach presents a complete decision procedure for their specific combined domain. In contrast, our method supports set domains that go beyond Boolean algebra formulas and can thus express more complicated invariants. More significantly our approach provides an effective method for computing transformers for performing abstract interpretation, which their method does not. Fortunately, by careful design of the interface between the abstract domains, we avoid solving the complex constraints which their algorithm handles.

In [NDQC07] a logic based approach that involves providing an entailment procedure is presented. Their logic allows for user-defined well-founded inductive predicates for expressing shape and size properties of data-structures. They can express invariants that involve other numeric properties of data structures such as height of trees. However, their approach is limited to separation logic while ours can be used in a more general context. In addition their approach does not infer invariants, requiring a heavy annotation burden, while our approach is based on abstract interpretation and can thus infer loop and recursive invariants.

Static Heap Space Usage Analysis

Several researchers have studied the problem of estimating heap memory consumption for functional languages [HJ03, HP99, USL03] and imperative object-oriented languages [CKQ05, CNQR05, Ghe02, BGY06]. Hughes and Pareto proposed a type and effect system on space usage estimation based on the notion of sized types [HP99] for a variant of ML such that well typed programs are proven to execute within the given memory bounds. Hofmann and Jost statically infer linear bounds on heap space usage of first-order functional programs running under a special memory mechanism [HJ03]. They use linear typing and an inference method through linear programming to derive these bounds. Their linear programming technique requires no fix-point analysis but it restricts the memory effects to a linear form without disjunction. Unnikrishnan et al. construct a new function that symbolically models the memory allocations of the original [USL03]. This function is executed over a valuation of parameters to obtain memory bounds but this process may not terminate, even when the original program does.

For imperative object-oriented languages, Chin et al. propose a type-based system similar to [HP99] that statically verifies programmer-specified size annotations expressed as Presburger formulas [CKQ05, CNQR05]. It does not synthesize memory bounds but instead checks the size annotations. Gheorghioiu manipulates symbolic arithmetic expressions on unknowns that are not necessarily program inputs to estimate upper bounds on heap usage [Ghe02]. Since some of the unknowns are not program inputs it is unclear how these are instantiated to obtain upper bounds. Braberman et al. automatically synthesize parametric non-linear over-approximations of memory consumption for each method of Java-like imperative object-oriented programs [BGY06]. They do not handle recursion and use Daikon [ECGN99], a dynamic tool that discovers likely invariants, to generate local invariants. The use of Daikon renders their method unsound.
Chapter 9

Conclusion

Shape analysis continues to be a challenging and actively studied problem in program analysis. In this thesis we explore techniques that can take advantage of existing tools and program analysis algorithms such as automated theorem provers and numerical abstract domains and use them to generate more powerful shape analysis algorithms. This approach of building on existing tools and domains is very powerful as it allows us to take advantage of further developments in this active field.

9.1 Using Automated Theorem Provers for Shape Analysis

Automated theorem provers have advanced significantly in the last decade. Using them effectively for shape analysis is challenging because of the complexity of logics involved and the number of calls required to the theorem prover.

9.1.1 Partial Axiomatization of Transitive Closure

Chapter 3 reports on our proposal of a novel methodology for using off-the-shelf first-order theorem provers to reason about reachability in programs. We have explored many of the theoretical issues as well as presenting examples that, while still preliminary, suggest that this is indeed a viable approach.

As mentioned earlier, proving the absence of paths is the difficult part of proving formulas with TC. The promise of our approach is that it is able to handle such formulas effectively and reasonably automatically, as shown by the fact that it can successfully handle the programs described in Section 3.4 and the success of the TVLA system, which uses similar transitive-closure reasoning. Future research directions include:

- Exploring other heuristics for identifying color classes.
- Exploring variations of the algorithm given in Figure 3.5 for instantiating coloring axioms.
- Exploring the use of additional axiom schemes, such as two of the schemes from [Nel83], which are likely to be useful when dealing with predicates that are partial functions. Such predicates arise in programs that manipulate singly-linked or doubly-linked lists—or,
more generally, data structures that are acyclic in one or more “dimensions” \cite{HHN92} (i.e., in which the iterated application of a given field selector can never return to a previously visited node).

- Recent advancements in interpolation for first-order theorem provers \cite{McM08} use proofs on first-order formulas to discover possible components of the inductive invariants. Exploring the interaction between interpolation and the techniques of this chapter is promising.

### 9.1.2 Labelled Clauses

In Chapter 4, we suggested the methodology of labelled clauses for the study and implementation of superposition-based classical first-order-logic calculi. We believe that using labelled clauses as an extension of ordinary ones will become as fruitful for advances in automated theorem proving as it is in the context of non-classical logics.

Our work on labelled clauses offers new possibilities to study saturation and tableau-like calculi in a common framework. This might also be fruitful for advances in theorem-proving search strategies, as, e.g., suggested by Bonacina \cite{Bon05}.

We have shown how to instantiate the general framework of labelled clauses for several interesting cases, including clause splitting and slicing. For the case of proving multiple conjectures simultaneously, we have also implemented the calculus as an extension of the SPASS theorem prover and report on convincing experimental results in the context of software verification.

We believe that there are other techniques in the world of theorem proving that can benefit from the idea of labelled clauses. Investigating them is the subject of future work. In particular, the combination of theorem proving techniques represented in the labelling methodology simplifies to the combination of the labelling disciplines.

### 9.2 Specialized Shape Analysis

Specializing the shape analysis algorithms for given data structures and programs allows us to ensure predictability of the shape analysis by computing the most precise abstract transformers.

#### 9.2.1 Specialized Shape Analysis for Commonly Used Data Structures

In Chapter 5, we have simplified the task of shape analysis for sequential procedure-less programs which create limited data structures. It will be interesting to extend our method to handle procedures and concurrency.

Another area for further research is the development of methods for integrating our abstractions with other abstractions besides Booleans. Methods in the spirit of Nelson-Oppen such as \cite{GT06} can be useful.

#### 9.2.2 Constructing Specialized Shape Analyses for Uniform Change

In Chapter 6, we make an interesting connection between dynamic descriptive complexity and abstract interpretation. We believe that exploring this connection will benefit both communi-
ties. On the pragmatic side, developing specialized algorithms for abstract interpretation, that can establish a limited set of properties for programs that use a specified set of language features, is one of the most promising directions in (fully automatic) program verification. The methodology presented in this chapter can be used to facilitate the creation of such algorithms.

9.3 Combining Shape Analysis Domains

In Chapter 7, we have shown the usefulness of reasoning about sizes of data structures. Adding support for numeric fields (e.g., based on [GDD+04]) can greatly increase the applicability of this technique. Exploring which shape analysis domains can be adapted to support the interface required to be used as a set domain will enable using this technique with other domains.

The use of a combination framework, which allows the use of existing domains, helps taking advantage of the large body of research in program analysis. Another promising direction is the design of standard interfaces for shape analysis domains as we have done in this chapter. Standard interfaces will allow the use of the abstract domain most suitable for the problem at hand. Another example of standard interface for shape analysis in the context of concurrency was developed by Gotsman et al. [GBCS07].
Bibliography


Appendix A

Further Examples for Chapter 3

A.1 Further Examples

This section shows the code (Figure A.1) and the complete specification of two additional examples: appending two linked lists, and the mark phase of a simple mark and sweep garbage collector.

Specification of append

The specification of append (see Figure A.1(a)) is given in Figure A.2. The specification includes procedure’s pre-condition, a transformer of the procedure’s body effect, and the procedure’s post-condition. The pre-condition (Figure A.2(a)) states that the lists pointed to by $x$ and $y$ are acyclic, unshared and disjoint. It also states there is no garbage. The post condition (Figure A.2(b)) states that after the procedure’s execution, the list pointed to by $x'$ is exactly the union of the lists pointed to by $x$ and $y$. Also, the list is still acyclic and unshared. The transformer is given in Figure A.2(c). The result of the loop in the procedure’s body is summarized as a formula defining the $last$ variable. The only change to $n$ is the addition of an edge between $last$ and $y$.

The coloring axioms needed to prove append are given in Figure A.3.

Specification of the mark phase

Another example proven is the mark phase of a mark-and-sweep sequential garbage collector, shown in Figure A.1(b). The example goes beyond the reverse example in that it manipulates a general graph and not just a linked list. Furthermore, as far as we know, ESC/Java [FLL+02] was not able prove its correctness because it could not show that unreachable elements were not marked. Note that the axiom needed to prove this property is NoExit, which we have shown to be beyond the power of Nelson’s axiomatization.

The loop invariant of mark is given in Figure A.4(a). The first disjunct of the formula holds only in the first iteration, when only the nodes in root are pending and nothing is marked. The second holds from the second iteration on. Here, the nodes in root are marked or pending (they start as pending, and the only way to stop being pending is to become marked). No node is both marked and pending (because the procedure checks if the node is marked before adding it to pending). All nodes that are marked or pending are reachable from the root set (we start
Node append(Node x, Node y) {
    [0] Node last = x;
    [1] if (last == null)
    [2]     return y;
    [3] while (last.next != null) {
    [4]     last = last.next;
    [5] }
    [6] last.next = y;
    [7] return x;
}

void mark(NodeSet root, NodeSet marked) {
    [0] Node x;
    [1] if(!root.isEmpty()){
    [3]     pending.addAll(root);
    [4]     marked.clear();
    [5]     while (!pending.isEmpty()) {
    [6]         x = pending.selectAndRemove();
    [7]         marked.add(x);
    [8]         if (x.car != null &&
    [9]             !marked.contains(x.car))
    [10]        pending.add(x.car);
    [11]         if (x.cdr != null &&
    [12]             !marked.contains(x.cdr))
    [13]        pending.add(x.cdr);
    [14]     }
    [15] }
}

Figure A.1: A simple Java-like implementation of (a) the concatenation procedure for two singly-linked lists; (b) the mark phase of a mark-and-sweep garbage collector.
A.1. Further Examples

(a) \[\text{pre} \overset{\text{def}}{=} \text{acyclic}[n] \land \text{unshared}[n] \land \text{unique}[x] \land \text{unique}[y] \land \text{func}[n] \land (\forall v. \lnot r_{x,n}(v) \lor \lnot r_{y,n}(v)) \land \forall v. r_{x,n}(v) \lor r_{y,n}(v) \] (A.1)

(b) \[\text{post} \overset{\text{def}}{=} \text{acyclic}[n'] \land \text{unshared}[n'] \land \text{unique}[x'] \land \text{unique}[\text{last}] \land \text{func}[n'] \land (\forall v. r_{x',n'}(v) \leftrightarrow (r_{x,n}(v) \lor r_{y,n}(v))) \land \forall v_1, v_2. n'(v_1, v_2) \leftrightarrow n(v_1, v_2) \lor (\text{last}(v_1) \land y(v_2)) \] (A.2)

T is the conjunction of the following formulas:

(c) \[\forall v. x'(v) \leftrightarrow x(v) \] (A.3)
\[\forall v. \text{last}(v) \leftrightarrow r_{x,n}(v) \land \forall u. \lnot n(v, u) \] (A.4)
\[\exists v. \text{last}(v) \] (A.5)
\[\forall v_1, v_2. n'(v_1, v_2) \leftrightarrow n(v_1, v_2) \lor (\text{last}(v_1) \land y(v_2)) \] (A.6)

Figure A.2: Example specification of append procedure: (a) precondition pre, (b) postcondition post, (c) transformer T (effect of the procedure body).

Figure A.3: The instances of coloring axioms used in proving append.
with only the root nodes as pending, and after that only nodes that are neighbors of pending nodes became pending; furthermore, only pending nodes may become marked). There are no edges between marked nodes and nodes that are neither marked nor pending (because when we mark a node we add all its neighbors to pending, unless they are marked already). Our method succeeded in proving the loop invariant in Figure A.4(a) using only the positive axioms.

