REFINEMENT-BASED PROGRAM VERIFICATION VIA THREE-VALUED-LOGIC ANALYSIS

by

Alexey A. Loginov

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2006

Dedicated to the loving memory of my father, Alexander Volodin, and my mother, Elena Loginova. They would have been proud ...

ACKNOWLEDGMENTS

First and foremost, I am very grateful to my advisor, Professor Thomas Reps. His infinite patience and thoughtful counsel guided me every step of the way. He always put the student's interest first; he taught taste in research and demanded perfection.

I thank my wife, Wendy, for her unfaltering love and support through my years in graduate school. I could not have reached this point without her strength and caring.

I thank my brother, Vladimir Lifschitz, and my sister-in-law, Elena Lifschitz, who brought me to the United States and had a profound influence on me in ways that I cannot enumerate. My life has been transformed since the day we met.

I thank Sasha and Amy, Wally and Kathy, Randy and Heather, and the rest of my family. Their love and support helped get through the low points and celebrate the high ones.

I am deeply indebted to Mooly Sagiv, with whom I had the pleasure of collaborating on many projects. Mooly played a big role in my development as a researcher. I am thankful to many other collaborators, Bertrand Jeannet, Susan Horwitz, Suan Yong, Eran Yahav, Roman Manevich, Greta Yorsh, Noam Rinetzky, and Tal Lev-Ami, as well as the students in Susan's and Tom's research groups, Raghavan Komondoor, David Melski, Denis Gopan, Gogul Balakrishnan, Akash Lal, Nick Kidd, and Junghee Lim, for our many interesting discussions. I thank Neil Immerman and William Hesse for their invaluable help on finite differencing and Richard Bornat for suggesting an interesting challenge problem.

I thank Irene Ong, who introduced me to Inductive Logic Programming and patiently answered many questions as I was learning about this fascinating technique.

I thank the members of my Ph.D. committee, Susan Horwitz, Marvin Solomon, Ben Liblit, and C. David Page, for their insightful comments on my thesis and stimulating discussions before, during, and after my defense.

I thank Ian, Irene, Denis, Shura, Mihai, Sondra, Kevin, Ina, Hao, Jaime, Pedro, Ana, Vuk, Magdalena, Brad, Jen, Dave, Suan, Marc, Cheryl, Peter, and the rest of my friends in this very special place. I cannot imagine life in Madison without them. I will also remember with fondness the many wonderful dinner parties that were hosted by Jaime, Hao, and Giordano.

I thank Tim Jaglinski, Ainars Marnauzs, and the rest of the Flying Armadillos hockey team, as well as Andy Sullivan and the rest of the outdoor skating group, for the support, the fun, and the crisp passes.

I could not have completed my dissertation without the help and support from many people, all of whom could not be mentioned here. I am deeply grateful to all of you ...

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

Page

DISCARD THIS PAGE

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

REFINEMENT-BASED PROGRAM VERIFICATION VIA THREE-VALUED-LOGIC ANALYSIS

Alexey A. Loginov

Under the supervision of Professor Thomas W. Reps

At the University of Wisconsin-Madison

Recently, Sagiv, Reps, and Wilhelm introduced a powerful abstract-interpretation framework for program analysis based on three-valued logic [84]. Instantiations of this framework have been used to show a number of interesting properties of programs that manipulate a variety of linked data structures. However, two aspects of the framework represented significant challenges in its user-model. The work that is reported in this thesis addressed these two shortcomings, developed solutions to them, and carried out experiments to demonstrate their effectiveness.

The first aspect is the need to specify the set of *instrumentation relations*, which define the abstraction used in the analysis. This thesis presents a method that refines an abstraction automatically. Refinement is carried out by introducing new instrumentation relations (defined via logical formulas over core relations, which capture the basic properties of memory configurations). We present two strategies for refining an abstraction. The simpler strategy is effective in many cases. The second strategy uses a previously known machine-learning algorithm in a new way, namely, to learn an appropriate abstraction (by learning defining formulas for additional instrumentation relations). An advantage of our method is that it does not require the use of a theorem prover. The use of learning, in lieu of deduction-based techniques, constitutes a paradigm shift: the abstraction is constructed by observing (and generalizing) properties of memory configurations.

The second aspect is the need to specify *relation-maintenance* formulas, which describe how the effect of statements in the language (expressed using logical formulas that describe changes to core-relation values) can be reflected in the values of instrumentation relations. (These formulas

define the abstract transfer functions of the abstract semantics used for analyzing programs.) Manual creation of relation-maintenance formulas is a time-consuming and error-prone process. This thesis presents an algorithm to generate relation-maintenance formulas completely automatically. The algorithm is based on the principle of *finite differencing*, and transforms an instrumentation relation's defining formula into a relation-maintenance formula that captures what the instrumentation relation's new value should be.

ABSTRACT

Recently, Sagiv, Reps, and Wilhelm introduced a powerful abstract-interpretation framework for program analysis based on three-valued logic [84]. Instantiations of this framework have been used to show a number of interesting properties of programs that manipulate a variety of linked data structures. However, two aspects of the framework represented significant challenges in its user-model. The work that is reported in this thesis addressed these two shortcomings, developed solutions to them, and carried out experiments to demonstrate their effectiveness.

The first aspect is the need to specify the set of *instrumentation relations*, which define the abstraction used in the analysis. This thesis presents a method that refines an abstraction automatically. Refinement is carried out by introducing new instrumentation relations (defined via logical formulas over core relations, which capture the basic properties of memory configurations). We present two strategies for refining an abstraction. The simpler strategy is effective in many cases. The second strategy uses a previously known machine-learning algorithm in a new way, namely, to learn an appropriate abstraction (by learning defining formulas for additional instrumentation relations). An advantage of our method is that it does not require the use of a theorem prover. The use of learning, in lieu of deduction-based techniques, constitutes a paradigm shift: the abstraction is constructed by observing (and generalizing) properties of memory configurations.

The second aspect is the need to specify *relation-maintenance* formulas, which describe how the effect of statements in the language (expressed using logical formulas that describe changes to core-relation values) can be reflected in the values of instrumentation relations. (These formulas define the abstract transfer functions of the abstract semantics used for analyzing programs.) Manual creation of relation-maintenance formulas is a time-consuming and error-prone process. This thesis presents an algorithm to generate relation-maintenance formulas completely automatically.

The algorithm is based on the principle of *finite differencing*, and transforms an instrumentation relation's defining formula into a relation-maintenance formula that captures what the instrumentation relation's new value should be.

The framework of [84] has been implemented in the TVLA tool [54, 93]. We have extended TVLA with automatic abstraction refinement and finite differencing and applied it to a number of programs that manipulate (cyclic and acyclic) singly- and doubly-linked lists, binary trees, and binary-search trees. The tool was able to demonstrate a number of interesting properties, such as the partial correctness of the programs.

Additionally, this thesis reports on the automated verification of the *total correctness* (partial correctness and termination) of the Deutsch-Schorr-Waite (DSW) algorithm. DSW is an algorithm for traversing a binary tree without the use of a stack by means of destructive pointer manipulation. Prior approaches to the verification of the algorithm involved semi-automated applications of theorem provers or hand-written proofs. TVLA's abstract-interpretation approach made possible the automatic symbolic exploration of all memory configurations that can arise. With the introduction of a few simple core and instrumentation relations, TVLA was able to establish the partial correctness and termination of DSW.

Chapter 1

Introduction

A particularly challenging setting for the static analysis of software arises when trying to establish properties of programs that manipulate linked data structures, when these programs are written in a language that supports heap-allocated storage and destructive updating of address-valued fields. Recently, Sagiv, Reps, and Wilhelm introduced a novel approach, based on 3-valued logic, which addresses this setting [84]. Instantiations of the framework presented in [84], referred to here as SRW, have been used to show the safety of heap manipulation in programs [26], find opportunities for compile-time garbage collection [88], show the partial correctness of linked-list sorting procedures [53], ensure the correct usage of various Java interfaces, collections, and iterators [78, 96], and to show the partial correctness of tricky concurrent algorithms [95, 97].

In SRW, two related logics come into play: an ordinary 2-valued logic, as well as a related 3-valued logic. A memory configuration, or store, is modeled by what logicians call a *logical structure*; an individual of the structure's universe either models a single memory element or, in the case of a *summary individual*, it models a collection of memory elements. A run of the analyzer carries out an abstract interpretation to collect a set of structures at each program point P. This involves finding the least fixed point of a certain set of equations. When the fixed point is reached, the structures that have been collected at program point P describe a superset of all the execution states that can occur at P. To determine whether a query is always satisfied at P, one checks whether it holds in all of the structures that were collected there. Instantiations of this framework are capable of establishing nontrivial properties of programs that perform complex pointer-based manipulations of *a priori* unbounded-size heap-allocated data structures. The TVLA system (**T**hree-**V**alued-**L**ogic **A**nalyzer) implements this approach [54, 93].

Summary individuals play a crucial role. They are used to ensure that abstract descriptors have an *a priori* bounded size, which guarantees that a fixed-point is always reached. However, the constraint of working with limited-size descriptors implies a loss of information about the store. Intuitively, certain properties of concrete individuals are lost due to abstraction, which groups together multiple individuals into summary individuals: a property can be true for some concrete individuals of the group but false for other individuals. It is for this reason that 3-valued logic is used; uncertainty about a property's value is captured by means of the third truth value, $1/2$.

An advantage of using 2- and 3-valued logic as the basis for static analysis is that the language used for extracting information from the concrete world and the abstract world is identical: *every* syntactic expression—i.e., every logical formula—can be interpreted either in the 2-valued world or the 3-valued world. The consistency of the 2-valued and 3-valued viewpoints is ensured by a basic theorem that relates the two logics. Thus, formulas that define the concrete semantics, when interpreted in 2-valued logic, define a sound abstract semantics when interpreted in 3-valued logic.

Unfortunately, unless some care is taken in the design of an analysis, there is a danger that as abstract interpretation proceeds, the indefinite value $1/2$ will become pervasive. This can destroy the ability to recover interesting information from the 3-valued structures collected (although soundness is maintained). A key role in combating indefiniteness is played by *instrumentation relations*, which record auxiliary information in a logical structure. SRW annunciated the benefit of introducing instrumentation relations as the Instrumentation Principle:

Observation 1.1 (Instrumentation Principle [84, Observation 2.8]). Suppose that S is a 3-valued structure that represents the 2-valued structure $S^{\natural}$. By explicitly "storing" in S the values that a formula φ has in $S^{\natural}$, it is sometimes possible to extract more precise information from S than can be obtained just by evaluating φ in S.

Instrumentation relations provide a mechanism to fine-tune an abstraction: an instrumentation relation, which is defined by a logical formula φ over the core relation symbols, captures a property that an individual memory cell may or may not possess. In general, the introduction of additional

instrumentation relations (whose values are stored and maintained in response to the changes effected by program statements) refines an abstraction into one that is prepared to track finer distinctions among stores. This allows more properties of the program's stores to be identified.

As a means of verifying properties of programs, the advantages of the methodology of SRW are:

- No loop invariants are required.
- The methodology is based on abstract interpretation, and thus each run of the analysis must terminate.
- The methodology applies to static analyses based on 3-valued first-order logic, and hence it applies to programs that manipulate pointers and heap-allocated data structures, and eliminates the need for the user to write the usual proofs required for abstract interpretation—i.e., to demonstrate that the abstract structures that the analyzer manipulates correctly model the concrete heap-allocated data structures that the program manipulates.

The key challenge to wider applicability of SRW is the need to manually define instantiations of the framework that are capable of yielding a precise answer to a query for a given program. It is desirable to replace the need for user involvement with an automatic mechanism that is capable of refining the abstraction in an adaptive fashion. Such a mechanism requires:

1. a parameterizable analysis framework, i.e., the ability to change the underlying abstraction and abstract transformers,
2. the ability to create abstract transformers, given an abstraction (e.g., after refinement), and
3. the ability to select the next abstraction and to perform multiple iterations of the analysis.

While SRW addresses requirement 1, it includes no solution for requirements 2 and 3. Previously known techniques for automatic abstraction-refinement, such as counterexample-guided abstraction refinement (CEGAR) [4, 16, 22, 28, 36, 47, 49, 72], do not apply in our setting. A key difference between our setting and the CEGAR approach is the abstract domain: prior work has used abstract

domains that are fixed, finite, Cartesian products of Boolean values (i.e., predicate-abstraction domains), and hence the only relations introduced are nullary relations. Our work applies to a richer class of abstractions—3-valued structures—that generalize predicate-abstraction domains. A solution to requirements 2 and 3 needs to be able to introduce unary, binary, ternary, etc. relations (together with appropriate transformers), in addition to nullary relations. A second distinguishing feature of our setting is that theorem provers have limited applicability. There do not currently exist theorem provers for first-order logic extended with transitive closure capable of identifying infeasible counterexample traces [38]; hence we needed to develop techniques different from those used in tools such as SLAM, BLAST, etc.

1.1 Our Solution and Organization of the Thesis

Chapter 2 introduces terminology and notation; it presents the logic that we employ and describes the use of logical structures for representing memory stores.

Chapter 3 addresses requirement 2, which is an instance of the following fundamental challenge in applying abstract interpretation:

> Given the concrete semantics for a language and a desired abstraction, how does one create the associated abstract transformers?