The post-condition of mark is given in Figure A.4(b). To prove it, we had to use the fact that there are no edges between marked and unmarked nodes (i.e., there are no pending nodes at the end of the loop). Thus, we instantiate the axiom \texttt{NoExit}[marked, f], and this is enough to prove the post-condition.
Appendix B

Proofs for Chapter 5

B.1 Abstraction

We denote the set of nodes that can be contracted by \( B(S, D) \) as

\[
J(S, D) \overset{\text{def}}{=} U^S - ((D - sm(S)) \cup \text{distinct}(S))
\]

This is an extension of the original definition to handle cases in which the original state is not concrete. We now define a valid contraction:

\[
\text{contract}(S_1, S_2, a, b, f, D) \iff a \in J(S_1, D) \land a \neq b \land b \in J(S_1, D) \\
\land (a, b) \in ref^{S_1}(f) \land S_2 = \text{contract}(S_1, a, b)
\]

We define a derivation relation \( S_1 \xrightarrow{D} S_2 \) stating that a single contraction has been made on an edge not incident to nodes in \( D \)

\[
S_1 \xrightarrow{D} S_2 \iff \bigvee_{f \in \mathcal{P} \text{Ref}} \exists a, b. \text{contract}(S_1, S_2, a, b, f, D)
\]

Lemma B.1.1 \( \xleftarrow{D} \) is a reduction order:

(Confluent) \( S \xleftarrow{D_1} S_1 \land S \xleftarrow{D_2} S_2 \Rightarrow S_1 = S_2 \lor \exists S'. S_1 \xleftarrow{D_2} S' \land S_2 \xleftarrow{D_1} S' \)

(Terminating) \( \xleftarrow{D} \) has no infinite chains

Proof:

(Local Confluent) Let \( S_1 = \text{contract}(S, a_1, b_1) \) and \( S_2 = \text{contract}(S, a_2, b_2) \). Thus, \( a_1, b_1 \in J(S, D_1) \) and \( a_2, b_2 \in J(S, D_2) \). We consider the possible cases, depicted in Figure B.1:

- (a) \( a_1 = a_2 \land b_1 = b_2 \): Here \( S_1 = S_2 \).

- (b) \( a_1 = b_2 \land b_1 = a_2 \): Here \( S_1 \) and \( S_2 \) are isomorphic as the only difference between them is the name of the resulting node \( a_1 \) in \( S_1 \) and \( a_2 \) in \( S_2 \). Thus, \( S_1 = S_2 \).

- (c) \( a_1, a_2, b_1, b_2 \) are disjoint: It is easy to see that \( \text{contract}(S_1, a_2, b_2) = \text{contract}(S_2, a_1, b_1) \) and that \( a_1, b_1 \in J(S_2, D_1) \) and \( a_2, b_2 \in J(S_1, D_2) \)

- (d) \( a_1 = a_2 \land b_1 \neq b_2 \): Similar to (c)
(Confluent) Simple extension of the Diamond Lemma [New42].

(Terminating) By definition of contract the universe of the result is strictly smaller than the universe of the input.

Since $\rightarrow_D$ is a reduction order, each chain has a maximal element and we define $B$ formally as,

$$B(S, D) = S' \iff S \rightarrow_D S' \land \lnot \exists S'' . S' \rightarrow_D S''$$

Let $M_{S,S'}$ denote the embedding function from $S$ to $S'$. For ease of presentation we define $M_{S,S} \equiv \text{id}$. Let $\text{sm}(S, S') \equiv \{ n \mid |M_{S,S'}^{-1}(n)| > 1 \}$, and $\text{self}(S) \equiv \{ n \mid (n,n) \in f^{ld_S} \}$. Proposition B.1.2 states some properties of $M$ and Proposition B.1.3 state some properties of the contraction.

**Proposition B.1.2 (Properties of $M$)**

1. If $S' = \text{contract}(S, a, b)$ then $M_{S,S'} = up_{b \to a}$
2. If $S \rightarrow_D S' \land S' \rightarrow_D S''$ then $M_{S,S''} = M_{S',S''} \circ M_{S,S'}$
3. If $S \rightarrow_D S' \land S' \rightarrow_D S''$ then $\text{sm}(S, S'') = M_{S',S''}(\text{sm}(S, S')) \cup \text{sm}(S', S'')$

Proof: Trivial.

**Proposition B.1.3** Let $S \rightarrow_D S'$, the following statements hold:

1. $(n_1', n_2') \in \text{ref}^{S'}(f) \iff \exists n_1 \in M_{S,S'}^{-1}(n_1') . n_2 \in M_{S,S'}^{-1}(n_2') . (n_1, n_2) \in \text{ref}^S(f)$
2. $\text{self}(S') = M_{S,S'}(\text{self}(S)) \cup \text{sm}(S, S')$
3. If $S'$ has no garbage, then $S$ has no garbage

Proof:

1. Trivial
2. Trivial
3. To see that $\rightarrow_D$ preserves garbage note that the contraction is on an edge, thus any path in $S$ is also a path in $S'$, it can only be shorter.
Definition B.1.4 \( n_1, ..., n_k \) is an undirected path in a state \( S \) if

\[
\forall 1 \leq i < k . (n_i, n_{i+1}) \in \text{fld}^S \lor (n_{i+1}, n_i) \in \text{fld}^S
\]

A simple undirected path \( n_1, ..., n_k \) is an undirected cycle of length \( k \) in a state \( S \) if \( n_1 = n_k \) and one of the following conditions hold

1. (SelfLoop) \( k = 2 \)
2. (Directed3Cycle) \( k = 3 \) and \( (n_1, n_2) \in \text{fld}^S \land (n_2, n_1) \in \text{fld}^S \)
3. (ParallelEdges) \( k = 3 \) and \( (n_1, n_2) \in \text{ref}^S(f_1) \land (n_1, n_2) \in \text{ref}^S(f_2) \land f_1 \neq f_2 \)
4. (Undirected4Cycle) \( k > 3 \)

We say that \( S \) is loop-free admissible if removing all self-loops from \( S \) makes it admissible.

Lemma B.1.5 Let \( S \xrightarrow{d} S' \) s.t. \( S \) is loop-free admissible and \( \text{self}(S) \cap \text{distinct}(S) = \emptyset \).

1. \( \text{shared}(S) = \text{shared}(S') \)
2. \( \text{distinct}(S) = \text{distinct}(S') \)
3. \( \text{self}(S') \cap \text{distinct}(S') = \emptyset \)
4. \( S' \) is loop-free admissible
5. If \( n_1 \neq n_2 \) then

\[
(n_1', n_2') \in \text{ref}^S(f) \Leftrightarrow \exists n_1 \in M_{S,S'}^{-1}(n_1'), n_2 \in M_{S,S'}^{-1}(n_2') . (n_1, n_2) \in \text{ref}^S(f)
\]

Proof:

Let \( S' = \text{contract}(S) \). From Definition B.1.4, we need to show that we never merge two non-adjacent nodes in a state \( S \) if \( n_1 = n_k \) and one of the following conditions hold

1. (SelfLoop) \( k = 2 \)
2. (Directed3Cycle) \( k = 3 \) and \( (n_1, n_2) \in \text{fld}^S \land (n_2, n_1) \in \text{fld}^S \)
3. (ParallelEdges) \( k = 3 \) and \( (n_1, n_2) \in \text{ref}^S(f_1) \land (n_1, n_2) \in \text{ref}^S(f_2) \land f_1 \neq f_2 \)
4. (Undirected4Cycle) \( k > 3 \)

We say that \( S \) is loop-free admissible if removing all self-loops from \( S \) makes it admissible.

Lemma B.1.5 Let \( S \xrightarrow{d} S' \) s.t. \( S \) is loop-free admissible and \( \text{self}(S) \cap \text{distinct}(S) = \emptyset \).

1. \( \text{shared}(S) = \text{shared}(S') \)
2. \( \text{distinct}(S) = \text{distinct}(S') \)
3. \( \text{self}(S') \cap \text{distinct}(S') = \emptyset \)
4. \( S' \) is loop-free admissible
5. If \( n_1 \neq n_2 \) then

\[
(n_1', n_2') \in \text{ref}^S(f) \Leftrightarrow \exists n_1 \in M_{S,S'}^{-1}(n_1'), n_2 \in M_{S,S'}^{-1}(n_2') . (n_1, n_2) \in \text{ref}^S(f)
\]

Proof:

Let \( S' = \text{contract}(S) \). From Definition B.1.4, we need to show that we never merge two non-adjacent nodes in a state \( S \) if \( n_1 = n_k \) and one of the following conditions hold

1. (SelfLoop) \( k = 2 \)
2. (Directed3Cycle) \( k = 3 \) and \( (n_1, n_2) \in \text{fld}^S \land (n_2, n_1) \in \text{fld}^S \)
3. (ParallelEdges) \( k = 3 \) and \( (n_1, n_2) \in \text{ref}^S(f_1) \land (n_1, n_2) \in \text{ref}^S(f_2) \land f_1 \neq f_2 \)
4. (Undirected4Cycle) \( k > 3 \)

We say that \( S \) is loop-free admissible if removing all self-loops from \( S \) makes it admissible.
Lemma B.1.6 Let $C$ be an admissible concrete state, and $C \rightsquigarrow_{D} S$

1. $sm(C, S) = self(S)$
2. For every summary node $n$ in $S$, $M_{C,S}^{-1}(n)$ induces a rooted tree $T_n$ in $C$ and $n$ is its root

Proof: We show that the properties are preserved by $\rightsquigarrow_{D}$. Assume $S$ satisfies the properties. Let $S'$ be state s.t. $S \rightsquigarrow_{D} S'$, specifically $S' = contract(S, a, b)$.

1. The base case is trivial as $sm(C, C) = \emptyset = self(C)$. For the step case by Proposition B.1.3(2), $self(S') = M_{S,S}(self(S)) \cup sm(S, S')$. By Proposition B.1.2, we have $sm(C, S') = M_{S,S}(sm(C, S)) \cup sm(S, S')$ and by assumption $sm(C, S) = self(S)$ thus $sm(C, S') = self(S')$.

2. Since all contractions are made on edges, $T_n$ is a weakly connected component. Since shared nodes are distinct they cannot be contracted, thus $T_n$ has no sharing. Since $C$ has no garbage, any cycle must either have a variable pointing to a node on it, or there is a path from a variable to the cycle, thus one of its nodes is shared. In any case, an entire cycle cannot be contracted into a single node, thus $T_n$ is weakly connected, has no sharing and no cycles, i.e. it is a tree. Since in contraction a node is always contracted into its parent. The resulting node will be the root of the tree.

We mark $\rightsquigarrow_{D} \rightsquigarrow_{\emptyset}$. $\beta$ can be defined as $\beta \overset{def}{=} \{(S, S') \mid S' = \mathcal{B}(S, \emptyset)\}$.

Definition B.1.7 (Viable) A state $S$ is viable (we write $viable(S)$) iff the following properties hold:

1. A node that has two outgoing $f$ edges has a self-loop of a different edge type
2. Deleting all self-loops from $S$ makes it admissible
3. Distinct nodes are never summary nodes

Lemma B.1.8 Let $C$ be an admissible concrete state. If $C \rightsquigarrow_{D} S$, then $viable(S)$

Proof: By induction on the number of contractions. Since the edge relations of concrete states are partial functions and concrete states have no summary nodes, we have $viable(C)$. Assume $viable(S)$ and let $S'$ be a state s.t. $S \rightsquigarrow_{D} S'$ (Specifically $S' = contract(S, a, b)$). We show $viable(S')$ and complete the proof.

1. Let $n$ be a node with two outgoing $f$ edges. If $n$ has no self-loops, it is not a summary node, thus $n$ represents a single node in every state modeled. Since $ref^{S}(f)$ is a partial function when $C$ is concrete, $n$ can’t have two outgoing $f$ edges. If $n$ has a single $f$ self-loop it embeds a singly linked list of $f$ edges, thus it can have only a single outgoing $f$ edge (from its tail).
2. Immediate from Lemma B.1.5(4).
3. Immediate from Lemma B.1.5(3) and Lemma B.1.6(1).

Theorem 5.2.1 For every feasible abstract state $S$ the following hold:

1. Every $f$ edge in $S$ is an $f$ may edge
2. Every non self-loop $f$ edge is an $f$ unique may edge
3. Every $f$ edge between non-summary nodes is an $f$ must edge
4. A node in $S$ is a summary node iff it has self-loops
5. For every summary node $n$ the subgraph induced by $M^{-1}(n)$ is a tree and has a unique incoming edge which leads to its root.
6. Let $n_1 \neq n_2$ where $n_1$ has no self-loops or a single self-loop of the same type as its outgoing edge. A path from $n_1$ to $n_2$ is a must path.
Proof:
1. Immediate from Proposition B.1.3(1)
2. Immediate from Lemma B.1.5(5)
3. If \( n \) is not a summary node \( |M^{-1}(n)| = 1 \), thus the statement is immediate from Proposition B.1.3(1)
4. See Lemma B.1.6(1)
5. See Lemma B.1.6(2)
6. Since every summary node represents a tree and the first statement states that every edge is a may-edge. Thus paths between non-summary nodes are must paths. Since a summary node is a tree, all the nodes in it are reachable from the root and so if the target node is a summary node it is still a must path. If the source node has a single self-loop it is a singly linked list. The only outgoing edge from a single linked list of the same type as the self-loop is from its last node, thus reachable from all nodes.

**Definition B.1.9 (Expand)**

\[
\begin{align*}
\text{expand}(S, S', a, b, f) & \overset{\text{def}}{=} \\
1. & \quad b \not\in U^S \\
2. & \quad (a, a) \in \text{ref}^S(f) \\
3. & \quad U^{S'} = U^S \cup \{b\} \\
4. & \quad \text{out}^2(\{a, b\}, S') = \text{out}^2(\{a\}, S) \\
5. & \quad \text{within}(\{a, b\}, S') = \text{within}(\{a\}, S) \\
6. & \quad \bigwedge_{g \in \text{PRef}} \text{succ}(U^S - \{a\}, \text{ref}^{S'}(g)) = \text{succ}(U^S - \{a\}, \text{ref}^S(g)) \\
7. & \quad |\text{out}^1(\{a, b\}, S')| = |\text{out}^1(\{a\}, S)| \\
8. & \quad \text{pred}(\{b\}, \text{ref}^{S'}(f)) = \{(a, b)\} \\
9. & \quad \bigwedge_{g \in \text{PRef} - f} \text{pred}(\{b\}, \text{ref}^{S'}(g)) = \emptyset \\
10. & \quad \bigwedge_{g \in \text{PRef}} \text{pred}(\{a\}, \text{ref}^{S'}(g)) \subseteq \text{pred}(\{a\}, \text{ref}^S(g)) \\
11. & \quad \bigwedge_{g \in \text{PRef}} \forall n \in \{a, b\}. |\text{out}^2(\{n\}, S')\{g\}| > 1 \implies \\
12. & \quad |\text{within}(\{n\}, S') - \{g\}| > 0 \\
13. & \quad \text{env}^{S'} = \text{env}^S
\end{align*}
\]

Let \( \leftrightarrow_D \overset{\text{def}}{=} \{(S, S') \mid \text{expand}(S, S', a, b, f, D)\} \). Also, \( \leftrightarrow = \leftrightarrow_D \).

**Lemma B.1.10** Let \( S \) be a state s.t. viable(S) and let

\[
\text{count}(S, X) \overset{\text{def}}{=} \sum_{n \in X} \begin{cases} \text{if } \text{out}^2(S, \{n\}) = \emptyset & \text{then} \\ |\text{within}(S, \{n\})| & \text{else} \\ |\text{out}^2(S, \{n\})| & \text{if } S \end{cases}
\]

- \( a, b \not\in D \implies (\text{expand}(S, S', a, b, f) \iff (\text{contract}(S', S, a, b, f, D) \land \text{viable}(S')))
- \( \text{self}(S) \neq \emptyset \implies \bigvee_{f \in \text{PRef}} \exists S', a, b. \text{expand}(S, S', a, b, f) \land \text{count}(S', \text{self}(S')) < \text{count}(S, \text{self}(S)) \)
- **If \( S \) has no edges between non-distinct nodes then, \( \gamma\{S\} = \{C \mid S \leftrightarrow^* C \land \text{self}(C) = \emptyset\} \)**
Proof:

- Assume $\text{expand}(S, S', a, b, f)$ holds. Line 4,6-10 ensure that $\text{shared}(S) = \text{shared}(S')$ and line 13 that $\text{var}(S) = \text{var}(S')$ thus $\text{distinct}(S) = \text{distinct}(S')$. Line 2 ensures that $a \in \text{sm}(S)$ and by property 3 of $\text{viable}(S)$ this means that $a \notin \text{distinct}(S)$, thus $a \notin \text{distinct}(S')$, since $a \notin D$ we have $a \in J(S', D)$. Lines 8-9 ensure that $b \notin \text{distinct}(S')$ and since $b \notin D$ we have $b \in J(S', D)$. Line 8 also ensures that $(a, b) \in \text{ref}^S(f)$ and line 1 ensures that $a \neq b$. Thus all the preconditions hold. Lines 1,3 ensure that $U^S = U^{S'} - \{b\}$. Line 13 ensures that $\text{env}^S = \text{env}^{S'}$. Line 4-6 ensure that $\text{contract}(S', a, b) = S$. Lines 5,6 ensure that $\text{sm}(S) - \{a\} = \text{sm}(S') - \{a, b\}$. We have seen that $a, b \notin \text{distinct}(S')$ and by property 3 $\text{distinct}(S) \cap \text{sm}(S) = \emptyset$ thus $\text{distinct}(S') \cap \text{sm}(S') = \emptyset$ and property 3 holds for $S'$. Lines 11-12 ensure that property 1 holds for $S'$. All that remains is to show that $S'$ is loop-free admissible (property 2). To see that notice that lines 4,6-10 ensure that the only non self loop edge added is an $f$ edge from $a$ to $b$ and no edges were removed (some may have moved from $a$ to $b$, but there is an edge from $a$ to $b$ so the connectivity does not change). Since $S$ is loop-free admissible it means that $S'$ is loop-free admissible.

Assume $\text{contract}(S', S, a, b, f, D) \land \text{viable}(S')$ holds. Line 1, 3 hold since $U^S = U^{S'} - \{b\}$. Line 2 holds since $(a, b) \in \text{ref}^{S'}(f)$ and it has been contracted. Line 4-6 hold from the definition of $\text{contract}(S', a, b)$. Line 7 holds since $S'$ is loop free admissible thus contract can never merge edges (see also Lemma B.1.5(5)). Line 8-9 hold since $b \notin \text{distinct}(S')$ and $(a, b) \in \text{ref}^{S'}$. Line 10 holds from Lemma B.1.5(5). Lines 11-12 hold because property 1 of $\text{viable}(S')$. Line 13 hold by definition of contract.

2. Let $a \in \text{self}(S)$. If $a$ has two outgoing $g$ edges, by property 2 it has a self-loop of a different type (call it $f$). Let $b \notin U^S$. Look at the state formed by adding $b$ to the universe giving it a single incoming edge (from $a$ of type $f$), moving all the self-loops from $a$ to $b$ and moving all the outgoing edges from $a$ to $b$ (except one $g$ edge that will remain in $a$). It is easy to see that $\text{expand}(S, S', a, b, f)$. Since $a \notin \text{self}(S')$ and $|\text{out}^2(S', \{b\})| = |\text{out}^2(S, \{a\})| - 1$ we have $\text{count}(S', \text{self}(S')) < \text{count}(S, \text{self}(S))$.

Otherwise, $a$ has at most one outgoing $g$ edge for every $g$, choose some self-loop $f$ of $a$, do the same as above except remove the $f$ self-loop from $b$ (we do not need it for the two outgoing edges limitation). It is easy to see that the $\text{count}(S', \text{self}(S')) < \text{count}(S, \text{self}(S))$ here as well ($b$ has no more outgoing edges than $a$ had and has one less self-loop).

- Let $C \in \gamma\{S\}$. By definition $C$ is an admissible concrete state and $C \leftrightarrow^* S$. By Lemma B.1.8 and the first statement we have $S \leftrightarrow^* C$.

Let $C$ be a state s.t. $S \leftrightarrow^* C \land \text{self}(C) = \emptyset$. By the first statement $\text{viable}(C)$ and $C \leftrightarrow^* S$. Since $C$ is viable and has no self-loops, it is concrete and admissible. Furthermore, since $S$ has no edges between non distinct nodes, no further contractions are possible, thus $(C, S) \in \beta$. Thus, $C \in \gamma\{S\}$.

**Lemma 5.2.2 (Feasibility)** (reposted) Abstract state $S$ is feasible iff $\text{viable}(S)$ and there are no edges between two different non-distinct nodes

Proof:
Lemma B.1.8 shows that every feasible state is also viable. In addition, if there is an edge between two different non-distinct nodes it can be contracted. Which contradicts that \( S \) is the image of \( \beta \).

We now show the other direction. First notice that a viable state that has no self-loops is an admissible concrete state. Lemma B.1.10 shows that every state \( S \) that satisfies the above properties can be expanded to an admissible concrete state whose abstraction is \( S \). Thus, \( S \) is feasible.

## B.2 Most Precise Abstract Transformer

**Lemma B.2.1** Let \( S \leftarrow_D S' \). If there is an edge \((n_1,n_2) \in \text{ref}^S(f)\) s.t. \( n_1,n_2 \in (D - \text{sm}(S)) \cup \text{var}(S) \) and the edge is on an undirected cycle, then there is an undirected cycle containing \((n_1,n_2)\) in \( S' \) as well and \( n_1,n_2 \in (D - \text{sm}(S')) \cup \text{var}(S') \).

Proof: First, it is easy to see that neither \( n_1 \) nor \( n_2 \) can be contracted (since \( n_1,n_2 \notin J(S,D) \)), thus they cannot become summary nodes. Furthermore, \( \text{var}(S) = \text{var}(S') \), thus \( n_1,n_2 \in (D - \text{sm}(S')) \cup \text{var}(S') \).

We will examine the cases in Definition B.1.4.

- (SelfLoop) \((n,n) \in \text{fld}^S\). This means that \( n_1 = n_2 = n \). Since \( n \) cannot be contracted and the cycle still exists in \( S' \).
- (Directed3Cycle) \((n_1,n_2) \in \text{fld}^S \land (n_2,n_1) \in \text{fld}^S\). Since neither \( n_1 \) nor \( n_2 \) can be contracted and the cycle still exists in \( S' \).
- (ParallelEdges) \((n_1,n_2) \in \text{ref}^S(f_1) \land (n_1,n_2) \in \text{ref}^S(f_2) \land f_1 \neq f_2\). Since neither \( n_1 \) nor \( n_2 \) can be contracted and the cycle still exists in \( S' \).
- (Undirected4Cycle) \( k > 3 \land n_1 = n_k \). We can limit ourselves to chordless cycles (if a cycle has a chord, the chord induces a smaller cycle). Thus, the contraction has to be on one of the edges of the cycle. If the length of the cycle is greater than 4 the result will be a simple undirected cycle of size greater than 3, and since neither \( n_1 \) nor \( n_2 \) can be contracted the edge is still on that cycle. Otherwise the cycle is of length 4, i.e. contains 3 different nodes. Since 2 of them cannot be contracted, no edge on the cycle can be contracted. Thus, the cycle still exists in \( S' \).

**Proposition B.2.2** Let \( S \leftarrow_D S' \)

1. \( \text{env}^S\{x\} = \text{env}^{S'}\{x\} \)
2. \( \text{ref}^S(g) \circ \text{env}^{S'} \subseteq \text{ref}^S(g) \circ \text{env}^S \)

Proof:

1. Immediate from \( \text{env}^S = \text{env}^{S'} \).
2. Let \((x,n) \in \text{env}^{S'} \land (n,n') \in \text{sel}^{S'}(g)\). By definition of contraction \((x,n) \in \text{env}^S\). Since \( n \in \text{distinct}(S) \) we have \((n,n') \in \text{ref}^S(g)\)

Let \( \text{contract}(E,a,b) \overset{df}{=} \{(u_{p_a \rightarrow a}(n_1),u_{p_b \rightarrow a}(n_2)) \mid (n_1,n_2) \in E\} \)

**Lemma 5.3.2** For every \((S,C) \in \gamma\), let \( D = D(st,C) \) and \( S' = \mathcal{B}(C,D) \). Then:

- \( \beta\{S'\} = \beta\{S\} \) (Idempotence),
• \( B(st(C), D) = B(st(S'), D) \) (Commutative Diagrams), and

• \((\beta \circ st)\{C\} = (\beta \circ st)\{S'\}\) (Equivalence under \(\beta\))

Proof:

• We shall actually prove something stronger: For any state \(S\), \(\beta\{B(S, D)\} = \beta\{S\}\).

  Call \(B(S, D) = S'\). By definition, \(S \hookrightarrow_D S'\). Let \(\beta\{S\} = \{\bar{S}\}\). By definition, \(S \hookrightarrow^* \bar{S}\).

  By confluence there is \(S''\) s.t. \(\bar{S} \hookrightarrow_D^* S''\) and \(S' \hookrightarrow^* S''\). However, \(\bar{S} \hookrightarrow_D^* S''\) implies \(\bar{S} \hookrightarrow^* S''\) and by definition of \(\beta\) this means that \(\bar{S} = S''\). Thus, \(S' \hookrightarrow^* \bar{S}\). Since \(\bar{S}\) is the maximal element for \(\hookrightarrow\), we have \(\beta\{S'\} = \{\bar{S}\}\).

• We shall see that \(\text{viable}(S_1) \land S_1 \hookrightarrow_D S_2 \implies st(S_1) \hookrightarrow_D st(S_2)\). Thus \(st(C) \hookrightarrow_D st(S'')\) and so by the confluence \(B(st(C), D) = B(st(S''), D)\).

Let \(\text{contract}(S_1, S_2, a, b, f, D)\). We will show that the statement holds for \(x = y.h\) and \(x.h = y\), the rest are very similar. For each operation we need to show that we can actually compute \(st\) on \(S_2\) if we could compute it on \(S_1\), i.e., that if the preconditions of \(st\) hold for \(S_1\), they also hold for \(S_2\).