In our context, the semantics of statements is expressed using logical formulas that describe changes to core-relation values. When instrumentation relations have been introduced to refine an abstraction, the challenge is to reflect the changes in core-relation values in the values of the instrumentation relations [3,23,32,61,84]. The algorithm presented in Chapter 3 provides a way to create formulas that maintain correct values for the instrumentation relations, and thereby provides a way to generate, completely automatically, the part of the transformers of an abstract semantics that deals with instrumentation relations. The algorithm runs in time linear in the size of the instrumentation relation's defining formula. This research was motivated by work on static analysis based on 3-valued logic (SRW); however, any analysis method that relies on logic—2-valued or 3-valued—to express a program's semantics may be able to benefit from these techniques.

From the standpoint of the concrete semantics, instrumentation relations represent cached information that could always be recomputed by reevaluating the instrumentation relation's defining formula in the local state. From the standpoint of the abstract semantics, however, reevaluating a formula in the local (3-valued) state can lead to a drastic loss of precision. To gain maximum benefit from instrumentation relations, an abstract-interpretation algorithm must obtain their values in some other way.

This problem, *instrumentation-relation maintenance*, is solved by incremental computation. The new value that instrumentation relation p should have after a transition via abstract state transformer τ from state σ to σ' is computed incrementally from the known value of p in σ. The contributions of the work reported in Chapter 3 can be summarized as follows:

- We give an algorithm for the relation-maintenance problem; it creates a relation-maintenance formula by applying a finite-differencing transformation to p's defining formula. The algorithm runs in time linear in the size of the defining formula.

- We present experimental evidence that our technique is an effective one, at least for the analysis of programs that manipulate (cyclic and acyclic) singly-linked lists, doubly-linked lists, and binary trees, and for certain sorting programs. In particular, the relation-maintenance formulas produced automatically using our approach are as effective for maintaining precision as the best available hand-crafted ones.

- This work is related to the view-maintenance problem in databases. Compared with that work, the novelty is the ability to create relation-maintenance formulas that are suitable for use when abstraction has been performed.

Chapter 4 presents *inductive logic programming* [50, 68, 69, 75, 77], [65, §10], a machine-learning technique that is instrumental in our solution to requirement 3 (which itself is the subject of Chapter 5). Chapter 4 describes an existing algorithm that implements the technique, discusses some implications of using the algorithm in the setting of 3-valued logic, and presents some extensions that we implemented while adapting the algorithm to the need that is articulated in requirement 3.

Chapter 5 addresses requirement 3, which is an instance of the following fundamental challenge in applying abstract interpretation:

> Given a program and a query of interest, how does one create an abstraction that is sufficiently precise to verify that the program satisfies the query?

We introduce an approach to creating abstractions automatically that, as in some previous work [4, 16, 22, 36, 47, 49, 72], involves the successive refinement of the abstraction in use. Unlike previous work, the work presented in this thesis is aimed at the analysis of programs that manipulate pointers and heap-allocated data structures (also known as *shape analysis*). However, while we demonstrate our approach on shape-analysis problems, the approach is applicable in any program-analysis setting that uses first-order logic.

Chapter 5 presents an abstraction-refinement method for use in static analyses based on 3-valued logic (SRW), where the semantics of statements and the query of interest are expressed using logical formulas. Refinement is performed by introducing new instrumentation relations (defined via logical formulas over core relations, which capture the basic properties of memory configurations). Our abstraction-refinement method uses two refinement strategies. The first strategy, *subformula-based refinement*, analyzes the sources of imprecision in the evaluation of the query, and chooses how to define new instrumentation relations using subformulas of the query. The second strategy, *ILP-based refinement*, employs inductive logic programming (ILP) to learn new instrumentation relations that can stave off imprecision due to abstraction. The steps of ILP go beyond merely forming Boolean combinations of existing relations (as in many previously introduced refinement techniques); ILP can create new relations by introducing quantifiers during the learning process.

The choice of instrumentation relations is crucial to the precision, as well as the cost, of the analysis. Until now, TVLA users have been faced with the task of identifying an instrumentation-relation set that gives them a definite answer to the query, but does not make the cost prohibitive. This was arguably one of the key remaining challenges in the TVLA user-model. The contributions of the work described in Chapter 5 can be summarized as follows:

- It establishes a new connection between program analysis and machine learning by showing that ILP is relevant to the problem of creating abstractions automatically. We use ILP for learning new instrumentation relations that preserve information that would otherwise be lost due to abstraction.

- The method has been implemented as an extension of TVLA. The input required to specify a program analysis consists of: (i) a transition system, (ii) a query (a formula that identifies acceptable outputs), and (iii) a characterization of the program's valid inputs. Only the latter has a somewhat nonstandard form: the valid inputs must be specified by giving a non-deterministic program (a transition system) that generates valid inputs (starting from scratch). These "Data-Structure Constructors" are discussed in Sect. 5.2.4.

- We present experimental evidence that the use of this approach in an iterative abstraction-refinement loop can yield precise answers to queries. We tested the effectiveness of the method using sortedness, stability, and antistability queries on a collection of programs that perform destructive list manipulation, as well as by using it to establish partial correctness of two binary-search-tree programs and total correctness of an *in-situ* list-reversal program when applied to possibly-cyclic lists. The method is successful in all cases tested here.

Inductive learning concerns identifying general rules from a set of observed instances—in our case, from relationships observed in a logical structure. An advantage of an approach based on inductive learning is that it does not require the use of a theorem prover. This is particularly beneficial in our setting because our logic is undecidable.

Chapter 6 discusses the Deutsch-Schorr-Waite (DSW) algorithm. DSW provides a way to traverse a tree without the use of a stack by temporarily—but systematically—stealing pointer fields of the tree's nodes to serve in place of the stack that one ordinarily needs during, e.g., an in-order tree traversal. The subtlety of the algorithm (and the complexity of analyzing it) is due to the fact that, during the traversal, the algorithm visits each node of the tree three times, and performs a kind of pointer rotation on each node visit [56]. By the time the algorithm finishes, it

has restored the original values of each node's left-child and right-child pointers, thus restoring the original tree.

Richard Bornat singles out the algorithm as a key test for formal methods: "The [Deutsch-]Schorr-Waite algorithm is the first mountain that any formalism for pointer analysis should climb." [9] Past approaches have involved hand-written proofs of complicated invariants to verify the partial correctness of the algorithm. Even with some automation, these efforts were usually laborious: a proof performed in 2002 with the help of the Jape proof editor took 152 pages! [8] The key advantage of TVLA's abstract-interpretation approach over proof-theoretic approaches is that a relatively small number of concepts are involved in defining an abstraction of the structures that can arise on any execution, and verification is then carried out automatically by symbolic exploration of all memory configurations that can arise.

Our initial intention was to apply our abstraction-refinement approach to automatically create an abstraction that could be used to verify the correctness of the algorithm. However, DSW's complexity made it very challenging to verify the algorithm even using a manually-identified abstraction. Thus, although this represents only a partial victory for our techniques—finite differencing was employed to automatically create the relation-maintenance formulas, but abstraction refinement was not used to learn the appropriate abstraction—the verification of the algorithm's correctness constituted a standalone contribution, which was worthy of inclusion in this thesis. The contributions of the work reported in Chapter 6 can be summarized as follows:

- We defined an abstraction (in the canonical-abstraction framework used by TVLA) that captures sufficient invariants of DSW to demonstrate partial correctness and termination. We defined the abstraction using a few simple instrumentation relations—eight key formulas—each containing only two atomic subformulas.

- We used the fact that each tree node passes through four states (induced by the original state and the three visits to each node) to define a *state-dependent* abstraction, which requires fewer structures to represent the memory configurations that can arise in DSW than would be necessary without state dependence.

- We used the abstraction to establish the partial correctness of DSW via automatic symbolic exploration of all memory configurations.
- We used the *state-dependent* abstraction to establish that a certain measure strictly decreases on each loop iteration, thus establishing that DSW terminates.

While studying the question of precision in analyses based on 3-valued logic, we encountered a non-standard logic-minimization problem that arises in 3-valued propositional logic. Chapter 7 presents our results in regards to this problem. To illustrate the issue, consider the following trivial example (where we use $[\![\varphi]\!](a)$ to denote the value of a formula φ with respect to an assignment a of truth values to propositional variables): In 2-valued logic, the formula $p \vee \neg p$ is equivalent to the formula $\mathbf{1}$; that is, in 2-valued logic, $[\![p \vee \neg p]\!](a) = [\![\mathbf{1}]\!](a) = 1$ for all assignments a that assign some truth value to p. In contrast, $p \vee \neg p$ and $\mathbf{1}$ are not equivalent in 3-valued logic, as can be seen by considering their values under various (3-valued) assignments:

$$\begin{array}{lcccl} [\![\mathbf{1}]\!]([p \mapsto 0]) & = & 1 & & = [\![p \vee \neg p]\!]([p \mapsto 0]) \\ [\![\mathbf{1}]\!]([p \mapsto 1/2]) & = & 1 & \neq 1/2 & = [\![p \vee \neg p]\!]([p \mapsto 1/2]) \\ [\![\mathbf{1}]\!]([p \mapsto 1]) & = & 1 & & = [\![p \vee \neg p]\!]([p \mapsto 1]) \end{array}$$

In particular, for $[p \mapsto 1/2]$, the formula $\mathbf{1}$ provides a definite answer (i.e., 1), but $p \vee \neg p$ provides an indefinite answer (i.e., 1/2).

As this example demonstrates, in 3-valued logic there is a notion of one formula being "better" than another: among the formulas that are equivalent in 2-valued logic, some may evaluate to a definite value on more 3-valued assignments. Our interest in this phenomenon is motivated by the possibility of exploiting it to obtain better answers in applications that use 3-valued logic. An answer of 0 or 1 provides precise (definite) information; an answer of $1/2$ provides imprecise (indefinite) information. By replacing a formula φ with a "better" formula ψ, we may improve the precision of the answers obtained.

One might approach the problem of "improving" φ by simplifying φ's subterms using rewriting rules, such as

$$\gamma \vee \neg\gamma \longrightarrow \mathbf{1} \qquad\qquad \gamma \wedge \neg\gamma \longrightarrow \mathbf{0}.$$

However, one is left with the question of whether such an approach always produces a formula that is as good as possible.

In Chapter 7, we give an algorithm that uses a different approach; the algorithm always produces a formula that is "best" (in a certain well-defined sense). This work makes the following contributions:

- We provide a formalization of the "semantic-minimization" problem: Given a formula φ, the goal is to find a best formula ψ. (See Sect. 7.2.)
- We show that one can always find a best formula.
- We present several methods for creating a best formula.

Chapter 8 presents some concluding remarks on the lessons learned and possible future directions for this line of research.

Finally, the Appendix presents proofs of certain propositions. App. A presents a proof of the correctness of our solution to requirement 2 and App. B presents proofs of propositions stated in Chapter 7.

Chapter 2

Background

The present chapter introduces terminology and notation; it presents the logic that we employ and describes the use of logical structures for representing memory stores.

The first half of Sect. 2.1 introduces 2-valued first-order logic with transitive closure. These concepts are standard in logic. The latter half of the section presents a straightforward extension of the logic to the 3-valued setting, in which a third truth value—$1/2$—is introduced to denote uncertainty. The remainder of the chapter summarizes the program-analysis framework described in [84]. In that approach, memory configurations are encoded as logical structures, the semantics of programs, as well as the properties of memory configurations, is encoded as logical formulas, and abstract interpretation computes the set of logical structures that describe the memory configurations that can arise at each point in the program being analyzed.

2.1 First-Order Logic with Transitive Closure

2-Valued First-Order Logic with Transitive Closure The syntax of first-order formulas with equality and reflexive transitive closure is defined as follows:

Definition 2.1 Let R_i denote a set of arity-i relation symbols. A *formula* over the *vocabulary* $\mathcal{R} = \{eq\} \cup \bigcup_i R_i$ is defined by

$$
\begin{array}{lll}
p \in \mathcal{R}_k & \varphi ::= & \mathbf{0} \mid \mathbf{1} \mid p(v_1, \ldots, v_k) \\
\varphi \in \textit{Formulas} & & \mid (\neg\varphi_1) \mid (\varphi_1 \wedge \varphi_2) \mid (\varphi_1 \vee \varphi_2) \mid (\exists v\colon \varphi_1) \mid (\forall v\colon \varphi_1) \\
v \in \textit{Variables} & & \mid (\mathbf{RTC}\ v_1', v_2'\colon \varphi_1)(v_1, v_2)
\end{array}
$$

The set of free variables of a formula is defined as usual. "**RTC**" stands for reflexive transitive closure. In $\varphi \equiv (\textbf{RTC}\ v_1', v_2' \colon \varphi_1)(v_1, v_2)$, if φ_1's free-variable set is V, we require $v_1, v_2 \notin V$. The free variables of φ are $(V - \{v_1', v_2'\}) \cup \{v_1, v_2\}$.

We use several shorthand notations: $(v_1 = v_2) \stackrel{\text{def}}{=} eq(v_1, v_2)$; $(v_1 \neq v_2) \stackrel{\text{def}}{=} \neg eq(v_1, v_2)$; and for a binary relation p, $p^*(v_1, v_2) \stackrel{\text{def}}{=} (\textbf{RTC}\ v_1', v_2' \colon p(v_1', v_2'))(v_1, v_2)$. We also use a C-like syntax for conditional expressions: $\varphi_1\ ?\ \varphi_2\ :\ \varphi_3$.[1] The order of precedence among the connectives, from highest to lowest, is as follows: $\neg$, $\wedge$, $\vee$, $\forall$, and $\exists$. We drop parentheses wherever possible, except for emphasis.

Definition 2.2 A *2-valued interpretation* over $\mathcal{R}$ is a *2-valued logical structure* $S = \langle U^S, \iota^S \rangle$, where U^S is a set of *individuals* and ι^S maps each relation symbol p of arity k to a truth-valued function: $\iota^S(p) \colon (U^S)^k \to \{0, 1\}$. In addition, (i) for all $u \in U^S$, $\iota^S(eq)(u, u) = 1$, and (ii) for all $u_1, u_2 \in U^S$ such that u_1 and u_2 are distinct individuals, $\iota^S(eq)(u_1, u_2) = 0$.

An *assignment* Z is a function that maps variables to individuals (i.e., it has the functionality $Z \colon \{v_1, v_2, \ldots\} \to U^S$). When Z is defined on all free variables of a formula φ, we say that Z is *complete* for φ. (We generally assume that every assignment that arises in connection with the discussion of some formula φ is complete for φ.)

The *(2-valued) meaning* of a formula φ, denoted by $[\![\varphi]\!]_2^S(Z)$, yields a truth value in $\{0, 1\}$; it is defined inductively as follows:

[1] In 2-valued logic, one can think of $\varphi_1\ ?\ \varphi_2\ :\ \varphi_3$ as a shorthand for $(\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \varphi_3)$. In 3-valued logic, it becomes a shorthand for $(\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \varphi_3) \vee (\varphi_2 \wedge \varphi_3)$, as explained in Chapter 7.

$$
\begin{array}{rcl@{\qquad}rcl}
[\![\mathbf{0}]\!]_2^S(Z) & = & 0 & [\![\varphi_1 \wedge \varphi_2]\!]_2^S(Z) & = & \min([\![\varphi_1]\!]_2^S(Z), [\![\varphi_2]\!]_2^S(Z)) \\
[\![\mathbf{1}]\!]_2^S(Z) & = & 1 & [\![\varphi_1 \vee \varphi_2]\!]_2^S(Z) & = & \max([\![\varphi_1]\!]_2^S(Z), [\![\varphi_2]\!]_2^S(Z)) \\
[\![p(v_1, \ldots, v_k)]\!]_2^S(Z) & = & \iota^S(p)(Z(v_1), \ldots, Z(v_k)) & [\![\exists v\colon \varphi_1]\!]_2^S(Z) & = & \max\limits_{u \in U^S} [\![\varphi_1]\!]_2^S(Z[v_1 \mapsto u]) \\
[\![\neg\varphi_1]\!]_2^S(Z) & = & 1 - [\![\varphi_1]\!]_2^S(Z) & [\![\forall v\colon \varphi_1]\!]_2^S(Z) & = & \min\limits_{u \in U^S} [\![\varphi_1]\!]_2^S(Z[v_1 \mapsto u])
\end{array}
$$

$$
[\![(\mathbf{RTC}\ v_1', v_2'\colon \varphi_1)(v_1, v_2)]\!]_2^S(Z)
$$

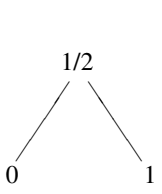
$$
= \begin{cases}
1 & \text{if } Z(v_1) = Z(v_2) \\
\max\limits_{\substack{n \geq 1, \\ u_1, \ldots, u_{n+1} \in U, \\ Z(v_1) = u_1, \\ Z(v_2) = u_{n+1}}} \min\limits_{i=1}^{n} [\![\varphi_1]\!]_2^S(Z[v_1' \mapsto u_i, v_2' \mapsto u_{i+1}]) & \text{otherwise}
\end{cases}
$$

S and Z *satisfy* φ if $[\![\varphi]\!]_2^S(Z) = 1$. The set of 2-valued structures is denoted by 2-STRUCT$[\mathcal{R}]$.

3-Valued Logic and Embedding In 3-valued logic, the formulas that we work with are identical to the ones used in 2-valued logic. At the semantic level, a third truth value—$1/2$—is introduced to denote uncertainty.

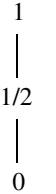
Definition 2.3 The truth values 0 and 1 are *definite values*; $1/2$ is an *indefinite value*. For $l_1, l_2 \in \{0, 1/2, 1\}$, the *information order* is defined as follows: $l_1 \sqsubseteq l_2$ iff $l_1 = l_2$ or $l_2 = 1/2$. $l_1 \sqsubseteq l_2$ denotes that l_1 is at least as definite as l_2. We use $l_1 \sqsubset l_2$ when $l_1 \sqsubseteq l_2$ and $l_1 \neq l_2$. The symbol $\sqcup$ denotes the least-upper-bound operation with respect to $\sqsubseteq$.

As shown in Fig. 2.1, we place two orderings on 0, 1, and $1/2$: (i) the *information order*, denoted by $\sqsubseteq$ and illustrated in Fig. 2.1(a), captures "(un)certainty"; (ii) the *logical order*, shown in Fig. 2.1(b), defines the meaning of $\wedge$ and $\vee$; that is, $\wedge$ and $\vee$ are meet and join in the logical order. 3-valued logic retains a number of properties that are familiar from 2-valued logic, such as De Morgan's laws, associativity of $\wedge$ and $\vee$, and distributivity of $\wedge$ over $\vee$ (and vice versa). Because $\varphi_1 \,?\, \varphi_2 : \varphi_3$ is treated as a shorthand for $(\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \varphi_3) \vee (\varphi_2 \wedge \varphi_3)$ in 3-valued logic (see Chapter 7), the value of $1/2 \,?\, V_1 : V_2$ equals $V_1 \sqcup V_2$. We now generalize Defn. 2.2 to define the meaning of a formula with respect to a 3-valued structure.



$\sqcup$	0	1/2	1
0	0	1/2	1/2
1/2	1/2	1/2	1/2
1	1/2	1/2	1

(a)



$\vee$	0	1/2	1
0	0	1/2	1
1/2	1/2	1/2	1
1	1	1	1

(b)

Figure 2.1 (a) The information order ($\sqsubseteq$) and its join operation ($\sqcup$). (b) The logical order and its join operation ($\vee$).

Definition 2.4 A *3-valued interpretation* over $\mathcal{R}$ is a *3-valued logical structure* $S = \langle U^S, \iota^S \rangle$, where U^S is a set of individuals and ι^S maps each relation symbol p of arity k to a truth-valued function: $\iota^S(p)\colon (U^S)^k \rightarrow \{0, 1/2, 1\}$. In addition, (i) for all $u \in U^S$, $\iota^S(eq)(u, u) \sqsupseteq 1$, and (ii) for all $u_1, u_2 \in U^S$ such that u_1 and u_2 are distinct individuals, $\iota^S(eq)(u_1, u_2) = 0$.

For an assignment Z, the *(3-valued) meaning* of a formula φ, denoted by $[\![\varphi]\!]_3^S(Z)$, yields a truth value in $\{0, 1/2, 1\}$. The meaning of φ is defined exactly as in Defn. 2.2, but interpreted over $\{0, 1/2, 1\}$. S and Z *potentially satisfy* φ if $[\![\varphi]\!]_3^S(Z) \sqsupseteq 1$. The set of 3-valued structures is denoted by 3-STRUCT$[\mathcal{R}]$.

Defn. 2.4 requires that for each individual u, the value of $\iota^S(eq)(u, u)$ is 1 or $1/2$. An individual for which $\iota^S(eq)(u, u) = 1/2$ is called a *summary individual*. In the abstract-interpretation context, a summary individual abstracts one or more fragments of a data structure, and can represent more than one concrete memory cell.

The embedding ordering on structures is defined as follows:

Definition 2.5 Let $S = \langle U^S, \iota^S \rangle$ and $S' = \langle U^{S'}, \iota^{S'} \rangle$ be two structures, and let $f\colon U^S \rightarrow U^{S'}$ be a surjective function. We say that f *embeds* S in S' (denoted by $S \sqsubseteq^f S'$) if for every relation symbol $p \in \mathcal{R}$ of arity k and for all $u_1, \ldots, u_k \in U^S$, $\iota^S(p)(u_1, \ldots, u_k) \sqsubseteq \iota^{S'}(p)(f(u_1), \ldots, f(u_k))$. We say that S *can be embedded in* S' (denoted by $S \sqsubseteq S'$) if there exists a function f such that $S \sqsubseteq^f S'$.

The Embedding Theorem says that if $S \sqsubseteq^f S'$, then every piece of information extracted from S' via a formula φ is a conservative approximation of the information extracted from S via φ. To

formalize this, we extend mappings on individuals to operate on assignments: if $f\colon U^S \to U^{S'}$ is a function and $Z\colon Var \to U^S$ is an assignment, $f \circ Z$ denotes the assignment $f \circ Z\colon Var \to U^{S'}$ such that $(f \circ Z)(v) = f(Z(v))$.

Theorem 2.6 (Embedding Theorem [84, Theorem 4.9]). Let $S = \langle U^S, \iota^S \rangle$ and $S' = \langle U^{S'}, \iota^{S'} \rangle$ be two structures, and let $f\colon U^S \to U^{S'}$ be a function such that $S \sqsubseteq^f S'$. Then, for every formula φ and complete assignment Z for φ, $[\![\varphi]\!]^S_3(Z) \sqsubseteq [\![\varphi]\!]^{S'}_3(f \circ Z)$.

2.2 Stores as Logical Structures and their Abstractions

Program Analysis Via 3-Valued Logic The remainder of this chapter summarizes the program-analysis framework described in [84]. In that approach, concrete memory configurations (i.e., *stores*) are encoded as logical structures (associated with a *vocabulary* of relation symbols with given arities) in terms of a fixed collection of *core relations*, $\mathcal{C}$. Core relations are part of the underlying semantics of the language to be analyzed; they record atomic properties of stores. For instance, Fig. 2.3 gives the definition of a C linked-list datatype, and lists the relations that would be used to represent the stores manipulated by programs that use type `List`, such as the store in Fig. 2.2. (The core relations are fixed for a given combination of language and datatype; in general, different languages and datatypes require different collections of core relations.) 2-valued logical structures then represent memory configurations: the individuals are the set of memory cells; a nullary relation represents a Boolean variable of the program; a unary relation represents either a pointer variable or a Boolean-valued field of a record; and a binary relation represents a pointer field of a record. In this example, unary relations represent pointer variables and binary relation n represents the `n`-field of a `List` cell. Numeric-valued variables and numeric-valued fields (such as `data`) can be modeled by introducing other relations, such as the binary relation dle (which stands for "`data` less-than-or-equal-to") listed in Fig. 2.3; dle captures the relative order of two nodes' `data` values. (Alternatively, numeric-valued entities can be handled by combining

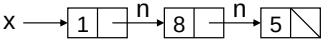


Figure 2.2 A possible store for a linked list

```
typedef struct node {
  struct node *n;
  int data;
} *List;
```

(a)

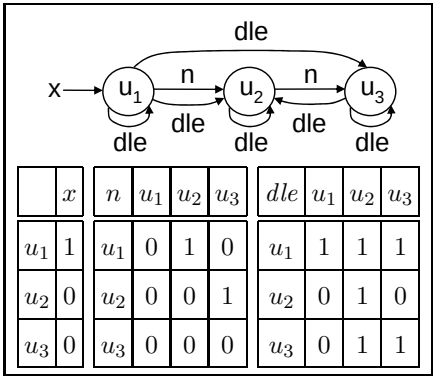
Relation	Intended Meaning
$eq(v_1, v_2)$	Do v_1 and v_2 denote the same memory cell?
$x(v)$	Does pointer variable `x` point to memory cell v?
$n(v_1, v_2)$	Does the `n` field of v_1 point to v_2?
$dle(v_1, v_2)$	Is the `data` field of v_1 less than or equal to that of v_2?

(b)

Figure 2.3 (a) Declaration of a linked-list datatype in C. (b) Core relations used for representing the stores manipulated by programs that use type `List`.

abstractions of logical structures with previously known techniques for creating numeric abstractions [31].) Fig. 2.4 shows 2-valued structure $S_{2.4}$, which represents the store of Fig. 2.2 using the relations of Fig. 2.3. $S_{2.4}$ has three nodes, u_1, u_2, and u_3, which represent the three list elements.

Information can be extracted from logical structures by evaluating formulas. A concrete operational semantics is defined by specifying, for each kind of statement st in the programming language, a structure transformer for each outgoing control-flow graph (CFG) edge $e = (st, st')$. A structure transformer is specified by providing a collection of *relation-transfer formulas*, $\tau_{c,st}$, one for each core relation c. These formulas define how the core relations of a 2-valued logical structure S that arises at st are transformed by e to create a 2-valued logical structure S' at st'; typically, they define the value of relation c in S' as a function of c's value in S. Edge e may optionally have a *precondition formula*, which filters out structures that should not follow the transition along e. The postcondition operator $post$ for edge e is defined by lifting e's structure transformer to sets of structures.



	x
u_1	1
u_2	0
u_3	0

n	u_1	u_2	u_3
u_1	0	1	0
u_2	0	0	1
u_3	0	0	0

dle	u_1	u_2	u_3
u_1	1	1	1
u_2	0	1	0
u_3	0	1	1

Figure 2.4 A logical structure $S_{2.4}$ that represents the store shown in Fig. 2.2 in graphical and tabular forms using the relations of Fig. 2.3 (Relation eq is not shown explicitly; each node has an eq self-loop, and the relation in tabular form is the identity matrix.)