\(- x.h = y\)

**Precondition**
\(\text{env}^{S_2}\{x\} \neq \emptyset \land \text{ref}^{S_2}(h) \circ \text{env}^{S_2}\{x\} = \emptyset\) Immediate from Proposition B.2.2.

**Transformer**
\(st(S_1) = (Us_1, env^{S_1})\),
\(\text{ref}^{S_1}[h := \text{ref}^{S_1}(h) \cup \{(n_x, n_y)|(x, n_x) \in env^{S_1}, (y, n_y) \in env^{S_1}\}]\)
\(st(S_2) = (Us_2, env^{S_2})\),
\(\text{ref}^{S_2}[h := \text{ref}^{S_2}(h) \cup \{(n_x, n_y)|(x, n_x) \in env^{S_2}, (y, n_y) \in env^{S_2}\}]\)
\(st(S_2) = (Us_1 - \{b\}, env^{S_1}, \lambda g.\)
if \(g = h\) then
\(\text{contract}(\text{ref}^{S_1}(h), a, b) \cup \{(n_x, n_y)|(x, n_x) \in env^{S_1}, (y, n_y) \in env^{S_1}\}\)
else
\(\text{contract}(\text{ref}^{S_1}(g), a, b)\)

In \(st(S_1)\) we only added edges to \(S_1\), thus \((a, b) \in \text{ref}^{st(S_1)}(f)\). The only case in which \(a, b\) will be not be in \(J(st(S_1), D)\) is if one of them became shared, but only edges to nodes pointed to by variables where added and none of them are because they are not distinct. Thus \(a, b \in J(st(S_1), D)\)

\(\text{contract}(st(S_1), a, b) =\)
\((Us^{st(S_1)} - \{n_2\}, env^{st(S_1)}, \lambda g.\text{contract}(\text{ref}^{st(S_1)}(g), a, b)) =\)
\((Us_1 - \{b\}, env^{S_1}, \lambda g.\)
if \(g = h\) then
\(\text{contract}(\text{ref}^{S_1}(h) \cup \{(n_x, n_y)|(x, n_x) \in env^{S_1}, (y, n_y) \in env^{S_1}\}, a, b)\)
else
\(\text{contract}(\text{ref}^{S_1}(g), a, b)\)
To complete the proof we show that

\[ \text{contract}(\text{ref}^{S_1}(h) \cup \{(x, n_y) | (x, n_x) \in \text{env}^{S_1} \land (y, n_y) \in \text{env}^{S_1}\}, a, b) = \text{contract}(\text{ref}^{S_1}(h), a, b) \cup \{(x, n_x) | (x, n_x) \in \text{env}^{S_1} \land (y, n_y) \in \text{env}^{S_1}\} \]

However, since \( a, b \notin \text{env}^{S_1}\{\{x\}\} \) and \( a, b \notin \text{env}^{S_1}\{\{y\}\} \) it is immediate.

\(- x = y.h \)

**Precondition**

\( \text{env}^{S_2}\{x\} = \emptyset \land \text{env}^{S_2}\{y\} \neq \emptyset \)

Immediate from Proposition B.2.2.

**Transformer**

\[
\begin{align*}
st(S_1) &= (U^{S_1}, \text{env}^{S_1} \cup \{(x, n) | (y, n) \in \text{ref}^{S_1}(h) \circ \text{env}^{S_1}\}, \text{ref}^{S_1}) \\
st(S_2) &= (U^{S_2}, \text{env}^{S_2} \cup \{(x, n) | (y, n) \in \text{ref}^{S_2}(h) \circ \text{env}^{S_2}\}, \text{ref}^{S_2}) \\
st(S_2)' &= (U^{S_1} - \{b\}, \\
\text{env}^{S_1} \cup \{(x, n) | (y, n) \in \text{contract}(\text{ref}^{S_1}(h), a, b) \circ \text{env}^{S_1}\}, \text{ref}^{S_2})
\end{align*}
\]

Note that \((a, b) \in \text{ref}^{S_1}(f) = \text{ref}^{st(S_1)}(f)\)

The only case in which \(a, b\) will be not be in \(J(st(S_1), D)\) is if one of them became pointed to by a variable. However, if this was the case that node would also be in \(\text{distinct}(st(S_1))\) and thus in \(D\), by Lemma B.1.5 it is also in \(D - sm(S_1)\) and thus not in \(J(S_1, D)\) contradiction. Thus \(a, b \in J(st(S_1), D)\).

\[
\begin{align*}
\text{contract}(st(S_1), a, b) &= (U^{st(S_1)} - \{b\}, \text{env}^{st(S_1)}\{(x, n) | (y, n) \in \text{contract}(\text{ref}^{st(S_1)}(g), a, b)\}) = \\
(U^{S_1} - \{b\}, \text{env}^{S_1} \cup \{(x, n) | (y, n) \in \text{ref}^{S_1}(h) \circ \text{env}^{S_1}\}, \text{ref}^{S_2})
\end{align*}
\]

All that remains is to prove that

\[
\text{ref}^{S_1}(h) \circ \text{env}^{S_1} = \text{contract}(\text{ref}^{S_1}(h), a, b) \circ \text{env}^{S_1}, \text{i.e.,}
\]

if \((x, n) \in \text{env}^{S_1}\)

\((n, n') \in \text{ref}^{S_1}(h) \iff (n, n') \in \text{contract}(\text{ref}^{S_1}(h), a, b).

The only problem can happen if \(n = b\) or \(n' = b\). However, \(n\) is distinct (pointed to by variable) and thus cannot be equal to \(b\), and if \(n' = b\) it is pointed to by a variable in \(S_2\) which we proved cannot happen.

- By commutative diagrams we have \(\mathcal{B}(st(C), D) = \mathcal{B}(st(S'), D)\). By idempotence we have \(\beta\{\mathcal{B}(st(C), D)\} = \beta\{st(C)\}\). By idempotence we have \(\beta\{\mathcal{B}(st(S'), D)\} = \beta\{st(S')\}\). Thus, \(\beta\{st(C)\} = \beta\{st(S')\}\).

**Lemma B.2.3** Let \(C\) be an admissible concrete state and \(D = st(C)\). \(st(C)\) is admissible if and only if \(\text{viable}(\text{ref}(\mathcal{B}(C, D)))\).

Proof: The only if direction is covered by Lemma B.1.8. Assume \(st(C)\) is not admissible. By Proposition B.1.3 if \(st(C)\) has garbage, so does \(\mathcal{B}(st(C), D)\), thus it is not loop-free admissible and \(\neg \text{viable}(\text{ref}(\mathcal{B}(C, D)))\). If \(st(C)\) has an undirected cycle and \(C\) does not (it is admissible), it means that we added an edge that closed a cycle. Thus, the operation has to be \(x.h = y\) and both \(x\) and \(y\) point to the new undirected cycle and there is an \(f\) edge between them. By
Focus(S, x = y.f) { 
    \( N_y = \text{env}^S(y) \)
    \( N_f = \text{ref}^S(f)\{N_y\} \)
    \( \text{AllNewN} = \{(g, \text{newNode}()) \mid g \in \text{within}(S, N_f)\} \)
    return \( \{S'\mid S' = (U^S \cup \text{newN}\{\text{PRef}\}, \text{env}^S, \text{ref}^{S'}), \text{NewN} \subseteq \text{AllNewN}, N'_f = N_f \cup \text{newN}\{\text{PRef}\}, \text{out2}(N'_f, S') = \text{out2}(N_f, S), \text{|out1}(N'_f, S')| = |\text{out1}(N_f, S)|, \) 
    \( \forall (g, n_g) \in \text{NewN}. \text{pred}\{n_g, \text{ref}^{S'}(g)\} = \{(n_f, n_g)\} \wedge \) 
    \( \bigwedge_{g' \in \text{PRef} - \{g\}} \text{pred}\{n_g, \text{ref}^{S'}(g')\} = \emptyset, \) 
    \( \text{within}(N'_f, S') = \text{within}(N_f, S), \) 
    \( \bigwedge_{g \in \text{PRef}} \text{succ}(U^S - N_f, \text{ref}^{S'}(g)) = \text{succ}(U^S - N_f, \text{ref}^{S'}(g)), \) 
    \( \bigwedge_{g \in \text{PRef}} \forall n \in N'_f. |\text{out2}\{n\}, S'\{g\}| > 1 \implies |\text{within}\{n\}, S' - \{g\}| > 0 \) 
} 

Figure B.2: Pseudo code for image computation of \( \text{focus}[x = y.f] \)

Lemma B.2.1 it means that \( \mathcal{B}(st(C), D) \) also has an undirected cycle. If the cycle is a self-loop than we have a distinct node (pointed to by \( x \)) that has a self-loop, thus \( \neg \text{viable}(st(\mathcal{B}(C, D))) \). Otherwise, \( \mathcal{B}(st(C), D) \) is not loop-free admissible and again \( \neg \text{viable}(st(\mathcal{B}(C, D))) \).

Lemma 5.3.4 \( \text{focus}[x = y.f]\{S\} = \text{Focus}(S, x = y.f) \)

Proof: See Figure B.2 for pseudo code of the Focus algorithm. We use the following shorthand:

\( \text{out1}(X, S) \overset{\text{def}}{=} \{(n, n', g) \mid (n, n') \in \text{ref}^S(g), n \in X, n' \notin X\} \)

(Outgoing edges from \( X \) and their type)

\( \text{out2}(X, S) \overset{\text{def}}{=} \{(g, n') \mid (n, n') \in \text{out1}(S, X)\} \)

\( \text{out1}(X, S) \) omitting source nodes

\( \text{within}(X, S) \overset{\text{def}}{=} \{g \mid (n, n') \in \text{ref}^S(g), n \in X, n' \in X\} \)

(Edge types used within \( X \))

Let \( (S, S') \in \text{focus}[x = y.f] \) and \( (S, C) \in \gamma \) s.t. \( S' = \mathcal{B}(C, D(x = y.f, C)) \). Observe that \( D(x = y.f, C) = \text{distinct}(C) \cup N_f \) thus the only contractions that have been made in \( S \) and not in \( S' \) are on edges going out of \( N_f \). Since \( N_f \subseteq D(x = y.f, C) \), \( S' \) can have at most one extra node (call it \( n_g \)) for every self-loop of field \( g \). Since \( S \) can be recovered from \( S' \) by contracting these extra edges (Idempotence) the algorithm simply enumerates only the subgraphs that can be embedded into \( N_f \) in \( S \). In addition, since \( S' \) is the result of \( \mathcal{B} \) on an admissible concrete state the property that a node that has two outgoing \( g \) edges has a self-loop of different reference field, is maintained. Another way to understand focus is by noticing that it is equivalent to expanding all the outgoing edges from \( N_f \) at once.
B.3 Evaluation

**Theorem 5.4.2** For every formula $\varphi$, $\lambda S.\llbracket TR(\varphi)\rrbracket^S$ is a sound evaluation function. If $\varphi$ is guarded, it is also a complete evaluation function.

Proof: The evaluation of $TR(\varphi)$ on $\hat{S}$ simulates the evaluation of a $\varphi$ on any concrete state $C$ s.t. $(S, C) \in \gamma$. Assume an assignment $v_i \mapsto n_i$ satisfies a literal $L(v_1, ..., v_k)$ in $S'$, we shall see that $v_i \mapsto M(n_i)$ satisfies $TR(L)(v_1, ..., v_k)$. By Theorem 5.2.1 we have that predicates of $\hat{S}$ capture exactly the may and must properties of the state. The last case requires further explanation $L \equiv \neg v_1 = v_2$. Here we may chose $n_1 \neq n_2$ s.t. $M(n_1) = M(n_2)$, but in this case $sm(M(n_1))$ thus $TR(L)(v_1, v_2)$ still evaluates to true. Since an NNF formula has no negation outside of literals this is enough for soundness.

For completeness notice that if all nodes involved in a property are non-summary nodes then every may property is also a must property. The guarded formula makes sure that any assignment that may satisfy the formula is to non-summary nodes (nodes pointed to by variables in this case). Thus, any time a may property holds, the appropriate must property hold, and every time a must property does not hold, the appropriate may property does not hold as well.
Appendix C

Proofs and Additional Applications for Chapter 6

C.1 Proofs

Definition C.1.1 (Syntax of FO(TC) formulas) Let \( R = \{ r_1, \ldots, r_n \} \) be a finite set of relation symbols, each with a fixed arity. We assume that \( R \) includes the designated binary relation \( \text{eq} \), denoting equality of individuals (sometimes written as infix \( = \)).

\[ \text{Let } C = \{ c_1, \ldots, c_m \} \text{ be a set of constant symbols. We assume that } C \text{ includes a designated constant } \text{null}. \]

We refer to \( \tau = \langle P, C \rangle \) as the Vocabulary, which is specific for the program and the property to be verified.

We write FO(TC) formulas over \( \tau \) using the logical connectives \( \land, \lor, \neg, \) and the quantifiers \( \forall \) and \( \exists \).