Abstract stores are 3-valued logical structures. Concrete stores are abstracted to abstract stores by means of *embedding functions*—onto functions that map individuals of a 2-valued structure $S^\natural$

to those of a 3-valued structure S. The Embedding Theorem ensures that every piece of information extracted from S by evaluating a formula φ is a conservative approximation ($\sqsupseteq$) of the information extracted from $S^\natural$ by evaluating φ.

To obtain a computable abstract domain, we need a way to ensure that the 3-valued structures used to represent memory configurations are always of finite size. We do this by defining an equivalence relation on individuals and considering the (finite) quotient structure with respect to this equivalence relation; in particular, each individual of a 2-valued logical structure (representing a concrete memory cell) is mapped to an individual of a 3-valued logical structure according to the vector of values that the concrete individual has for a user-chosen collection of unary abstraction relations:

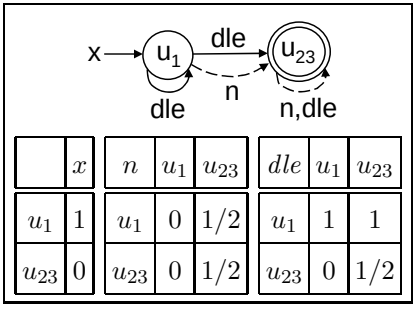
Definition (Canonical Abstraction). Let $S \in$ 2-STRUCT$[\mathcal{R}]$, and let $\mathcal{A} \subseteq \mathcal{R}_1$ be some chosen subset of the unary relation symbols. The relations in $\mathcal{A}$ are called *abstraction relations*; they define the following equivalence relation $\simeq_{\mathcal{A}}$ on U^S:

$$u_1 \simeq_{\mathcal{A}} u_2 \iff \text{for all } p \in \mathcal{A}, p^S(u_1) = p^S(u_2),$$

and the surjective function $f_{\mathcal{A}} : U^S \to U^S/\simeq_{\mathcal{A}}$, such that $f_{\mathcal{A}}(u) = [u]_{\simeq_{\mathcal{A}}}$, which maps an individual to its equivalence class. The *canonical abstraction* of S with respect to $\mathcal{A}$ (denoted by $f_{\mathcal{A}}(S)$) performs the join (in the information order) of relation values, thereby introducing $1/2$'s.

Intuitively, canonical abstraction maps a group of individuals that are indistinguishable according to the set of (unary) abstraction relations $\mathcal{A}$ to a single individual.

If all unary relations are abstraction relations ($\mathcal{A} = \mathcal{R}_1$), the canonical abstraction of 2-valued logical structure $S_{2.4}$ is $S_{2.5}$, shown in Fig. 2.5, with $f_{\mathcal{A}}(u_1) = u_1$ and $f_{\mathcal{A}}(u_2) = f_{\mathcal{A}}(u_3) = u_{23}$. In addition to $S_{2.4}$, $S_{2.5}$ represents any list with two or more elements that is pointed to by program variable `x`, and in which the first element's `data` value is (definitely) lower than the `data` values in the rest of the list (note the absence of either a 1-valued or $1/2$-valued dle



	x
u_1	1
u_{23}	0

n	u_1	u_{23}
u_1	0	1/2
u_{23}	0	1/2

dle	u_1	u_{23}
u_1	1	1
u_{23}	0	1/2

Figure 2.5 A 3-valued structure $S_{2.5}$ that is the canonical abstraction of structure $S_{2.4}$

edge from individual u_{23} to individual u_1). The following graphical notation is used for depicting 3-valued logical structures:

- Individuals are represented by circles containing their names and (non-0) values for unary relations. Summary individuals are represented by double circles.
- A unary relation p corresponding to a pointer-valued program variable is represented by a solid arrow from p to the individual u for which $p(u) = 1$, and by the absence of a p-arrow to each node u' for which $p(u') = 0$. (If $p = 0$ for all individuals, the relation name p is not shown.)
- A binary relation q is represented by a solid arrow labeled q between each pair of individuals u_i and u_j for which $q(u_i, u_j) = 1$, and by the absence of a q-arrow between pairs u'_i and u'_j for which $q(u'_i, u'_j) = 0$.
- Relations with value $1/2$ are represented by dotted arrows.

Canonical abstraction ensures that each 3-valued structure is no larger than some fixed size, known *a priori*.

2.2.1 Instrumentation Relations

The abstraction function on which an analysis is based, and hence the precision of the analysis defined, can be tuned by (i) choosing to equip structures with additional *instrumentation relations* to record derived properties, and (ii) varying which of the unary core and unary instrumentation relations are used as the set of abstraction relations. The set of instrumentation relations is denoted by $\mathcal{I}$. Each arity-k relation symbol $p \in \mathcal{I}$ is defined by an *instrumentation-relation definition formula* $\psi_p(v_1, \ldots, v_k)$. Instrumentation relations may appear in the defining formulas of other instrumentation relations as long as there are no circular dependences.

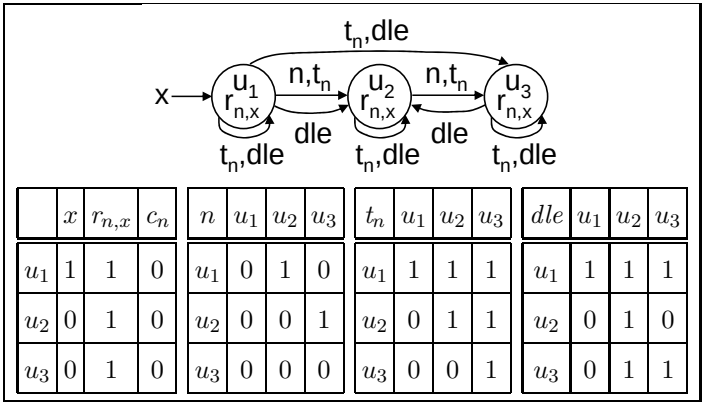
The introduction of unary instrumentation relations that are used as abstraction relations provides a way to control which concrete individuals are merged together into an abstract individual, and thereby control the amount of information lost by abstraction. Instrumentation relations that

p	**Intended Meaning**	ψ_p
$is_n(v)$	Do n fields of two or more list nodes point to v?	$\exists\, v_1, v_2 \colon n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$
$t_n(v_1, v_2)$	Is v_2 reachable from v_1 along zero or more n fields?	$n^*(v_1, v_2)$
$r_{n,x}(v)$	Is v reachable from pointer variable x along zero or more n fields?	$\exists\, v_1 \colon x(v_1) \wedge t_n(v_1, v)$
$c_n(v)$	Is v on a directed cycle of n fields?	$\exists\, v_1 \colon n(v_1, v) \wedge t_n(v, v_1)$

Figure 2.6 Defining formulas of some commonly used instrumentation relations. The relation name is_n abbreviates "is-shared". There is a separate reachability relation $r_{n,x}$ for every program variable x. (Recall that $v_1 \neq v_2$ is a shorthand for $\neg eq(v_1, v_2)$, and $n^*(v_1, v_2)$ is a shorthand for $(\mathbf{RTC}\ v_1', v_2' \colon n(v_1', v_2'))(v_1, v_2)$.)

involve reachability properties, which can be defined using **RTC**, often play a crucial role in the definitions of abstractions. For instance, in program-analysis applications, reachability properties from specific pointer variables have the effect of keeping disjoint sublists summarized separately. Fig. 2.6 lists some instrumentation relations that are important for the analysis of programs that use type `List`.

Fig. 2.7 shows 2-valued structure $S_{2.7}$, which represents the store of Fig. 2.2 using the core relations of Fig. 2.3, as well as the instrumentation relations of Fig. 2.6. If all unary relations are abstraction relations ($\mathcal{A} = \mathcal{R}_1$), the canonical abstraction of 2-valued logical structure $S_{2.7}$ is $S_{2.8}$, shown in Fig. 2.8, with $f_{\mathcal{A}}(u_1) = u_1$ and $f_{\mathcal{A}}(u_2) = f_{\mathcal{A}}(u_3) = u_{23}$.



	x	$r_{n,x}$	c_n
u_1	1	1	0
u_2	0	1	0
u_3	0	1	0

n	u_1	u_2	u_3
u_1	0	1	0
u_2	0	0	1
u_3	0	0	0

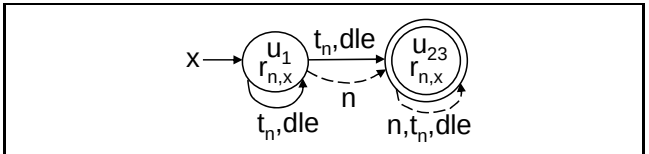
t_n	u_1	u_2	u_3
u_1	1	1	1
u_2	0	1	1
u_3	0	0	1

dle	u_1	u_2	u_3
u_1	1	1	1
u_2	0	1	0
u_3	0	1	1

Figure 2.7 A logical structure $S_{2.7}$ that represents the store shown in Fig. 2.2 in graphical and tabular forms using the relations of Figs. 2.3 and 2.6

2.2.2 History Relations

We are sometimes interested in making assertions that compare the state of a store at the end of a procedure with its state at the start. For instance, we may be interested in checking that all list elements reachable from variable `x` at the start of a procedure are guaranteed to be reachable from `x` at the end. To allow the user to make such assertions, we double the vocabulary: for each relation p, we extend the program-analysis specification with a *history* relation, p^0, which serves as an indelible record of the state of the store at the entry point. We will use the term history relations to refer



	x	$r_{n,x}$	c_n
u_1	1	1	0
u_{23}	0	1	0

n	u_1	u_{23}
u_1	0	1/2
u_{23}	0	1/2

t_n	u_1	u_{23}
u_1	1	1
u_{23}	0	1/2

dle	u_1	u_{23}
u_1	1	1
u_{23}	0	1/2

Figure 2.8 A 3-valued structure $S_{2.8}$ that is the canonical abstraction of structure $S_{2.7}$

to the latter kind of relations, and the term *active* relations to refer to the relations from the original vocabulary. We can now express the property mentioned above:

$$\forall v : r^0_{n,x}(v) \Leftrightarrow r_{n,x}(v). \tag{2.1}$$

If Formula (2.1) evaluates to 1, then the elements reachable from `x` after the procedure executes are exactly the same as those reachable at the beginning of the procedure, and consequently the procedure performs a permutation of list `x`.

In addition to history relations, we introduce a collection of nullary instrumentation relations that track whether active relations have changed from their initial values. For each active relation $p(v_1, \ldots, v_k)$, the relation $same_p()$ is defined by $\psi_{same_p} = \forall v_1, \ldots, v_k : p(v_1, \ldots, v_k) \Leftrightarrow p^0(v_1, \ldots, v_k)$. We can now use $same_{r_{n,x}}()$ in place of Formula (2.1) when asserting the permutation property.

2.2.3 Abstract Interpretation

For each kind of statement in the programming language, the abstract semantics is again defined by a collection of formulas: the *same* relation-transfer formula that defines the concrete semantics, in the case of a core relation, and, in the case of an instrumentation relation p, by a *relation-maintenance formula* $\mu_{p,st}$.[2]

Abstract interpretation collects a set of 3-valued structures at each program point. It can be implemented as an iterative procedure that finds the least fixed point of a certain set of equations [84]. (It is important to understand that although the analysis framework is based on logic, it is model theoretic, not proof theoretic: the abstract interpretation collects sets of 3-valued logical structures—i.e., abstracted models; its actions do not rely on deduction or theorem proving.) When the fixed point is reached, the structures that have been collected at program point P describe a

[2] In [84], relation-transfer formulas and relation-maintenance formulas are both called "relation-update formulas". Here we use separate terms so that we can refer easily to relation-maintenance formulas, which are the main subject of Chapter 3. The term "relation-maintenance formula" emphasizes the connection to work in the database community on *view maintenance* (see Sect. 3.5). ("View updating" is something different: an update is made to the value of a view relation and changes are propagated back to the base relations.)

superset of all the execution states that can occur at P. To determine whether a property always holds at P, one checks whether it holds in all of the structures that were collected there.

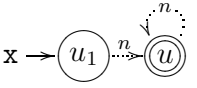
Fig. 2.9 illustrates the abstract execution of the statement $\mathtt{y} = \mathtt{x}$ on a 3-valued logical structure that represents concrete lists of length 2 or more. Instrumentation relations and relation-maintenance formulas have been omitted from the figure. The abstract execution of the statement $\mathtt{y} = \mathtt{x}$ is revisited in Ex. 3.2 of Chapter 3, which discusses relation-maintenance formulas.