We use \( (\text{when } \varphi_1 \implies \psi_1, \ldots, \text{when } \varphi_k \implies \psi_k, \text{default } \implies \psi) \) as shorthand for a sequential case split, formally:
\[ \ldots \lor (\neg \varphi_1 \land \ldots \land \neg \varphi_{i-1} \land \varphi_i \land \psi_1) \lor \ldots \lor (\neg \varphi_1 \land \ldots \land \neg \varphi_k \land \psi) \]

The operator \( \text{‘TC’} \) denotes transitive closure on a general formula. If \( r \) is a binary relation, \( r^+ \) is shorthand for the transitive closure of \( r \) and \( r^* \) is shorthand for the reflexive-transitive closure of \( r \). Atomic formulas are either \( 1, 0 \), or a relation formula. An atomic formula is called ground if it is does not contain variables. A literal is an atomic formula or its negation.

Definition C.1.2 (Semantics of FO(TC) formulas) A 2-valued interpretation of the language of formulas over \( \tau \) is a 2-valued logical structure \( S = \langle U^S, R^S, C^S \rangle \), where \( U^S \) is a set of individuals, \( R^S \) maps each relation symbol \( r \) of arity \( k \) to a truth-valued function: \( R^S(r): (U^S)^k \to \{0, 1\} \), and \( C^S \) maps each constant symbol \( c \in C \) to an element \( C^S(c) \in U^S \).

For a given formula \( \varphi(v_1, \ldots, v_k) \), with distinct free variables \( v_1, \ldots, v_k \), and a tuple \( \overline{u} \in (U^S)^k \), \([\varphi(\overline{u})]^S\) denotes the value of \( \varphi \) in \( S \) on the tuple \( \overline{u} \). Also, we write \( S, \overline{u} \models \varphi \) when \([\varphi(\overline{u})]^S = 1 \). We sometimes refer to 2-valued logical structures as concrete structures.

Definition C.1.3 A 3-valued interpretation of the language of formulas over \( \tau \) is a 3-valued logical structure \( S = \langle U^S, R^S, C^S \rangle \), as in Definition C.1.2 with the exception that \( R^S \) includes a third truth value \( \frac{1}{2} \) denoting uncertain values, i.e., \( R^S \) maps each relation symbol \( r \) of arity \( k \) to a truth-valued function: \( R^S(r): (U^S)^k \to \{0, 1, \frac{1}{2}\} \). \( C^S \) maps each constant symbol \( c \in C \) into an element \( C^S(c) \in U^S \). For a given formula \( \varphi(v_1, \ldots, v_k) \), with distinct free variables...
Definition C.1.4 gives a formal definition of canonical embedding:

**Definition C.1.4** Let $S$ be a structure, an embedding function $\text{proj}_c$ is a **canonical embedding function** if for every $u_1 \neq u_2 \in U^S$, $\text{proj}_c(u_1) \neq \text{proj}_c(u_2)$ iff there is a unary relation $r$ s.t. $[r(u_1)]^S = 1$ and $[r(u_2)]^S = 0$ (or vice versa) or there is a constant symbol $c$, s.t. $C^S(c) = u_1$ or $C^S(c) = u_2$.

**Proposition C.1.5** Let $S$ be a structure and let $f_1$, $f_2$ be canonical embedding functions. If for every $u \in U^S$, $[r(u)]^S$ is definite, then $f_1(S)$ and $f_2(S)$ are isomorphic (we write $f_1(S) = f_2(S)$).

**Proposition C.1.6** Let $r$ be a unary relation symbol. For every $u \in U^S$, $[r(u)]^S = [r(\beta(u))]^{\beta(S)}$.

Lemma C.1.7 shows that because tight embedding is preserved, all unary relations are definite, and all the constants are mapped to non-summary nodes, $\beta$ will return the same value for both updated structures. Corollary 6.3.3 entails that a monadic-uniform transformer is actually the most precise abstract transformer for focused abstract structures.

**Lemma C.1.7** Let $f(S) = S'$. If for every unary relation $r$ and element $u$ of $S$ we have $[r(u)]^S = [r(f(u))]^{S'}$ and every constant $c$ is mapped by $C^{S'}$ to a concrete node, then $\beta(S) = \beta(S')$.

**Proof:** Let $\text{proj}_c$ and $\text{proj}_c'$ be the canonical embedding functions s.t. $\text{proj}_c(S) = \beta(S)$ and $\text{proj}_c'(S') = \beta(S')$. First it is easy to see that $(\text{proj}_c' \circ f)(S) = \beta(S')$.

Let $g = \text{proj}_c' \circ f$. By Proposition C.1.5 it suffices to show that $g$ is a canonical embedding function, i.e., that for every $u_1 \neq u_2 \in U^S$, $g(u_1) \neq g(u_2)$ iff there is a unary relation $r$ s.t. $[r(u_1)]^S = 1$ and $[r(u_2)]^S = 0$ (or vice versa) or there is a constant symbol $c$, s.t. $C^S(c) = u_1$ or $C^S(c) = u_2$.

Let $u_1 \neq u_2 \in U^S$. For the only if direction, assume that $g(u_1) \neq g(u_2)$, i.e., $\text{proj}_c'(f(u_1)) \neq \text{proj}_c'(f(u_2))$. First, assume that there is a unary relation $r$ s.t. $[r(f(u_1))]^{S'} = 1$ and $[r(f(u_2))]^{S'} = 0$. By assumption we have $[r(u_1)]^S = [r(f(u_1))]^{S'} = 1$ and $[r(u_2)]^S = [r(f(u_2))]^{S'} = 0$. Otherwise, because $\text{proj}_c'$ is a canonical embedding function, there is a constant $c$ s.t. $C^{S'}(c) = f(u_1)$ (the other case is symmetric). However, $C^S(c) = f(C^S(c))$ and because $C^{S'}$ maps $c$ to a concrete node, $1 = [C^{S'}(c) = C^S(c)]^{S'} = [f(u_1) = f(C^S(c))]^{S'}$ and by embedding $[u_1 = C^S(c)]^S = 1$. Thus, $C^S(c) = u_1$.

For the if direction, first, assume that there is a unary relation $r$ s.t. $[r(u_1)]^S = 1$ and $[r(u_2)]^S = 0$. By assumption we have $[r(f(u_1))]^{S'} = [r(u_1)]^S = 1$ and $[r(f(u_2))]^{S'} = [r(u_2)]^S = 0$. Because $\text{proj}_c'$ is a canonical embedding function, we have $\text{proj}_c'(f(u_1)) \neq \text{proj}_c'(f(u_2))$, i.e., $g(u_1) \neq g(u_2)$. Otherwise, assume that there is a constant symbol $c$, s.t. $C^{S'}(c) = u_1$ (the other case is symmetric).

Because $f$ is a embedding function we have $C^{S'}(c) = f(u_1)$. We need to show that $f(u_1) \neq f(u_2)$. However, if $f(u_1) = f(u_2)$, because $C^{S'}$ maps $c$ to a concrete node, we have $1 = [C^{S'}(c) = C^{S'}(c)]^{S'} = [f(u_1) = f(u_2)]^{S'}$ and by tight embedding $[u_1 = u_2]^S = 1$ which contradicts $u_1 \neq u_2$. Thus, $f(u_1) \neq f(u_2)$ and since $\text{proj}_c'$ is a canonical embedding function we have $\text{proj}_c'(f(u_1)) \neq \text{proj}_c'(f(u_2))$, i.e., $g(u_1) \neq g(u_2)$.
Theorem C.1.8 (An embedding theorem (Sagiv, Reps, Wilhelm [SRW02])) Let $S = \langle U^S, R^S, C^S \rangle$ be a 2-valued structure $S' = f(S)$ for some embedding function $f$. Then, for every formula $\varphi$ with free variables $v_1, \ldots, v_k$ for $\varphi$, and $\overline{u} \in (U^S)^k$, we have $[\varphi(\overline{u})]^S \subseteq [\varphi(\text{proj}(\overline{u}))]^S$

Lemma 6.3.2 Let $\tau$ be a monadic-uniform transformer, $S$ be a structure s.t. focused, $\tau(S)$ holds, $C$ be a concrete structure, and $f$ be an embedding function s.t. $f(C) = S$. The following properties hold: (1) $f(\tau(C)) = \tau(S)$, (2) $[\text{guard}_r]^C = [\text{guard}_r]^S$, (3) for every unary relation $r$ and node $u$ we have $r(u)^C = r(f(u))^\tau(S)$, and (4) for every constant $c$, $\tau(S)$ maps $c$ to a concrete node.

Proof: First note that because $S$ is focused it is expanded, i.e., it gives an interpretation to all of the free variables of guard $\tau$. Because $f(C) = S$ so must $C$.

Let $\varphi(v_1, \ldots, v_k)$ be either an update formula or the guard formula. Because $\tau$ is monadic-uniform, $\varphi = (\ldots, \text{when } v_i \mapsto \psi_i, \ldots, \text{default } \mapsto \psi_i)$.

Let $\overline{\tau}$ be a $k$-tuple of nodes of $C$. Because focused, $\tau(S)$ holds, every monadic atomic formula of $\varphi$ has a definite value in $S$. By the definition of Kleene evaluation and the embedding theorem, this means that for every $i$, $[\varphi_i(\overline{\tau})]^C = [\varphi_i(f(\overline{\tau}))]^S$. Let $j$ be the first index for which $[\varphi_j(f(\overline{\tau}))]^S = 1$, or $l$ if all the $\varphi_i$’s evaluate to false. By definition, $[\varphi(f(\overline{\tau}))]^S = [\psi_j(f(\overline{\tau}))]^S$, and because all the $\varphi_i$’s evaluate to definite values, we also have $[\varphi(\overline{\tau})]^C = [\psi_j(\overline{\tau})]^C$.

By definition, $\psi_j$ can be either 1, 0, or a literal whose atomic formula is some relation $q$. Either $[\psi_j(f(\overline{\tau}))]^S$ is definite, in which case, $[\varphi(\overline{\tau})]^C = [\varphi(f(\overline{\tau}))]^S$. Otherwise, $[\psi_j(f(\overline{\tau}))]^S = \frac{1}{2}$, i.e., $R^C(q)(\overline{\tau}) = \frac{1}{2}$, by tight embedding there is some tuple $\overline{u}$ s.t. $f(\overline{\tau}) = f(\overline{u})$ and $R^C(q)(\overline{u}) \neq R^C(q)(\overline{u})$, i.e., $[\varphi_j(\overline{\tau})]^C \neq [\psi_j(\overline{u})]^C$.

However, because for every $i$ the value of $\varphi_i$ is definite, we have, $[\varphi_i(\overline{\tau})]^C = [\varphi_i(f(\overline{\tau}))]^S = [\varphi_i(\overline{u})]^S$, and thus, $[\varphi(\overline{\tau})]^C = [\psi_j(\overline{u})]^C$ and $[\varphi(f(\overline{\tau}))]^S = [\psi_j(f(\overline{u}))]^S$.

Thus, $[\varphi(\overline{\tau})]^C \neq [\varphi(\overline{u})]^C$. In all cases

$$[\varphi(f(\overline{\tau}))]^C = \bigcup_{f(\overline{u}) = f(\overline{\tau})} [\varphi(\overline{u})]^C \quad \text{(C.1)}$$

Let $r$ be a $k$-ary relation and $\overline{\tau}$ be a $k$-tuple of nodes of $C$. Because $[r(\overline{\tau})]^r(S) = [r(\overline{\tau})]^C$ and $[r(f(\overline{\tau}))]^r(S) = [\varphi_r(f(\overline{\tau}))]^S$, Eq. (C.1) implies that the equations of Definition 6.2.4 holds for $r$ and $\overline{\tau}$.

For $\varphi = \text{guard}_r$, because the free variables of guard $\tau$ are treated as constants, $\psi_j$ must be ground and we have $[\text{guard}_r]^C = [\text{guard}_r]^S$.

If $r$ is unary, $\psi_j$ must have at most one variable (call it $v$) and $v$ can appear at most once. Thus $\psi_j$ is monadic or ground, thus evaluates to a definite value and we have $[r(u)]^r(S) = [r(f(u))]^S$ in this case.

Finally, for the update formula $\varphi_c$ for a constant $c$, we have $\psi_j = (v = s_j)$ where $s_j$ is some constant. Since $\psi_j$ is monadic we have for every $u$, $[\varphi_c(f(u))]^S = [\psi_j(f(u))]^S = [\psi_j(u)]^C = [\varphi_c(u)]^C$. However, $C^{\tau(S)}(c) = u_c$, s.t. $[\varphi_c(u_c)]^C = 1$. Thus, $f(u_c)$ is the single node for which $[\varphi_c(f(u))]^S = 1$. In particular, $[f(u) = s_j]^S = 1$ thus $C^{\tau(S)}(c) = C^S(s_j)$, and it must be a concrete node.