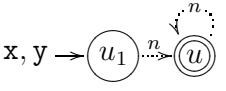
Other Operations on Logical Structures $focus[\varphi]$ is a heuristic that elaborates a 3-valued structure—causing it to be replaced by a collection of more precise structures that, taken together, represent the same set of concrete stores;[3] the criterion for refinement is to ensure that the formula φ evaluates to a definite value for all complete assignments to φ's free variables. The operation thus brings φ "into focus".

By invoking $focus$ before applying each structure transformer, focusing is used to reduce the number of indefinite values that arise when relation-transfer and relation-maintenance formulas are evaluated in 3-valued structures. The $focus$ formulas aim to *sharpen* the values of relations when applied to the individuals that are affected by the transformer. (This often involves the *materialization* of a concrete individual out of a summary individual.) For program-analysis applications, it was proposed in [84] that for a statement of the form *lhs* = *rhs*, the focus formula should identify the memory cells that correspond to the L-value of *lhs* and the R-value of *rhs*. This ensures that the application of an abstract transformer performs a *strong update* of the values of core relations that represent pointer variables and fields that are updated by the statement, i.e., does not set those values to $1/2$.

Not all logical structures represent admissible stores. To exclude structures that do not, we impose integrity constraints. For instance, relation $x(v)$ of Fig. 2.3 captures whether pointer variable $\mathtt{x}$ points to memory cell v; x would be given the attribute "unique", which imposes the integrity constraint that x can hold for at most one individual in any structure: $\forall\, v_1, v_2 \colon x(v_1) \wedge x(v_2) \Rightarrow v_1 = v_2$. This formula evaluates to 1 in any 2-valued logical structure that corresponds to an

[3] This operation can be viewed as a partial concretization.

<table>
<tr><td>Structure before</td><td>
unary rels.
<table>
<tr><th>indiv.</th><th>x</th><th>y</th></tr>
<tr><td>u_1</td><td>1</td><td>0</td></tr>
<tr><td>u</td><td>0</td><td>0</td></tr>
</table>
binary rels.
<table>
<tr><th>n</th><th>u_1</th><th>u</th></tr>
<tr><td>u_1</td><td>0</td><td>1/2</td></tr>
<tr><td>u</td><td>0</td><td>1/2</td></tr>
</table>
<table>
<tr><th>eq</th><th>u_1</th><th>u</th></tr>
<tr><td>u_1</td><td>1</td><td>0</td></tr>
<tr><td>u</td><td>0</td><td>1/2</td></tr>
</table>


</td></tr>
<tr><td>Statement</td><td>y = x</td></tr>
<tr><td>Relation-transfer formulas</td><td>
$\tau_{x,\mathtt{y=x}}(v) \;=\; x(v)$
$\tau_{y,\mathtt{y=x}}(v) \;=\; x(v)$
$\tau_{n,\mathtt{y=x}}(v_1, v_2) \;=\; n(v_1, v_2)$
$\tau_{eq,\mathtt{y=x}}(v_1, v_2) \;=\; eq(v_1, v_2)$
</td></tr>
<tr><td>Structure after</td><td>
unary rels.
<table>
<tr><th>indiv.</th><th>x</th><th>y</th></tr>
<tr><td>u_1</td><td>1</td><td>1</td></tr>
<tr><td>u</td><td>0</td><td>0</td></tr>
</table>
binary rels.
<table>
<tr><th>n</th><th>u_1</th><th>u</th></tr>
<tr><td>u_1</td><td>0</td><td>1/2</td></tr>
<tr><td>u</td><td>0</td><td>1/2</td></tr>
</table>
<table>
<tr><th>eq</th><th>u_1</th><th>u</th></tr>
<tr><td>u_1</td><td>1</td><td>0</td></tr>
<tr><td>u</td><td>0</td><td>1/2</td></tr>
</table>


</td></tr>
</table>

Figure 2.9 The relation-transfer formulas for x, y, and n express a transformation on logical structures that corresponds to the semantics of $\mathtt{y} = \mathtt{x}$

admissible store. Integrity constraints contribute to the concretization function (γ) for our abstraction [100]. Integrity constraints are enforced by *coerce*, a clean-up operation that may "sharpen" a 3-valued logical structure by setting an indefinite value ($1/2$) to a definite value (0 or 1), or discard a structure entirely if an integrity constraint is definitely violated by the structure (e.g., if it cannot represent any admissible store). To help prevent an analysis from losing precision, $coerce$ is applied at certain steps of the algorithm, e.g., after the application of an abstract transformer.

In addition, most of the operations described in this section are not constrained to manipulate 3-valued structures that are images of canonical abstraction; they rely on the Embedding Theorem, which applies to any pair of structures for which one can be embedded into the other. Thus, it is not necessary to perform canonical abstraction after the application of each abstract structure transformer. To ensure that abstract interpretation terminates, it is only necessary that canonical abstraction be applied somewhere in each loop, e.g., at the target of each backedge in the CFG.

Chapter 3

Finite Differencing of Logical Formulas

The present chapter addresses an instance of the following fundamental challenge in applying abstract interpretation:

> Given the concrete semantics for a language and a desired abstraction, how does one create the associated abstract transformers?

In our context, the semantics of statements is expressed using logical formulas that describe changes to core-relation values. When instrumentation relations (defined via logical formulas over core relations) have been introduced to refine an abstraction, the challenge is to reflect the changes in core-relation values in the values of the instrumentation relations. The algorithm presented in this chapter provides a way to create formulas that maintain correct values for the instrumentation relations, and thereby provides a way to generate, completely automatically, the part of the transformers of an abstract semantics that deals with instrumentation relations. The algorithm is based on the principle of *finite differencing*, and transforms an instrumentation relation's defining formula into a relation-maintenance formula that captures what the instrumentation relation's new value should be. The algorithm runs in time linear in the size of the instrumentation relation's defining formula. This research was motivated by work on static analysis based on 3-valued logic [84]; however, any analysis method that relies on logic—2-valued or 3-valued—to express a program's semantics may be able to benefit from these techniques.

The chapter is organized as follows: Sect. 3.1 defines the relation-maintenance problem. Sect. 3.2 presents a method for generating maintenance formulas for instrumentation relations. Sect. 3.3 discusses extensions to handle instrumentation relations that use transitive closure.

Sect. 3.4 presents experimental results. Sect. 3.5 discusses related work. App. A contains the proofs of certain propositions.

3.1 The Problem: Maintaining Instrumentation Relations

The execution of a statement st transforms a logical structure S, which represents a store that arises just before st, into a new structure S', which represents the corresponding store just after st executes. The structure that consists of just the core relations of S' is called a *proto-structure*, denoted by S'_{proto}. The creation of S'_{proto} from S, denoted by $S'_{proto} := [\![st]\!]_3(S)$, can be expressed as

$$\text{for each } c \in \mathcal{C} \text{ and } u_1, \ldots, u_k \in U^S,$$
$$\iota^{S'_{proto}}(c)(u_1, \ldots, u_k) := [\![\tau_{c,st}(v_1, \ldots, v_k)]\!]_3^S([v_1 \mapsto u_1, \ldots, v_k \mapsto u_k]). \tag{3.1}$$

In general, if we compare the various relations of S'_{proto} with those of S, some tuples will have been added and others will have been deleted.

We now come to the crux of the matter: Suppose that ψ_p defines instrumentation relation p; how should the static-analysis engine obtain the value of p in S'?

An instrumentation relation whose defining formula is expressed solely in terms of core relations is said to be in *core normal form*. Because there are no circular dependences, an instrumentation relation's defining formula can always be put in core normal form by repeated substitution until only core relations remain. When ψ_p is in core normal form, or has been converted to core normal form, it is possible to determine the value of each instrumentation relation p by evaluating ψ_p in structure S'_{proto}:

$$\text{for each } u_1, \ldots, u_k \in U^S,$$
$$\iota^{S'}(p)(u_1, \ldots, u_k) := [\![\psi_p(v_1, \ldots, v_k)]\!]_3^{S'_{proto}}([v_1 \mapsto u_1, \ldots, v_k \mapsto u_k]). \tag{3.2}$$

Thus, in principle it is possible to maintain the values of instrumentation relations via Eqn. (3.2). In practice, however, this approach does not work very well. As observed elsewhere [84], when working in 3-valued logic, it is usually possible to retain more precision by defining a special

instrumentation-relation maintenance formula, $\mu_{p,st}(v_1, \ldots, v_k)$, and evaluating $\mu_{p,st}(v_1, \ldots, v_k)$ in structure S:

$$\text{for each } u_1, \ldots, u_k \in U^S,$$
$$\iota^{S'}(p)(u_1, \ldots, u_k) := [\![\mu_{p,st}(v_1, \ldots, v_k)]\!]_3^S([v_1 \mapsto u_1, \ldots, v_k \mapsto u_k]). \tag{3.3}$$

The advantage of the relation-maintenance approach is that the results of program analysis can be more accurate. In 3-valued logic, when $\mu_{p,st}$ is defined appropriately, the relation-maintenance strategy can generate a definite value (0 or 1) when the evaluation of ψ_p on S'_{proto} generates the indefinite value $1/2$.

To ensure that an analysis is conservative, however, one must also show that the following property holds:

Definition 3.1 Suppose that p is an instrumentation relation defined by formula ψ_p. Relation-maintenance formula $\mu_{p,st}$ *maintains* p *correctly for statement* st if, for all $S \in$ 2-STRUCT$[\mathcal{R}]$ and all Z, $[\![\mu_{p,st}]\!]_2^S(Z) = [\![\psi_p]\!]_2^{[\![st]\!]_2(S)}(Z)$.

For an instrumentation relation in core normal form, it is always possible to provide a relation-maintenance formula that satisfies Defn. 3.1 by defining $\mu_{p,st}$ as

$$\mu_{p,st} \stackrel{\text{def}}{=} \psi_p[c \hookleftarrow \tau_{c,st} \mid c \in \mathcal{C}], \tag{3.4}$$

where $\varphi[q \hookleftarrow \varphi']$ denotes the formula obtained from φ by replacing each relation occurrence $q(w_1, \ldots, w_k)$ by $\varphi'\{w_1, \ldots, w_k\}$, and $\varphi'\{w_1, \ldots, w_k\}$ denotes the formula obtained from $\varphi'(v_1, \ldots, v_k)$ by replacing each free occurrence of variable v_i by w_i.

The formula $\mu_{p,st}$ defined in Eqn. (3.4) maintains p correctly for statement st because, by the 2-valued version of Eqn. (3.1), $[\![\tau_{c,st}]\!]_2^S(Z) = [\![c]\!]_2^{S'_{proto}}(Z)$; consequently, when $\mu_{p,st}$ of Eqn. (3.4) is evaluated in structure S, the use of $\tau_{c,st}$ in place of c is equivalent to using the value of c when ψ_p is evaluated in S'_{proto}; i.e., for all Z, $[\![\psi_p[c \hookleftarrow \tau_{c,st} \mid c \in \mathcal{C}]]\!]_2^S(Z) = [\![\psi_p]\!]_2^{S'_{proto}}(Z)$. However—and this is precisely the drawback of using Eqn. (3.4) to obtain the $\mu_{p,st}$—the steps of evaluating $[\![\psi_p[c \hookleftarrow \tau_{c,st} \mid c \in \mathcal{C}]]\!]_2^S(Z)$ *mimic exactly* those of evaluating $[\![\psi_p]\!]_2^{S'_{proto}}(Z)$. Consequently, when

we pass to 3-valued logic, for all Z, $[\![\psi_p[c \hookleftarrow \tau_{c,st} \mid c \in \mathcal{C}]]\!]_3^S(Z)$ yields exactly the same value as $[\![\psi_p]\!]_3^{S'_{proto}}(Z)$ (i.e., as evaluating Eqn. (3.2)). Thus, although $\mu_{p,st}$ that satisfy Defn. 3.1 can be obtained automatically via Eqn. (3.4), this approach does not provide a satisfactory solution to the relation-maintenance problem.

Example 3.2 Eqn. (3.5) shows the defining formula for the instrumentation relation is_n ("is-shared using `n` fields"),

$$is_n(v) \stackrel{\text{def}}{=} \exists\, v_1, v_2 \colon n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2, \tag{3.5}$$

which captures whether a memory cell is pointed to by two or more pointer fields of memory cells, e.g., see Fig. 3.1.

Fig. 3.2 illustrates how execution of the statement `y = x` causes the value of is_n to lose precision when its relation-maintenance formula is created according to Eqn. (3.4). The initial 3-valued structure represents all singly-linked lists of length 2 or more in which all memory cells are unshared. Because execution of `y = x` does not change the value of core relation n, $\tau_{n,\mathtt{y=x}}(v_1, v_2)$ is $n(v_1, v_2)$, and hence the formula $\mu_{is_n,\mathtt{y=x}}(v)$ created according to Eqn. (3.4) is $\exists\, v_1, v_2 \colon n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$. As shown in Fig. 3.2, the structure created using this maintenance formula is not as precise as we would like. In particular, $is_n(u) = 1/2$, which means that u can represent a shared cell. Thus, the final 3-valued structure also represents certain cyclic linked lists, such as

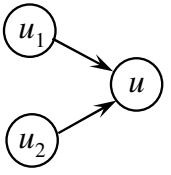
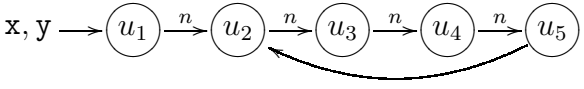
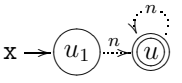


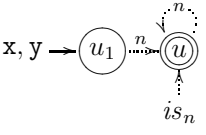
Figure 3.1 A store in which u is shared; i.e., $is_n(u) = 1$



□

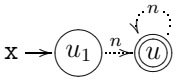
This sort of imprecision can usually be avoided by devising better relation-maintenance formulas. For instance, when $\mu_{is_n,\mathtt{y=x}}(v)$ is defined to be the formula $is_n(v)$—meaning that `y = x` does not change the value of $is_n(v)$—the imprecision illustrated in Fig. 3.2 is avoided (see Fig. 3.3). Hand-crafted relation-maintenance formulas for a variety of instrumentation relations are given in [54, 84, 93]; however, those formulas were created by *ad hoc* methods.

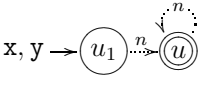
<table>
<tr><td>Structure before</td><td>
unary rels.
<table>
<tr><th>indiv.</th><th>x</th><th>y</th><th>is_n</th></tr>
<tr><td>u_1</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>u</td><td>0</td><td>0</td><td>0</td></tr>
</table>
binary rels.
<table>
<tr><th>n</th><th>u_1</th><th>u</th></tr>
<tr><td>u_1</td><td>0</td><td>1/2</td></tr>
<tr><td>u</td><td>0</td><td>1/2</td></tr>
</table>
<table>
<tr><th>eq</th><th>u_1</th><th>u</th></tr>
<tr><td>u_1</td><td>1</td><td>0</td></tr>
<tr><td>u</td><td>0</td><td>1/2</td></tr>
</table>


</td></tr>
<tr><td>Statement</td><td>y = x</td></tr>
<tr><td>Relation-transfer formulas</td><td>$\tau_{x,\mathtt{y=x}}(v) = x(v)$
$\tau_{y,\mathtt{y=x}}(v) = x(v)$
$\tau_{n,\mathtt{y=x}}(v_1, v_2) = n(v_1, v_2)$
$\tau_{eq,\mathtt{y=x}}(v_1, v_2) = eq(v_1, v_2)$</td></tr>
<tr><td>Relation-maintenance formula</td><td>$\mu_{is_n,\mathtt{y=x}}(v) = \exists\, v_1, v_2 \colon n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$</td></tr>
<tr><td>Structure after</td><td>
unary rels.
<table>
<tr><th>indiv.</th><th>x</th><th>y</th><th>is_n</th></tr>
<tr><td>u_1</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>u</td><td>0</td><td>0</td><td>1/2</td></tr>
</table>
binary rels.
<table>
<tr><th>n</th><th>u_1</th><th>u</th></tr>
<tr><td>u_1</td><td>0</td><td>1/2</td></tr>
<tr><td>u</td><td>0</td><td>1/2</td></tr>
</table>
<table>
<tr><th>eq</th><th>u_1</th><th>u</th></tr>
<tr><td>u_1</td><td>1</td><td>0</td></tr>
<tr><td>u</td><td>0</td><td>1/2</td></tr>
</table>


</td></tr>
</table>

Figure 3.2 An illustration of the loss of precision in the value of is_n when its relation-maintenance formula is defined by $\exists\, v_1, v_2 \colon n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$. The use of this relation-maintenance formula causes a structure to be created in which the individual u may represent a shared memory cell

<table>
<tr><td>Structure before</td><td>
unary rels.
<table>
<tr><th>indiv.</th><th>x</th><th>y</th><th>is_n</th></tr>
<tr><td>u_1</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>u</td><td>0</td><td>0</td><td>0</td></tr>
</table>
binary rels.
<table>
<tr><th>n</th><th>u_1</th><th>u</th></tr>
<tr><td>u_1</td><td>0</td><td>1/2</td></tr>
<tr><td>u</td><td>0</td><td>1/2</td></tr>
</table>
<table>
<tr><th>eq</th><th>u_1</th><th>u</th></tr>
<tr><td>u_1</td><td>1</td><td>0</td></tr>
<tr><td>u</td><td>0</td><td>1/2</td></tr>
</table>


</td></tr>
<tr><td>Statement</td><td>y = x</td></tr>
<tr><td>Relation-transfer formulas</td><td>$\tau_{x,\mathtt{y=x}}(v) = x(v)$
$\tau_{y,\mathtt{y=x}}(v) = x(v)$
$\tau_{n,\mathtt{y=x}}(v_1, v_2) = n(v_1, v_2)$</td></tr>
<tr><td>Relation-maintenance formula</td><td>$\mu_{is_n,\mathtt{y=x}}(v) = is_n(v)$</td></tr>
<tr><td>Structure after</td><td>
unary rels.
<table>
<tr><th>indiv.</th><th>x</th><th>y</th><th>is_n</th></tr>
<tr><td>u_1</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>u</td><td>0</td><td>0</td><td>0</td></tr>
</table>
binary rels.
<table>
<tr><th>n</th><th>u_1</th><th>u</th></tr>
<tr><td>u_1</td><td>0</td><td>1/2</td></tr>
<tr><td>u</td><td>0</td><td>1/2</td></tr>
</table>
<table>
<tr><th>eq</th><th>u_1</th><th>u</th></tr>
<tr><td>u_1</td><td>1</td><td>0</td></tr>
<tr><td>u</td><td>0</td><td>1/2</td></tr>
</table>


</td></tr>
</table>

Figure 3.3 Example showing how the imprecision that was illustrated in Fig. 3.2 is avoided with the relation-maintenance formula $\mu_{is_n,\mathtt{y=x}}(v) = is_n(v)$. (Ex. 3.3 shows how this is generated automatically.)

To sum up, prior to the work presented in this chapter, the user needed to supply a formula $\mu_{p,st}$ for each instrumentation relation p and each statement st. In effect, the user needed to write down two *separate* characterizations of each instrumentation relation p: (i) ψ_p, which defines p directly; and (ii) $\mu_{p,st}$, which specifies how execution of each kind of statement in the language affects p. Moreover, it was the user's responsibility to ensure that the two characterizations are mutually consistent. In contrast, with the new method for automatically creating relation-maintenance formulas presented in Sects. 3.2 and 3.3, the user's responsibility is reduced to defining the ψ_p. (This obligation is addressed in Chapter 5.)

3.2 A Finite-Differencing Scheme for 2-Valued (and 3-Valued) First-Order Logic

This section presents a finite-differencing scheme for creating relation-maintenance formulas. The discussion will be couched in terms of 2-valued logic; however, by the Embedding Theorem (Theorem 2.6, [84, Theorem 4.9]), the relation-maintenance formulas that we derive provide sound results when interpreted in 3-valued logic. In 3-valued logic, as demonstrated in Fig. 3.3 (and discussed further in Ex. 3.3), the resulting formula can lead to a strictly more precise result than merely reevaluating an instrumentation relation's defining formula.

A relation-maintenance formula $\mu_{p,st}$ for $p \in \mathcal{I}$ is defined in terms of two finite-differencing operators, denoted by $\Delta^{-}_{st}[\cdot]$ and $\Delta^{+}_{st}[\cdot]$, which capture the negative and positive changes, respectively, that execution of statement st induces in an instrumentation relation's value. The formula $\mu_{p,st}$ is created by combining p with $\Delta^{-}_{st}[\psi_p]$ and $\Delta^{+}_{st}[\psi_p]$ as follows: $\mu_{p,st} = p ? \neg\Delta^{-}_{st}[\psi_p] : \Delta^{+}_{st}[\psi_p]$. The formula $\mu_{p,st}$ states the conditions under which the new value of p (i.e., its value in S') is 1. These conditions are specified in terms of the old values of p, $\Delta^{-}_{st}[\psi_p]$, and $\Delta^{+}_{st}[\psi_p]$ (i.e., their values in S). The formula $\mu_{p,st}$ states that if p's old value is 1, then its new value is 1 unless there is a negative change; if p's old value is 0, then its new value is 1 if there is a positive change.

Fig. 3.4 depicts how the static-analysis engine evaluates $\Delta^{-}_{st}[\psi_p]$ and $\Delta^{+}_{st}[\psi_p]$ in S and combines these values with the old value p to obtain the desired new value p''. The operators $\Delta^{-}_{st}[\cdot]$ and $\Delta^{+}_{st}[\cdot]$ are defined recursively, as shown in Fig. 3.5. The definitions in Fig. 3.5 make use of the following

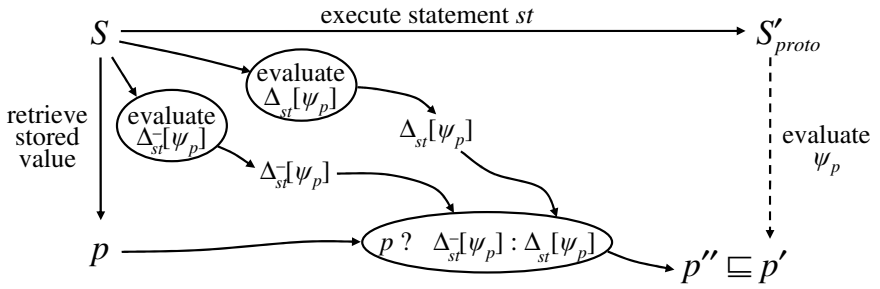


Figure 3.4 How to maintain the value of ψ_p in 3-valued logic in response to changes in the values of core relations caused by the execution of statement st

operator:

$$\mathbf{F}_{st}[\varphi] \stackrel{\text{def}}{=} \varphi \,?\, \neg\Delta^{-}_{st}[\varphi] : \Delta^{+}_{st}[\varphi]. \tag{3.6}$$

Thus, maintenance formula $\mu_{p,st}$ can also be expressed as $\mu_{p,st} = \mathbf{F}_{st}[p]$.

Eqn. (3.6) and Fig. 3.5 define a syntax-directed translation scheme that can be implemented via a recursive walk over a formula φ. The operators $\Delta^{-}_{st}[\cdot]$ and $\Delta^{+}_{st}[\cdot]$ are mutually recursive. For instance, $\Delta^{+}_{st}[\neg\varphi_1] = \Delta^{-}_{st}[\varphi_1]$ and $\Delta^{-}_{st}[\neg\varphi_1] = \Delta^{+}_{st}[\varphi_1]$. Moreover, each occurrence of $\mathbf{F}_{st}[\varphi_i]$ contains additional occurrences of $\Delta^{-}_{st}[\varphi_i]$ and $\Delta^{+}_{st}[\varphi_i]$.

Note how $\Delta^{-}_{st}[\cdot]$ and $\Delta^{+}_{st}[\cdot]$ for $\varphi_1 \vee \varphi_2$ and $\varphi_1 \wedge \varphi_2$ exhibit the "convolution" pattern characteristic of differentiation, finite differencing, and divided differencing.

Continuing the analogy with differentiation, it helps to bear in mind that the "independent variables" are the core relations—which are being changed by the $\tau_{c,st}$ formulas; the dependent variable is the value of φ. A formal justification of Fig. 3.5 is stated later (Theorem 3.5 and Cor. 3.6); here we merely explain informally a few of the cases from Fig. 3.5:

$\Delta^{+}_{st}[\mathbf{1}] = \mathbf{0}$, $\Delta^{-}_{st}[\mathbf{1}] = \mathbf{0}$. The value of atomic formula $\mathbf{1}$ does not depend on any core relations; hence its value is unaffected by changes in them.

$\Delta^{-}_{st}[\varphi_1 \wedge \varphi_2] = (\Delta^{-}_{st}[\varphi_1] \wedge \varphi_2) \vee (\varphi_1 \wedge \Delta^{-}_{st}[\varphi_2])$. Tuples of individuals removed from $\varphi_1 \wedge \varphi_2$ are either tuples of individuals removed from φ_1 for which φ_2 also holds (i.e., $(\Delta^{-}_{st}[\varphi_1] \wedge \varphi_2)$), or they are tuples of individuals removed from φ_2 for which φ_1 also holds, (i.e., $(\varphi_1 \wedge \Delta^{-}_{st}[\varphi_2])$.