Algorithm 6.3.5 gives a way to compute focus, for any monadic-uniform transformer $\tau$ when the feasibility check is decidable. Lemma C.1.9 states its correctness.

Lemma C.1.9 If $S$ is feasible, $\tau$ is monadic-uniform, and there is an algorithm to check every $F$ for feasibility, then $\text{focus}_\tau(S)$ is computable and Algorithm 6.3.5 computes it.
Theorem C.1.10 (Cousot and Cousot [CC79], rephrased\footnote{Note that this is a refinement of the definition given in the introduction to account for (possible) nondeterminism in the transformer.}) For any transformer $\tau$, the most precise abstract transformer for $\tau$, denoted by $bt_\tau$, can be computed by

$$bt_\tau(S) \equiv \{ \beta(C') \mid C' \in [\tau](C) \land C \in \gamma(S) \}$$

Theorem 6.3.6 If $S$ is feasible and $focus_\tau(S)$ is computable, then $bt_\tau(S) \equiv \{ \beta(S') \mid S' \in focus_\tau(S) \land [guard_\tau]|_{S'} = 1 \}$ and it is computable.

Proof: Let $bt'_\tau(S) \equiv \{ \beta(S') \mid S' \in focus_\tau(S) \land [guard_\tau]|_{S'} = 1 \}$. Let $C \in \gamma(S)$ and let $\tau(C') \in [\tau](C)$. Thus, $C' \in expand_\tau(C)$ and $[guard_\tau]|_{C'} = 1$. By Definition 6.3.4 there is $S' \in focus_\tau(S)$ s.t. $C' \in \gamma_\tau(S')$ and $focused_\tau(S')$. By Corollary 6.3.3 we have $\beta(\tau(C')) = \beta(\tau(S'))$, and by Lemma 6.3.2 $[guard_\tau]|_{S'} = [guard_\tau]|_{C'} = 1$. Thus, $bt_\tau(S) \subseteq bt'_\tau(S)$

Let $S' \in focus_\tau(S)$ s.t. $[guard_\tau]|_{S'} = 1$. By definition $\gamma(S') \neq \emptyset$. Thus, by Definition 6.3.4, there are structures $C' \in \gamma(S)$ and $C' \in expand_\tau(C)$, s.t. $C' \in \gamma_\tau(S')$. Thus, $\beta_\tau(C') = S'$. Furthermore, from focused_\tau(S') by Lemma 6.3.2 we have $[guard_\tau]|_{C'} = [guard_\tau]|_{S'} = 1$, i.e., $\tau(C') \in [\tau](C)$. By Corollary 6.3.3 we have $\beta(\tau(C')) = \beta(\tau(S'))$ and . Thus, $bt'_\tau(S) \subseteq bt_\tau(S)$

C.2 More Applications

This appendix expands Section 6.4 with more details on the shape analysis problems we have developed transformers for using the methodology presented in this chapter. The basic structure of the section is the same as in Section 6.4

To simplify the presentation, when giving an algorithm for checking feasibility, we assume that all the possible monadic atomic formulas are focused. This means that we can handle any monadic-uniform update formula for the given vocabulary and class of allowed structures.

Unless stated otherwise, the operations we support are intra-procedural statements handling pointers in Java-like programs (no pointer arithmetic). In all cases we assume that there is no garbage. The initial abstraction tracks at least pointer variables, pointer fields, and reachability. We list the monadic-uniform transformers for three major operations, addition of an edge ($x.next = y$), removal of an edge ($x.next = null$) and traversal of an edge ($x = y.next$). For simplicity, we assume that $x.next == null$ when adding an edge (this can be done by removing the old edge before adding the new one).

C.2.1 Singly-Linked Lists

The parts of the guard formulas that check for null dereferences are simple. In addition, the guard formula must guard against the creation of garbage or cycles: When traversing an edge, we need to make sure that the original value of $x$ is either $null$ or was reachable from some other program variable. When adding an edge, we need to make sure that a cycle has not been formed. This happens only if there was a path from $y$ to $x$. When removing an edge, we need
to make sure that garbage has not formed, which means that there must be a path from some program variable to \( x_n \) that does not go through \( x \).

We can rely only on the unary relations if we notice that the only problem is with \( p_n(v, x) \). However, since there is no garbage, we can replace \( p_n(v, x) \) with \( \bigvee_z r_{z,n}(x) \land r_{z,n}(v) \land \neg r_{x,n}(v) \),

### Cyclicality

The formal definition of \( \text{cut}_n \) is using the following integrity constraint:

\[
\begin{align*}
& (\forall v_1, v_2 . \text{cut}_n(v_1, v_2) \rightarrow n(v_1, v_2)) \land \\
& (\forall v . v \neq \text{null} \implies n^+(v, v) \iff \exists v_1, v_2 . \text{cut}_n(v_1, v_2) \land n^+(v, v_1) \land n^+(v_2, v)) \land \\
& \forall v_1, v_2, w_1, w_2 . \text{cut}_n(v_1, v_2) \land \text{cut}_n(w_1, w_2) \land v_1 \neq w_1 \implies \neg n^+(v_1, w_1)
\end{align*}
\]

As before, we show transformers for the three operations manipulating fields. Table C.1 specifies the monadic-uniform update formulas for these operations. The update formulas for variables and fields remain unchanged and are left out. Cyclicality is no longer considered an error, hence we only need to check for null dereferences and the formation of garbage. See [Hes03] for a detailed explanation of the update formulas.

Updating the unary reachability relations of a program variable \( x \) can be done by replacing \( v_1 \) with \( x \) and \( v_2 \) with \( v \) in the appropriate formulas for \( p'_n \) and \( pc'_n \). We can rely only on the unary relations (removing \( p_n \) and \( pc_n \) from the vocabulary) by observing that the only instances of \( p_n \) or \( pc_n \) that cannot merely be replaced by the appropriate unary reachability relation are \( pc_n(v, x) \) and \( p_n(v, x) \). However, because there is no garbage, we can replace \( pc_n(v, x) \) with \( \neg r_{x,n}(v) \land \bigvee_z r_{z,n}(x) \land r_{z,n}(v) \), and \( p_n(v, x) \) with \( ((c_n(v) \land c_n(x)) \lor \neg r_{x,n}(v)) \land \bigvee_z r_{z,n}(x) \land r_{z,n}(v) \).

**Algorithm C.2.1 (Checking feasibility)**

Replace every summary node with two nodes connected by an edge, all incoming edges to the summary node end in the first node, all outgoing edges start from the second node. If \( c_n \) is set for the summary node and it has no outgoing edges, add an additional edge from the second node to the first and mark the second edge as the cut edge. If there is an edge between two nodes and no \( pc_n \) between them, mark it as a cut edge. Now we can compute \( pc_n \) on the concrete structure. Each edge in the abstract structure is translated into a single edge in the concrete graph. We then simply check that the integrity constraints hold for the candidate structure and that its \( \beta \) is the original structure.

### C.2.2 Trees

Table C.2 specifies monadic-uniform transformers the same three operations. The update formula for variables and edges as the same as in the case of singly-linked lists (replacing \( \text{next} \) with the appropriate field). We list only operations involving \( \text{left} \), for operation involving \( \text{right} \) simply switch between the two everywhere in the update formulas.

The key to updating reachability in this case is the observation that between every two nodes there is at most one path. Thus, the paths that should be removed when removing an edge from \( x \) to \( x_1 \) are exactly the ones that would have been added if this edge was added.

---

2 Because the underlying relation is acyclic and functional, if \( v \) and \( x \) are both reachable from \( z \), either \( v \) is reachable from \( x \) or vice versa.

3 Similar to \( pc_n \), except that if \( x \) and \( v \) are on a cycle they are both reachable from each other.
<table>
<thead>
<tr>
<th>Relation</th>
<th>Update Formula</th>
</tr>
</thead>
</table>
| \( x = y \cdot \text{next} \) | \[
\begin{align*}
guard &\quad n(y, y_n) \land y \neq \text{null} \land (x = \text{null} \lor \bigvee_{z \neq x} r_{z,n}(x)) \\
r’_{x,n}(v) &= c_n(y) \land r_{y,n}(v) \land y \neq v \\
r_{c,x,n}(v) &= rc_{y,n}(y_n) \land r_{y,n}(v) \land y \neq v : r_{y,n}(v) \\
\end{align*}
\] |
| \( x \cdot \text{next} = \text{null} \) | \[
\begin{align*}
guard &\quad n(x, x_n) \land x \neq \text{null} \land (x_n = \text{null} \lor \bigvee_{z} (rc_{z,n}(x_n) \land \neg rc_{z,n}(x))) \\
cut_{n}(v_1, v_2) &\quad \neg pc_{n}(x, x_n) \land p_{n}(v_1, x) \implies pc_{n}(v_1, v_2) \\
p_{n}(v_1, v_2) &\quad \neg pc_{n}(x, x_n) \implies p_{n}(v_1, v_2) \\
rc_{n}(v_1, v_2) &\quad \neg pc_{n}(x, x_n) \implies pc_{n}(v_1, v_2) \land \neg (pc_{n}(v_1, x) \land p_{n}(x, v_2)) \\
&\quad \neg pc_{n}(v_1, x) \lor \neg p_{n}(x, v_2) \implies p_{n}(v_1, v_2) \\
&\quad \neg pc_{n}(v_1, x) \implies pc_{n}(v_1, v_2) \land pc_{n}(v_2, x) \\
&\quad \text{default} \implies pc_{n}(v_1, v_2) \lor pc_{n}(v_2, x) \\
p_{n}(v_1, v_2) &\quad \neg pc_{n}(x, x_n) \implies pc_{n}(v_1, v_2) \\
&\quad \neg pc_{n}(x, x_n) \implies pc_{n}(v_1, v_2) \land \neg (pc_{n}(v_1, x) \land pc_{n}(x, v_2)) \\
&\quad \neg pc_{n}(v_1, x) \lor \neg p_{n}(x, v_2) \implies pc_{n}(v_1, v_2) \\
&\quad pc_{n}(v_1, x) \implies pc_{n}(v_1, v_2) \land pc_{n}(v_2, x) \\
&\quad \text{default} \implies pc_{n}(v_1, v_2) \lor pc_{n}(v_2, x) \\
c_{n}(v) &\quad c_n(v) \land \neg (c_n(x) \land r_{x,n}(v)) \\
\end{align*}
\] |
| \( x \cdot \text{next} = y \) | \[
\begin{align*}
guard &\quad x \neq \text{null} \\
cut_{n}(v_1, v_2) &\quad cut_{n}(v_1, v_2) \lor (v_1 = x \land v_2 = y \land p_{n}(y, x)) \\
p_{n}(v_1, v_2) &\quad p_{n}(v_1, v_2) \lor (p_{n}(v_1, x) \land p_{n}(y, v_2)) \\
rc_{n}(v_1, v_2) &\quad pc(v_1, v_2) \lor (\neg p_{n}(y, x) \land p_{n}(v_1, x) \land p_{n}(y, v_2)) \\
\end{align*}
\] |

Table C.1: Monadic-uniform transformers for possibly cyclic singly-linked lists.
C.2. More Applications

### Relation | Update Formula
---|---
\(x = y.\text{left}\) | \((y_l, y_l) \land r(y_l, y_r) \land y \neq \text{null} \land (x = \text{null} \lor \bigvee_{z \neq x} p(z, x))\)
\(x'_l\) | \(y_l\)
\(x'_r\) | \(y_r\)
\(r'_{x,l}(v)\) | \(p(y_l, v)\)
\(r'_{x,r}(v)\) | \(p(y_r, v)\)

\(x.\text{left} = \text{null}\) | \(x \neq \text{null} \land (x_l = \text{null} \lor \bigvee_z z = x_l)\)
\(z'_l\) | \((\text{when } x = z \implies \text{null}, \text{default } \implies z_l)\)
\(r'_{z,l}(v)\) | \(r_{z,l}(v) \land x \neq z \land \neg(r_{z,l}(x) \land r_{x,l}(v))\)
\(r'_{z,r}(v)\) | \(r_{z,r}(v) \land \neg(r_{z,r}(x) \land r_{x,l}(v))\)
\(p'(v_1, v_2)\) | \((p(v_1, v_2) \land \neg(p(v_1, x) \land p(x_l, v_2)))\)

\(x.\text{left} = y\) | \(x \neq \text{null} \land \neg p(y, x) \land \bigwedge_z \neg(p(z, y) \land z \neq y)\)
\(z'_l\) | \((\text{when } x = z \implies y, \text{default } \implies z_l)\)
\(r'_{z,l}(v)\) | \(r_{z,l}(v) \lor ((z = x \lor r_{z,l}(v)) \land p(y, v))\)
\(r'_{z,r}(v)\) | \(r_{z,r}(v) \lor (r_{z,r}(v) \land p(y, v))\)
\(p'(v_1, v_2)\) | \((p(v_1, v_2) \lor (p(v_1, x) \land p(x_l, v_2)))\)

Table C.2: Monadic-uniform transformers for trees.