φ	$\Delta^+_{st}[\varphi]$	$\Delta^-_{st}[\varphi]$
$\mathbf{1}$	$\mathbf{0}$	$\mathbf{0}$
$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$
$p(w_1, \ldots, w_k)$, $p \in \mathcal{C}$, and $\tau_{p,st}$ is of the form $p \,?\, \neg\delta^-_{p,st} : \delta^+_{p,st}$	$(\delta^+_{p,st} \wedge \neg p)\{w_1, \ldots, w_k\}$	$(\delta^-_{p,st} \wedge p)\{w_1, \ldots, w_k\}$
$p(w_1, \ldots, w_k)$, $p \in \mathcal{C}$, and $\tau_{p,st}$ is of the form $p \vee \delta_{p,st}$ or $\delta_{p,st} \vee p$	$(\delta_{p,st} \wedge \neg p)\{w_1, \ldots, w_k\}$	$\mathbf{0}$
$p(w_1, \ldots, w_k)$, $p \in \mathcal{C}$, and $\tau_{p,st}$ is of the form $p \wedge \delta_{p,st}$ or $\delta_{p,st} \wedge p$	$\mathbf{0}$	$(\neg\delta_{p,st} \wedge p)\{w_1, \ldots, w_k\}$
$p(w_1, \ldots, w_k)$, $p \in \mathcal{C}$, but $\tau_{p,st}$ is not of the above forms	$(\tau_{p,st} \wedge \neg p)\{w_1, \ldots, w_k\}$	$(p \wedge \neg\tau_{p,st})\{w_1, \ldots, w_k\}$
$p(w_1, \ldots, w_k)$, $p \in \mathcal{I}$	$((\exists\, v\colon \Delta^+_{st}[\varphi_1]) \wedge \neg p)\{w_1, \ldots, w_k\}$ if $\psi_p \equiv \exists\, v\colon \varphi_1$ $\Delta^+_{st}[\psi_p]\{w_1, \ldots, w_k\}$ otherwise	$((\exists\, v\colon \Delta^-_{st}[\varphi_1]) \wedge p)\{w_1, \ldots, w_k\}$ if $\psi_p \equiv \forall\, v\colon \varphi_1$ $\Delta^-_{st}[\psi_p]\{w_1, \ldots, w_k\}$ otherwise
$\neg\varphi_1$	$\Delta^-_{st}[\varphi_1]$	$\Delta^+_{st}[\varphi_1]$
$\varphi_1 \vee \varphi_2$	$(\Delta^+_{st}[\varphi_1] \wedge \neg\varphi_2) \vee (\neg\varphi_1 \wedge \Delta^+_{st}[\varphi_2])$	$(\Delta^-_{st}[\varphi_1] \wedge \neg\mathbf{F}_{st}[\varphi_2]) \vee (\neg\mathbf{F}_{st}[\varphi_1] \wedge \Delta^-_{st}[\varphi_2])$
$\varphi_1 \wedge \varphi_2$	$(\Delta^+_{st}[\varphi_1] \wedge \mathbf{F}_{st}[\varphi_2]) \vee (\mathbf{F}_{st}[\varphi_1] \wedge \Delta^+_{st}[\varphi_2])$	$(\Delta^-_{st}[\varphi_1] \wedge \varphi_2) \vee (\varphi_1 \wedge \Delta^-_{st}[\varphi_2])$
$\exists\, v\colon \varphi_1$	$(\exists\, v\colon \Delta^+_{st}[\varphi_1]) \wedge \neg(\exists\, v\colon \varphi_1)$	$(\exists\, v\colon \Delta^-_{st}[\varphi_1]) \wedge \neg(\exists\, v\colon \mathbf{F}_{st}[\varphi_1])$
$\forall\, v\colon \varphi_1$	$(\exists\, v\colon \Delta^+_{st}[\varphi_1]) \wedge (\forall\, v\colon \mathbf{F}_{st}[\varphi_1])$	$(\exists\, v\colon \Delta^-_{st}[\varphi_1]) \wedge (\forall\, v\colon \varphi_1)$

Figure 3.5 Finite-difference formulas for first-order formulas

$$\Delta^{+}_{st}[is_n(v)] \;=\; \left(\exists\, v_1, v_2\colon \left(\begin{array}{rl} & (\Delta^{+}_{st}[n(v_1,v)] \wedge \mathbf{F}_{st}[n(v_2,v)]) \\ \vee & (\mathbf{F}_{st}[n(v_1,v)] \wedge \Delta^{+}_{st}[n(v_2,v)]) \end{array}\right) \wedge v_1 \neq v_2\right) \wedge \neg is_n(v)$$

$$\Delta^{-}_{st}[is_n(v)] \;=\; \left\{\begin{array}{l} \left(\exists\, v_1, v_2\colon \left(\begin{array}{rl} & (\Delta^{-}_{st}[n(v_1,v)] \wedge n(v_2,v)) \\ \vee & (n(v_1,v) \wedge \Delta^{-}_{st}[n(v_2,v)]) \end{array}\right) \wedge v_1 \neq v_2\right) \\ \wedge \\ \neg\left(\exists\, v_1, v_2\colon \left(\begin{array}{l} (n(v_1,v) \wedge n(v_2,v) \wedge v_1 \neq v_2) \\ \quad ?\;\; \neg\left(\left(\begin{array}{rl} & (\Delta^{-}_{st}[n(v_1,v)] \wedge n(v_2,v)) \\ \vee & (n(v_1,v) \wedge \Delta^{-}_{st}[n(v_2,v)]) \end{array}\right) \wedge v_1 \neq v_2\right) \\ \quad :\;\; \left(\begin{array}{rl} & (\Delta^{+}_{st}[n(v_1,v)] \wedge \mathbf{F}_{st}[n(v_2,v)]) \\ \vee & (\mathbf{F}_{st}[n(v_1,v)] \wedge \Delta^{+}_{st}[n(v_2,v)]) \end{array}\right) \wedge v_1 \neq v_2 \end{array}\right)\right) \end{array}\right.$$

Figure 3.6 Finite-difference formulas for the instrumentation relation $is_n(v)$

$\Delta^{+}_{st}[\exists\, v\colon \varphi_1] = (\exists\, v\colon \Delta^{+}_{st}[\varphi_1]) \wedge \neg(\exists\, v\colon \varphi_1)$. For $\exists\, v\colon \varphi_1$ to change value from 0 to 1, there must be at least one individual for which φ_1 changes value from 0 to 1 (i.e., $\exists\, v\colon \Delta^{+}_{st}[\varphi_1]$ holds), and $\exists\, v\colon \varphi_1$ must not already hold (i.e., $\neg(\exists\, v\colon \varphi_1)$ holds).

$\Delta^{+}_{st}[p(w_1, \ldots, w_k)] = (\exists\, v\colon \Delta^{+}_{st}[\varphi_1]) \wedge \neg p$, if $p \in \mathcal{I}$ and $\psi_p \equiv \exists\, v\colon \varphi_1$. This is similar to the previous case, except that the term to ensure that $\exists\, v\colon \varphi_1$ does not already hold (i.e., $\neg(\exists\, v\colon \varphi_1)$) is replaced by the formula $\neg p$. Thus, when $(\exists\, v\colon \Delta^{+}_{st}[\varphi_1]) \wedge \neg p$ is evaluated, the stored value of $\exists\, v\colon \varphi_1$, i.e., p, will be used instead of the value obtained by reevaluating $\exists\, v\colon \varphi_1$.

$\Delta^{+}_{st}[p(w_1, \ldots, w_k)] = \Delta^{+}_{st}[\psi_p\{w_1, \ldots, w_k\}]$, if $p \in \mathcal{I}$ and $\psi_p \not\equiv \exists\, v\colon \varphi_1$. To characterize the positive changes to p, apply Δ^{+}_{st} to p's defining formula ψ_p.

One special case is also worth noting: $\Delta^{+}_{st}[v_1 = v_2] = \mathbf{0}$ and $\Delta^{-}_{st}[v_1 = v_2] = \mathbf{0}$ because the value of the atomic formula $(v_1 = v_2)$ (shorthand for $eq(v_1, v_2)$) does not depend on any core relations; hence, its value is unaffected by changes in them.

Example 3.3 Consider the instrumentation relation is_n ("is-shared using `n` fields"), defined in Eqn. (3.5). Fig. 3.6 shows the formulas obtained for $\Delta^{+}_{st}[is_n(v)]$ and $\Delta^{-}_{st}[is_n(v)]$.

For a particular statement, the formulas in Fig. 3.6 can usually be simplified. For instance, for $\mathtt{y} = \mathtt{x}$, the relation-transfer formula $\tau_{n,\mathtt{y}=\mathtt{x}}(v_1, v_2)$ is $n(v_1, v_2)$; see Fig. 3.2. Thus, by Fig. 3.5, the formulas for $\Delta^{-}_{\mathtt{y}=\mathtt{x}}[n(v_1, v)]$ and $\Delta^{+}_{\mathtt{y}=\mathtt{x}}[n(v_1, v)]$ are both $n(v_1, v) \wedge \neg n(v_1, v)$, which simplifies to $\mathbf{0}$. (In our implementation, simplifications are performed greedily at formula-construction time; e.g., the constructor for $\wedge$ rewrites $\mathbf{0} \wedge p$ to $\mathbf{0}$, $\mathbf{1} \wedge p$ to p, $p \wedge \neg p$ to $\mathbf{0}$, etc.) The formulas in Fig. 3.6 simplify to $\Delta^{+}_{\mathtt{y}=\mathtt{x}}[is_n(v)] = \mathbf{0}$ and $\Delta^{-}_{\mathtt{y}=\mathtt{x}}[is_n(v)] = \mathbf{0}$. Consequently, $\mu_{is_n,\mathtt{y}=\mathtt{x}}(v) = \mathbf{F}_{\mathtt{y}=\mathtt{x}}[is_n(v)] = is_n(v)\,?\,\neg\mathbf{0} : \mathbf{0} = is_n(v)$. As shown in Fig. 3.3, this definition of $\mu_{is_n,\mathtt{y}=\mathtt{x}}(v)$ avoids the imprecision that was illustrated in Ex. 3.2. □

Correctness of the Finite-Differencing Scheme

The correctness of the finite-differencing scheme given above is established with the help of the following lemma:

Lemma 3.4 For every formula φ, φ_1, φ_2 and statement st, the following properties hold:[1]

(i) $\Delta^{+}_{st}[\varphi] \overset{\text{meta}}{\Longleftrightarrow} \mathbf{F}_{st}[\varphi] \wedge \neg\varphi$

(ii) $\Delta^{-}_{st}[\varphi] \overset{\text{meta}}{\Longleftrightarrow} \varphi \wedge \neg\mathbf{F}_{st}[\varphi]$

(iii) (a) $\mathbf{F}_{st}[\neg\varphi_1] \overset{\text{meta}}{\Longleftrightarrow} \neg\mathbf{F}_{st}[\varphi_1]$

(b) $\mathbf{F}_{st}[\varphi_1 \vee \varphi_2] \overset{\text{meta}}{\Longleftrightarrow} \mathbf{F}_{st}[\varphi_1] \vee \mathbf{F}_{st}[\varphi_2]$

(c) $\mathbf{F}_{st}[\varphi_1 \wedge \varphi_2] \overset{\text{meta}}{\Longleftrightarrow} \mathbf{F}_{st}[\varphi_1] \wedge \mathbf{F}_{st}[\varphi_2]$

(d) $\mathbf{F}_{st}[\exists\, v\colon \varphi_1] \overset{\text{meta}}{\Longleftrightarrow} \exists\, v\colon \mathbf{F}_{st}[\varphi_1]$

(e) $\mathbf{F}_{st}[\forall\, v\colon \varphi_1] \overset{\text{meta}}{\Longleftrightarrow} \forall\, v\colon \mathbf{F}_{st}[\varphi_1]$

Proof See App. A.

□

[1] To simplify the presentation, we use $lhs \overset{\text{meta}}{\Longleftrightarrow} rhs$ and $lhs \overset{\text{meta}}{\Longrightarrow} rhs$ as shorthands for $[\![lhs]\!]^S_2(Z) = [\![rhs]\!]^S_2(Z)$ and $[\![lhs]\!]^S_2(Z) \leq [\![rhs]\!]^S_2(Z)$, respectively, for any $S \in$ 2-STRUCT and assignment Z that is complete for lhs and rhs.

Lemma 3.4 shows that for 2-STRUCTs, $\Delta^{+}_{st}[\varphi]$ specifies the tuples that are not in the relation defined by φ, but need to be added in response to the execution of st, and that $\Delta^{-}_{st}[\varphi]$ specifies the tuples that are in the relation defined by φ that need to be removed. This lemma is used in the proof of the following theorem, which ensures the correctness of the finite-differencing transformation given in Fig. 3.5:

Theorem 3.5 Let S be a structure in 2-STRUCT, and let S'_{proto} be the proto-structure for statement st obtained from S. Let S' be the structure obtained by using S'_{proto} as the first approximation to S' and then filling in instrumentation relations in a topological ordering of the dependences among them: for each arity-k relation $p \in \mathcal{I}$, $\iota^{S'}(p)$ is obtained by evaluating $[\![\psi_p(v_1, \ldots, v_k)]\!]_2^{S'}([v_1 \mapsto u'_1, \ldots, v_k \mapsto u'_k])$ for all tuples $(u'_1, \ldots, u'_k) \in (U^{S'})^k$. Then for every formula $\varphi(v_1, \ldots, v_k)$ and complete assignment Z for $\varphi(v_1, \ldots, v_k)$, $[\![\mathbf{F}_{st}[\varphi(v_1, \ldots, v_k)]]\!]_2^{S}(Z) = [\![\varphi(v_1, \ldots, v_k)]\!]_2^{S'}(Z)$.

Proof See App. A.

□

For 3-STRUCTs, the soundness of the finite-differencing transformation given in Fig. 3.5 follows from Theorem 3.5 by the Embedding Theorem (Theorem 2.6):

Corollary 3.6 Let $S, S' \in$ 2-STRUCT be defined as in Theorem 3.5. Let $S^{\sharp} \in$ 3-STRUCT be such that $f : U^S \to U^{S^{\sharp}}$ embeds S in $S^{\sharp}$, i.e., $S \sqsubseteq^{f} S^{\sharp}$. Then for every formula $\varphi(v_1, \ldots, v_k)$ and complete assignment Z for $\varphi(v_1, \ldots, v_k)$, $[\![\mathbf{F}_{st}[\varphi(v_1, \ldots, v_k)]]\!]_3^{S^{\sharp}}(f \circ Z) \sqsupseteq [\![\varphi(v_1, \ldots, v_k)]\!]_2^{S'}(Z)$.