Since trees have no sharing, when removing an edge we require a variable to point to the target of the edge, otherwise the removal creates garbage. When adding an edge from \(x\) to \(y\) we need to make sure there is no sharing added to \(y\) and that no cycles have formed. Sharing on \(y\) is created only when there is a variable that reaches \(y\) but not equal to \(y\). Cycles are formed only when \(x\) was reachable from \(y\).

Algorithm C.2.2 describes how to check feasibility of a focused abstract structure. It is based on the same ideas of Algorithm 6.4.1.

**Algorithm C.2.2 (Checking feasibility)** There are three types of abstract nodes in an abstract structure for this shape analysis problem: Constants, nodes that have a (non-null) constant reachable from them (i.e. for node \(v\) and some constant \(z\), s.t. \([p(u, z)]^S = 1\), and nodes that do not reach any constant (we call them sink nodes).

In a method similar to singly-linked lists, we find a single candidate concrete structure. The nodes that reach a constant are replaced with a segment containing one edge for each self loop (e.g., if there is an \(l\) and an \(r\) self-loops, the segment will be \(u_1, u_2, u_3\) s.t. \(l(u_1, u_2) \land r(u_2, u_3)\)). Incoming edges are connected to the first node, outgoing edges to other non-sink nodes are connected to the last node of the segment, and an outgoing edge to a sink node starts at a non-last node in the direction that does not participate in the segment (except if \(p\) has the value 1 for this pair of nodes, in which case the edge is connected to the last node of the segment).

Sink nodes are replaced with a forest. Each incoming edge leads to the root of a separate tree. Each tree contains exactly one edge from each of the self-loops.

Return true iff the candidate structure satisfies the integrity constraints and when we compute \(\beta\) on this structure we get the original abstract structure.
Table C.3 lists the monadic-uniform transformers for the relations in the vocabulary and the three operations. Updating program variables, fields, and reachability is the same as in trees (since both have the property of at most one path between any two nodes). We need to maintain the $s_{u,w}$ constants and use them in guard formulas (i.e., to detect formation of garbage or undirected cycles). The update formulas for the $s_{u,w}$ constants may seem elaborate, but this is a simple case analysis of the position of the shared node in relation to the updated edge.

**Algorithm C.2.3** (Checking feasibility) Consider the nodes marked with the $s_{x,y}$ constants as extra program variables and apply Algorithm C.2.2, only this time check the candidate concrete structure with the different integrity constraints (i.e., has no undirected cycles instead of being a tree).

**C.2.3 No Undirected Cycles**

The formal definition of $s_{x,y}$ is given using the following integrity constraint:

$$(\exists v_x, v_y . p(x, v_x) \land \neg p(x, v_y) \land \neg p(y, v_y) \land \neg p(y, v_x) \land (l(v_x, s_{x,y}) \lor r(v_x, s_{x,y}) \lor (l(v_y, s_{x,y}) \lor r(v_y, s_{x,y})))) \leftrightarrow s_{x,y} \neq null$$

Table C.3 lists the monadic-uniform transformers for the graph with no undirected cycles.
Detecting Undirected Cycles

Undirected cycles can be formed when an edge is added. Adding an edge from $x$ to $y$ closes an undirected cycle only when before the addition there was already an undirected path from $y$ to $x$.

Consider the shortest undirected path between two nodes. Observe that the undirected path is like a directed path that can sometimes traverse the edges in the normal direction (down) and sometimes in the opposite direction (up). Let $ud_i$ be the $i$th node in which we switch directions from going up to going down, $ud_i$ has to be reachable from some program variable $z_i$. Let $du_i$ be the $i$th node in which we switch directions from going down to going up, $du_i$ has to a share point reachable from the two adjacent $ud$’s, there are program variables $u$ and $w$ s.t. $s_{u,w}$ is the shared node.

Thus, if there is an undirected path from $x$ to $y$ there is a sequence of program variables $z_1, \ldots, z_k$ s.t. $p(z_1, x)$ and every $z_i, z_{i+1}$ meet at a node, i.e., $s_{z_i, z_{i+1}} \neq \text{null}$, and $z_k$ either reaches $y$ directly or meets it at a node, i.e., $p(z_k, y) \lor s_{z_k, y} \neq \text{null}$. Trying to encode this directly as a formula would result in a formula exponential in the number of program variables. However, we can use $TC$ to compute exactly that. The basic relation is $\varphi(v_1, v_2) \overset{\text{def}}{=}$ $\bigvee_{z_1, z_2} v_1 = z_1 \land v_2 = z_2 \land (p(z_1, z_2) \lor p(z_2, z_1) \lor s_{z_1, z_2} \neq \text{null})$. This means that $v_1$ and $v_2$ are pointed to by program variables $z_1$ and $z_2$, and either one of them is reachable from the other, or they have a common shared node. An undirected path from $x$ to $y$ is simply $(TC\ v_1, v_2 : \varphi(v_1, v_2))(x, y)$.

Note that even though $TC$ is used, the formula is still monadic uniform. This example shows that the useful class of monadic-uniform formulas extends beyond quantifier-free formulas.

C.2.4 Shared Trees

Table C.4 specifies monadic-uniform transformers for the three operations. The update formulas for all relations except The guard formulas for most transformers also needs to be updated.

The difficulty in using monadic-uniform transformers for this class of structures is detecting when the shared-trees property has been violated when adding an edge, i.e., there are two nodes $w$ and $v$ s.t., there is already a path from $w$ to $v$ and adding an edge from $x$ to $y$ will create another path between them, i.e. there is a path from $w$ to $x$ and from $y$ to $v$. However, because there is no garbage, there is some program variable $z$ s.t. $p(z, w)$, and so $p(z, x)$ and $p(z, v)$. Thus, we can use the following guard formula for addition of an edge: $x \neq \text{null} \land \neg \bigvee_{z} p(z, x) \land \exists v. (p(z, v) \land p(y, v))$. Note the although the formula is quantified, all the atomic formulas are monadic or ground, thus the formula is monadic-uniform.
We are currently working on feasibility check for shared trees. Notice that we cannot simply translate this to a problem in MSO since shared trees have unbounded tree width and as such MSO is not decidable for them. However, we believe that for this vocabulary, feasibility is decidable.

C.2.5 Uninterpreted Unary Relations

Singly Linked Lists

At the moment, we have direct feasibility check only in the case we maintain only unary reachability.

Algorithm C.2.4 (Checking feasibility) First, generate a structure $S'$ in which we ignore the colors and collapse abstract nodes that are now indistinguishable. We can then use either Algorithm 6.4.1 or Algorithm C.2.1 as appropriate. If this structure is infeasible so is the original one, thus we return false. Next, we check whether there are 1 $n$-edges incident to summary nodes, returning false if such exist.

Otherwise, we take the substructures induced by each set of nodes we collapsed together and consider them one at a time.

Each substructure represents either a segment of the list or an uninterrupted cycle. Thus, it should either have a single incoming and outgoing edge or be a cycle with all its incoming edges pointing to a single node (call it the entry node). Since the only binary information we have is the $n$ relation, we must check that we can build a path from the incoming edge to the outgoing edge (or back to the entry node in case of a cycle). This can be done using a reduction to the Directed Chinese Postman Problem \cite{EJ73} (DCP). A DCP problem is to find, given a graph with weights on the edges and the path with lowest weight that goes through each edge at least once. Let $e$ be the number of edges in the segment. To model the concrete nodes that can only be traversed once in the path, we use the observation that the solution can traverse at most $e^2$ edges. Thus, we give edge incident to concrete nodes the weight $w = e^2 + 1$. Let $n$ be the number of edges incident to concrete nodes. The structure is feasible iff there is a solution to the DCP problem with weight $< w \ast (n + 1)$. DCP problems can be solved in polynomial time, see \cite{EJ73} for details.

Self-loops are unimportant except for the case in which a summary node has a single incoming edge, a single outgoing edge no self-loops. In this case to satisfy the requirement that the summary node embeds at least two node we split the summary node into two summary splitting the incoming and outgoing edge accordingly.

Lemma C.2.5 Algorithm C.2.4 returns true on a structure $S$ iff $S$ is feasible

Trees and No Undirected Cycles

We can translate the feasibility check to satisfiability check of an MSO formula on trees. In case of No Undirected Cycles, this also requires breaking the edges incoming to $s_{x,y}$ and replacing them with constants in the formula.
C.2.6 Doubly-Linked Lists

To analyze doubly linked lists we use a vocabulary similar to the singly-linked lists cases (adding $pc_{sel}$ and $c_{sel}$ only if we want to support cycles). Contrary to trees, for doubly linked list we use separate relations for reachability using the $f$ (forward) and $b$ (backward) fields. We do not track paths involving both fields.

As in [SRW02] we use two additional unary relations, $c_{f,b}(v)$ and $c_{b,f}(v)$. $c_{f,b}(v)$ means that traversing the $f$ field from $v$ and then the $b$ field brings us back to $v$. $c_{b,f}(v)$ means that traversing the $b$ field from $v$ and then the $f$ field brings us back to $v$. TVLA uses a slightly different formulation, for which we also have monadic-uniform transformers.

The update formulas for the additional relations is given in Table C.5. Only operations manipulating $f$ are listed. To get the update formulas for operations manipulating $b$, simply reverse the roles of $f$ and $b$ in the formulas.

Removing an $f$ edge means that it now points to null which cannot point back, thus, $c_{f,b}(x)$ cannot hold after the update. As for $c_{b,f}$, note that the only case in which removing an $f$ edge breaks the condition, is when the edge removed is the inverse of the removed edge. The rest of the updates are straightforward.

We can use an algorithm very similar to the one in [LAIS06] to check feasibility of doubly linked lists in which any segment between variables is either a full doubly linked lists or a singly linked list. This is the case in most algorithms manipulating doubly linked lists. We are working on a general feasibility check algorithm for this case, we believe it is decidable.

C.2.7 Ordering

In [LARSW00] we show how to abstract data values of fields using a binary $dle(v_1, v_2)$ relation (i.e., the data value in $v_1$ is less or equal to the data value in $v_2$). Since null has no data for every $v$ we have $\neg dle(v, null)$ and $\neg dle(null, v)$. We use the following auxiliary relations: $inOrd_{n,dle}(v)$ means that if $n(v, u)$ then $dle(v, u)$ $7$. $inROrd_{n,dle}(v)$ means that if $n(v, u)$ then $dle(u, v)$ $8$. This abstraction has been used to prove partial correctness of several sorting algorithms.

<table>
<thead>
<tr>
<th>Relation</th>
<th>Update Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x.f = null$</td>
<td>$v \neq x \land c_{f,b}(v)$</td>
</tr>
<tr>
<td>$c_{f,b}(v)$</td>
<td>$v = x \land c_{b,f}(y) \land \neg b(x,f,x) : c_{b,f}(v)$</td>
</tr>
<tr>
<td>$c_{b,f}(v)$</td>
<td>$v = y \land b(y,x) : c_{b,f}(v)$</td>
</tr>
</tbody>
</table>

Table C.5: Monadic-uniform transformers for doubly-linked lists.
Table C.6: Monadic-uniform transformers for ordering.

<table>
<thead>
<tr>
<th>Relation</th>
<th>Update Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.next = null</td>
<td>x \neq v \wedge \text{inOrd}_{\text{n,dle}}(v)</td>
</tr>
<tr>
<td>\text{inOrd}_{\text{n,dle}}(v)</td>
<td>v \neq x \wedge \text{inOrd}_{\text{n,dle}}(v)</td>
</tr>
<tr>
<td>\text{inROrd}_{\text{n,dle}}(v)</td>
<td>v \neq x \wedge \text{inROrd}_{\text{n,dle}}(v)</td>
</tr>
<tr>
<td>x.next = y</td>
<td>v = x \wedge \text{dle}(x,y) : \text{inOrd}_{\text{n,dle}}(v)</td>
</tr>
<tr>
<td>\text{inOrd}_{\text{n,dle}}(v)</td>
<td>v = x \wedge \text{dle}(x,y) : \text{inOrd}_{\text{n,dle}}(v)</td>
</tr>
<tr>
<td>\text{inROrd}_{\text{n,dle}}(v)</td>
<td>v = x \wedge \text{dle}(y,x) : \text{inROrd}_{\text{n,dle}}(v)</td>
</tr>
</tbody>
</table>

Table C.6 specifies the update formulas for the new relations. Only operations changing the pointer fields are supported, not the data fields. The idea behind the update formulas is very similar to the update of doubly-linked lists.

We are working on feasibility check for this case (with singly or doubly linked lists), we believe it is decidable.
Appendix D

Proofs for Chapter 7

D.1 Pre-Order

The proof of \( \preceq_{P\times N} \) being a pre-order follows from Claim D.1 and Claim D.1, stated and proved below. We start by introducing some useful notation and a useful lemma.

We use the notation \( \text{fst}(P,N) \) and \( \text{snd}(P,N) \) to project out the first and second elements of the input pair respectively, i.e., \( \text{fst}(P,N) = P \) and \( \text{snd}(P,N) = N \).

The following lemma states that the pre-order \( \preceq_{P\times N} \) is closed under the Saturate operator.

**Lemma D.1.1** Let \((P_1, N_1) \preceq_{P\times N} (P_2, N_2)\), i.e., \( P_1' \preceq_P P_2 \) and \( \text{p2n}(P_1', P_2, N_1') \preceq_N N_2 
\), where \((P_1', N_1') = \text{Saturate}(P_1, N_1)\).

Then \((P_1, N_1) \preceq_{P\times N} \text{Saturate}(P_2, N_2)\), i.e., \( P_1' \preceq_P P_2' \) and \( \text{p2n}(P_1', P_2', N_1') \preceq_N N_2' 
\), where \((P_2', N_2') = \text{Saturate}(P_2, N_2)\).

**Proof:** We first prove that \( P_1' \preceq_P P_2' \).

\[
\begin{align*}
P_1' &= \text{fst}(\text{Saturate}(P_1', N_1')) \quad \text{(D.1)} \\
     &= \text{fst}(\text{Saturate}(P_1', \text{p2n}(P_1', P_2, N_1'))) \quad \text{(D.2)} \\
     \preceq_P \text{fst}(\text{Saturate}(P_2, N_2)) \quad \text{(D.3)} \\
     &= P_2' \quad \text{(follows from Definition of } P_2')
\end{align*}
\]

Eq. D.1 follows from the definition of the Saturate operator, and Eq. D.3 follows from the fact that Saturate is (clearly) monotone in both its arguments.

Eq. D.2 follows from the fact that adding extra information to \( N_1' \) (using the \text{p2n} operator) about the special variables that denote the cardinalities of the base-sets in \( P_2 \) (which we can assume without loss of generality are fresh variables that did not already occur in \( N_1' \)) does not affect any information that \( N_1' \) can communicate to \( P_1' \) (in the Saturate process). This is because a formula \( \varphi \) is logically equivalent to a formula \( \exists X (\varphi \land \varphi') \), if \( X \) does not occur in \( \varphi \) and \( \varphi' \) has the property that \( \exists X (\varphi') \equiv \text{true} \).
We now prove that \( P_2 N(P'_1, P'_2, N'_1) \preceq_N N'_2 \).

\[
P_2 N(P'_1, P'_2, N'_1) \\
= \quad P_2 N(P'_1, P'_2, \text{snd}(\text{Saturate}(P'_1, N'_1))) \tag{D.4} \\
= \quad \text{snd}(\text{Saturate}(P'_1, P_2 N(P'_1, P'_2, N'_1))) \tag{D.5} \\
\preceq_N \quad \text{snd}(\text{Saturate}(P_2, N_2)) \tag{D.6} \\
= \quad N'_2 \tag{D.7}
\]

Eq. (D.4) follows from the definition of the Saturate operator, and Eq. (D.6) follows from the fact that Saturate is monotone in its arguments. Eq. (D.5) follows from the fact that during Saturate\((P, N)\), \( N \) communicates information to \( P \) only about the base-sets in \( \text{BaseSets}(P) \) while during \( P_2 N(P, P', N) \), \( P \) communicates information to \( N \) only about the base-sets in \( \text{BaseSets}(P') \); and without loss of any generality, we can assume that \( \text{BaseSets}(P_1) \) and \( \text{BaseSets}(P_2) \) are disjoint in Eq. (D.5).

Claim: The pre-order \( \preceq_{P_2 N} \) is transitive.

Proof: Suppose the following holds.

\begin{itemize}
  \item \( (P_1, N_1) \preceq_{P_2 N} (P_2, N_2) \), i.e., \( P'_1 \preceq_P P_2 \) and \( P_2 N(P'_1, P'_2, N'_1) \preceq_N N_2 \), where \( (P'_1, N'_1) = \text{Saturate}(P_1, N_1) \).
  \item \( (P_2, N_2) \preceq_{P_2 N} (P_3, N_3) \), i.e., \( P'_2 \preceq_P P_3 \) and \( P_2 N(P'_2, P'_3, N'_2) \preceq_N N_3 \), where \( (P'_2, N'_2) = \text{Saturate}(P_2, N_2) \).
\end{itemize}

We want to show that \( (P_1, N_1) \preceq_{P_2 N} (P_3, N_3) \), i.e., \( P'_1 \preceq_P P_3 \) and \( P_2 N(P'_1, P'_3, N'_1) \preceq_N N_3 \).

We first prove that \( P'_1 \preceq_P P_3 \). Note that it follows from Lemma D.1.1 that \( P'_1 \preceq_P P'_2 \). We are given that \( P'_2 \preceq_P P_3 \). Hence, it follows from the transitivity of \( \preceq_P \) that \( P'_1 \preceq_P P_3 \).

We now prove that \( P_2 N(P'_1, P'_3, N'_1) \preceq_N N_3 \). Let \( V = \text{SpecialVars}(P'_2) \).

\[
P_2 N(P'_1, P'_3, N'_1) \\
= \exists V : P_2 N(P'_2, P_3, P_2 N(P'_1, P'_2, N'_1)) \tag{Property 7.3.2} \\
\preceq_N \exists V : P_2 N(P'_2, P_3, N'_2) \tag{D.8} \\
\preceq_N \exists V : N_3 \\
= \quad N_3
\]

Eq. (D.8) follows from the fact that \( P_2 N \) is monotonic in its third argument and the fact that \( P_2 N(P'_1, P'_2, N'_1) \preceq_N N'_2 \) (the latter fact follows from Lemma D.1.1).

Claim: The pre-order \( \preceq_{P_2 N} \) is reflexive.

Proof: Let \((P, N)\) be any abstract element from the combined domain. Let \((P_1, N_1) = \text{Saturate}(P, N)\). Clearly, \( P_1 \preceq_P P \). Also, note that \( P_2 N(P_1, P, N) \preceq_N N_1 \preceq_N N \). Hence, the pre-order \( \preceq_{P_2 N} \) is reflexive.

**D.2 Lemmas used in Widening Proof**

This section states and proves the lemmas used in the proof of Theorem 7.4.12. We first introduce some notation.
Let \((P_i, N_i)\) be an increasing sequence, \((Q_1, M_1) = (P_1, N_1)\), and \((Q_{i+1}, M_{i+1}) = \text{Widen}_{\text{P2N}}((Q_i, M_i), (P_i, N_i))\). Following the \text{Widen}_{\text{P2N}} algorithm in Figure 7.8(a), we use the following notations:

\[
(P'_i, N'_i) = \text{Saturate}(P_i, N_i)
\]

\[
Q_{i+1} = \text{Widen}_{\text{P}}(Q_i, P'_i)
\]

\[
V_i = \text{SpecialVars}(M_i) - \text{SpecialVars}(Q_{i+1})
\]

\[
W_i = \text{SpecialVars}(N'_i) - \text{SpecialVars}(Q_{i+1})
\]

\[
M''_{i+1} = \text{Eliminate}_{\text{N}}(\text{P2N}(Q_i, Q_{i+1}, M_i), V_i)
\]

\[
N''_{i+1} = \text{Eliminate}_{\text{N}}(\text{P2N}(P'_i, Q_{i+1}, N'_i), W_i)
\]

\[
M_{i+1} = \text{Widen}_{\text{N}}(M''_{i+1}, N''_{i+1})
\]

We now state and prove the lemmas used in the proof of Theorem 7.4.12.

**Lemma D.2.1** The number of strictly increasing steps in the sequence \(Q_1, Q_2, \ldots\) is bounded.

**Proof:** For every \(i\), we have \(P'_i \leq_{\text{P}} P'_{i+1}\) by Lemma 1 using \((P_i, N_i) \leq_{\text{P2N}} (P_{i+1}, N_{i+1})\). Furthermore, since \(Q_i = P_1\) and \(Q_{i+1} = \text{Widen}_{\text{P}}(Q_i, P'_i)\), by convergence of \text{Widen}_{\text{P}}, the number of strictly increasing steps in the sequence \(Q_1, Q_2, \ldots\) is bounded.

**Lemma D.2.2** For every \(i > 0\), if \(Q_i\) and \(Q_{i+1}\) are equal up to variable renaming then \(N''_{i-1} \leq_{\text{N}} N''_i\) and \(M_{i+1}\) and \(\text{Widen}_{\text{N}}(M_i, N''_i)\) are equal up to variable renaming.

**Proof:** First, we prove that \(N''_{i-1} \leq_{\text{N}} N''_i\). Let \(U = W_{i-1} \cup \text{SpecialVars}(P'_i)\)

\[
N''_{i-1} = \text{Eliminate}_{\text{N}}(\text{P2N}(P'_{i-1}, Q_i, N'_{i-1}), W_{i-1})
\]

\[
\leq_{\text{N}} \text{Eliminate}_{\text{N}}(\text{P2N}(P'_{i-1}, Q_i, N'_{i-1}), U)
\]

\[
= \text{Eliminate}_{\text{N}}(\text{P2N}(P'_i, Q_i, \text{P2N}(P'_{i-1}, P'_i, N'_{i-1})), U) \quad (\text{D.9})
\]

\[
\leq_{\text{N}} \text{Eliminate}_{\text{N}}(\text{P2N}(P'_i, Q_i, N'_i), U) \quad (\text{D.10})
\]

\[
\leq_{\text{N}} \text{Eliminate}_{\text{N}}(\text{P2N}(P'_i, Q_{i+1}, N'_i), W_i) \quad (\text{D.11})
\]

Eq. D.9 follows from Property 1 since \(P'_{i-1} \leq_{\text{N}} P'_i \leq_{\text{N}} Q_i\). Eq. D.10 follows from the fact that \text{P2N} is monotonic in its third argument and the fact that \(\text{P2N}(P'_{i-1}, P'_i, N'_{i-1}) \leq_{\text{N}} N'_i\) (the latter fact follows from Lemma 1). Eq. D.11 holds because \(Q_i\) and \(Q_{i+1}\) are equal up to variable renaming and all the variables that appear in either \(P'_i\) or \(Q_i\) are in \(U \cap W_i\); thus only variables that have no effect are changed.

Now we prove that \(M_{i+1}\) and \(\text{Widen}_{\text{N}}(M_i, N''_i)\) are equal up to variable renaming. By definition we have \(M''_i = \text{Eliminate}_{\text{N}}(\text{P2N}(Q_i, Q_{i+1}, M_i), V_i)\), and thus, since \(Q_i\) and \(Q_{i+1}\) are equal up to variable renaming, this has the effect of renaming the variables of \(Q_i\) to their respective counterparts in \(Q_{i+1}\). Thus, \(M''_i\) and \(M_i\) are equal up to variable renaming which means that \(M_{i+1}\) and \(\text{Widen}_{\text{N}}(M_i, N''_i)\) are equal up to variable renaming.

**Lemma D.2.3** For every sequence \((Q_i, M_i)\) there is an \(m\) that satisfies the following property:
Let \(i\) be an index s.t., for every \(i \leq k \leq i + m\), \(Q_k\) and \(Q_k\) are equal up to variable renaming. There is an index \(j < i + m\) s.t., for every \(j \leq k \leq i + m\), \(M_j\) and \(M_k\) are equal up to variable renaming.
Proof: Immediate from Lemma D.2.2 by the convergence of the \( Widen_N \) operator.

**Lemma D.2.4** (1) \( Q_i \preceq_P Q_j \) for all \( i < j \), and (2) If \( Q_j \preceq_P Q_i \) for some \( i < j \), then \( M_k \preceq_N M_{k+1}\sigma \) and \( M_{k+1}\sigma \not\preceq_N M_k \) for all \( i \leq k < j \) (otherwise the fixed-point computation converges), where \( \sigma \) is some bijective variable renaming.

Proof: (1) comes from the soundness of \( Widen_P \) since \( Q_{i+1} = Widen_P(Q_i, P'_i) \). As for (2), anti-symmetry of \( \preceq_P \) implies that \( Q_i, \ldots, Q_j \) are equal up to variable renaming. Thus, by Lemma D.2.2, \( M_{k+1} \) and \( Widen_N(M_k, N''''_k) \) are equal up to variable renaming, i.e., \( M_{k+1}\sigma = Widen_N(M_k, N''''_k) \), where \( \sigma \) is some bijective variable renaming. Finally, soundness of \( Widen_N \) gives us (2).