Optimized Formulas for $\mathbf{F}_{st}[\varphi]$

For a non-atomic formula φ, the operator $\mathbf{F}_{st}[\varphi]$ as defined in Formula (3.6) evaluates φ because it has no stored value for φ. As a result, the version of the operator $\mathbf{F}_{st}[\cdot]$ as defined in Formula (3.6) does not result in higher precision. One way to overcome this problem is to propagate $\mathbf{F}_{st}[\cdot]$ into the subformulas of φ, as shown in Fig. 3.7. The correctness of the operator $\mathbf{F}_{st}[\cdot]$ as defined in Fig. 3.7 is guaranteed by Lemma 3.4. Generally, such propagation has the effect of generating smaller finite-difference formulas. This may result in faster evaluation of the finite-difference formulas. We rely on this optimization in our implementation.

φ	$\mathbf{F}_{st}[\varphi]$
$\neg\varphi_1$	$\neg\mathbf{F}_{st}[\varphi_1]$
$\varphi_1 \vee \varphi_2$	$\mathbf{F}_{st}[\varphi_1] \vee \mathbf{F}_{st}[\varphi_2]$
$\varphi_1 \wedge \varphi_2$	$\mathbf{F}_{st}[\varphi_1] \wedge \mathbf{F}_{st}[\varphi_2]$
$\exists\, v\colon \varphi_1$	$\exists\, v\colon \mathbf{F}_{st}[\varphi_1]$
$\forall\, v\colon \varphi_1$	$\forall\, v\colon \mathbf{F}_{st}[\varphi_1]$

Figure 3.7 Optimized formulas for the operator $\mathbf{F}_{st}[\varphi]$ for non-atomic formula φ

Discussion

Because earlier in the thesis we touted the advantages of being able to apply related 2-valued and 3-valued interpretation functions to a single formula, it may seem somewhat inconsistent for us to make use of a transformation-based approach to maintaining instrumentation relations, in lieu of an approach based on overloading. The reason that we use a transformation-based approach is that it gives us an opportunity to simplify the resulting formulas (either on the fly, or in a post-processing phase after finite differencing).

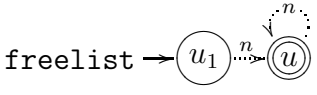
In the context of evaluation in 3-valued logic, simplification is important because even formulas that are tautologies in 2-valued logic may evaluate to $1/2$ in 3-valued logic. For instance, $p \vee \neg p$ yields $1/2$ when p has the value $1/2$, even when p is a nullary relation symbol. The finite-differencing transformation that we implemented uses a formula-minimization procedure for 3-valued logic that we developed. The minimization procedure described in Chapter 7 applies to propositional logic; for propositional logic, it is guaranteed to return an answer that captures the formula's "supervaluational meaning" [94]. This procedure is used as a subroutine in a heuristic method for minimizing first-order formulas; the method works on a formula bottom-up, applying the propositional minimizer to the body of each non-propositional operator (i.e., each quantifier or transitive-closure operator).

A relation-maintenance formula that has been simplified in this way can sometimes yield a definite value in situations where the evaluation of the unsimplified relation-maintenance formula—or, equivalently, an overloaded evaluation of the relation's defining formula—yields $1/2$. (For instance, minimizing $p \vee \neg p$ yields 1, which evaluates to 1 even when p has the value $1/2$.) Consequently, the formula-transformation approach to the relation-maintenance problem leads to more precise static-analysis algorithms.

Malloc and Free

In [84], the modeling of storage-allocation/deallocation operations is carried out with a two-stage statement transformer, the first stage of which changes the number of individuals in the structure. This creates some problems for the finite-differencing approach in establishing appropriate, mutually consistent values for relation tuples that involve the newly allocated individual. Such relation values are needed for the second stage, in which relation-transfer formulas for core relations and relation-maintenance formulas for instrumentation relations are applied in the usual fashion, using Eqns. (3.1) and (3.3).

However, there is a simple way to sidestep this problem, which is to model the free-storage list explicitly, making the following substructure part of every 3-valued structure:



(3.7)

A `malloc` is modeled by advancing the pointer `freelist` into the list, and returning the memory cell that it formerly pointed to. A `free` is modeled by inserting, at the head of `freelist`'s list, the cell being deallocated.

It is true that the use of structure (3.7) to model storage-allocation/deallocation operations also causes the number of individuals in a 3-valued structure to change; however, because the new individual is materialized using the usual mechanisms from [84] (namely, the *focus* and *coerce* operations), values for relation tuples that involve the newly materialized individual will always have safe, mutually consistent values.

p	**IntendedMeaning**	ψ_p
$t_n(v_1, v_2)$	Is v_2 reachable from v_1 along `n` fields?	$n^*(v_1, v_2)$
$r_{n,z}(v)$	Is v reachable from pointer variable `z` along `n` fields?	$\exists\, v_1 \colon z(v_1) \wedge t_n(v_1, v)$
$c_n(v)$	Is v on a directed cycle of `n` fields?	$\exists\, v_1 \colon n(v_1, v) \wedge t_n(v, v_1)$

Figure 3.8 Defining formulas of some instrumentation relations that depend on **RTC**. (Recall that $n^*(v_1, v_2)$ is a shorthand for $(\mathbf{RTC}\ v_1', v_2' \colon n(v_1', v_2'))(v_1, v_2)$.)

3.3 Extension of Sect. 3.2 for Reachability and Transitive Closure

Several instrumentation relations that depend on **RTC** are shown in Fig. 3.8.

Unfortunately, finding a good way to maintain instrumentation relations defined using **RTC** is challenging because it is not known, in general, whether it is possible to write a first-order formula (i.e., without using a transitive-closure operator) that specifies how to maintain the closure of a directed graph in response to edge insertions and deletions. Thus, our strategy has been to investigate special cases for classes of instrumentation relations for which first-order maintenance formulas do exist. Whenever these do not apply, the system falls back on safe maintenance formulas (which themselves use **RTC**).

In the next three sections, we confine ourselves to important special cases for the maintenance of instrumentation relations specified via the **RTC** of a binary formula $\varphi_1(v_1, v_2)$. In Sect. 3.3.1, we consider the case that $\varphi_1(v_1, v_2)$ defines an acyclic graph. In Sect. 3.3.2, we consider the case that $\varphi_1(v_1, v_2)$ defines a tree-shaped graph. Finally, in Sect. 3.3.3, we consider the case that $\varphi_1(v_1, v_2)$ defines a *deterministic graph*, a possibly-cyclic graph, in which every node has outdegree at most one (this class of graphs corresponds to possibly-cyclic linked lists). This collection of techniques allows us to handle most common data structures, such as lists (singly- and doubly-linked; cyclic and acyclic) and trees. The precision of all of these techniques is due to the fact that maintenance of RTC to reflect unit-size changes (single-edge additions or deletions)[2] is performed by first-order logical formulas only. However, maintaining RTC of an arbitrary graph, as well as maintaining

[2]These techniques can be extended to handle bounded-size addition and deletion sets.

RTC of restricted classes of graphs with arbitrary-size changes, is not known to be first-order expressible. In such cases, our algorithm returns a formula that uses the **RTC** operator; the evaluation of such a formula may yield more indefinite answers than necessary.

3.3.1 Transitive-Closure Maintenance in Acyclic Graphs

Consider a binary instrumentation relation p, defined by $\psi_p(v_1, v_2) \equiv (\mathbf{RTC}\, v_1', v_2' \colon \varphi_1)(v_1, v_2)$. If the graph defined by φ_1 is acyclic, it is possible to give a first-order formula that maintains p after the addition or deletion of a single φ_1-edge. The method we use is a minor modification of a method for maintaining non-reflexive transitive closure in an acyclic graph, due to Dong and Su [25].

In the case of an insertion of a single φ_1-edge, the maintenance formula is

$$\mathbf{F}_{st}[p](v_1, v_2) = p(v_1, v_2) \vee (\exists\, v_1', v_2' \colon p(v_1, v_1') \wedge \Delta_{st}^{+}[\varphi_1](v_1', v_2') \wedge p(v_2', v_2)). \tag{3.8}$$

The new value of p contains the old tuples of p, as well as those that represent two old paths (i.e., $p(v_1, v_1')$ and $p(v_2', v_2)$) connected with the new φ_1-edge (i.e., $\Delta_{st}^{+}[\varphi_1](v_1', v_2')$).

The maintenance formula to handle the deletion of a single φ_1-edge is a bit more complicated. We first identify the tuples of p that represent paths that might rely on the edge to be deleted, and thus may need to be removed from p (S stands for *suspicious*):

$$S[p, \varphi_1](v_1, v_2) = \exists\, v_1', v_2' \colon p(v_1, v_1') \wedge \Delta_{st}^{-}[\varphi_1](v_1', v_2') \wedge p(v_2', v_2).$$

We next collect a set of p-tuples that definitely remain in p (T stands for *trusted*):

$$T[p, \varphi_1](v_1, v_2) = (p(v_1, v_2) \wedge \neg S[p, \varphi_1](v_1, v_2)) \vee \mathbf{F}_{st}[\varphi_1](v_1, v_2). \tag{3.9}$$

Finally, the maintenance formula for p for a single φ_1-edge deletion is

$$\mathbf{F}_{st}[p](v_1, v_2) = \exists\, v_1', v_2' \colon T[p, \varphi_1](v_1, v_1') \wedge T[p, \varphi_1](v_1', v_2') \wedge T[p, \varphi_1](v_2', v_2). \tag{3.10}$$

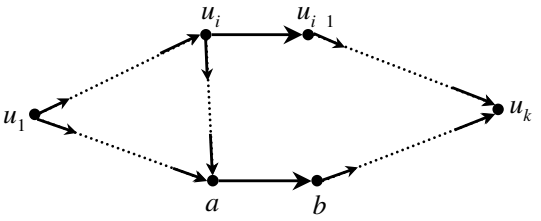


Figure 3.9 Edge (a, b) is being deleted; u_i is the last node along path $u_1, \ldots, u_i, u_{i+1}, \ldots, u_k$ from which a is reachable

Maintenance formulas (3.8) and (3.10) maintain p when two conditions hold: the graph defined by φ_1 is acyclic, and the change to the graph is a single edge addition or deletion (but not both). To see that under these assumptions the maintenance formula for a φ_1-edge deletion is correct, suppose that there is a suspicious tuple $p(u_1, u_k)$, i.e., $S[p, \varphi_1](u_1, u_k) = 1$, but there is a φ_1-path $u_1, \ldots, u_k$ that does not use the deleted φ_1-edge. We need to show that $\mathbf{F}_{st}[p](u_1, u_k)$ has the value 1. Suppose that (a, b) is the φ_1-edge being deleted; because the graph defined by φ_1 is acyclic, there is a $u_i \neq u_k$ that is the last node along path $u_1, \ldots, u_i, u_{i+1}, \ldots, u_k$ from which a is reachable (see Fig. 3.9). Because $p(u_1, u_i)$ and $p(u_{i+1}, u_k)$ both hold, and because u_i cannot be reachable from b (by acyclicity), neither tuple is suspicious; consequently, $T[p, \varphi_1](u_1, u_i) = 1$ and $T[p, \varphi_1](u_{i+1}, u_k) = 1$. Because (u_i, u_{i+1}) is an edge in the new (as well as the old) graph defined by φ_1, we have $\mathbf{F}_{st}[\varphi_1](u_i, u_{i+1}) = 1$, which means that $T[p, \varphi_1](u_i, u_{i+1}) = 1$ as well, yielding $\mathbf{F}_{st}[p](u_1, u_k) = 1$ by Eqn. (3.10).

Fig. 3.10 extends the method for generating relation-maintenance formulas to handle instrumentation relations specified via the **RTC** of a binary formula that defines an acyclic graph. Fig. 3.10 makes use of the operator $T[p, \varphi_1](v, v')$ (Eqn. (3.9)), but recasts Eqns. (3.8) and (3.10) as finite-difference expressions $\Delta^{+}_{st}[\psi_p]$ and $\Delta^{-}_{st}[\psi_p]$, respectively.

Figs. 3.11 and 3.12 show the formulas obtained via the finite-differencing scheme given in Figs. 3.5 and 3.10 for positive and negative changes, respectively, for instrumentation relations defined in Fig. 3.8.

3.3.1.1 Testing the Unit-Size-Change Assumption

To know whether this special-case maintenance strategy can be applied, for each statement st we need to know at analysis-generation time whether the change performed at st, to the graph defined by φ_1, always results in a single edge addition or deletion. If in any admissible 2-STRUCT$[\mathcal{R}]$

φ	$\Delta^{+}_{st}[\varphi]$	
$p(w_1, \ldots, w_k)$, $p \in \mathcal{I}$	$((\exists\, v\colon \Delta^{+}_{st}[\varphi_1]) \wedge \neg p)\{w_1, \ldots, w_k\}$	if $\psi_p \equiv \exists\, v\colon \varphi_1$
	$\wedge \left(\left(\begin{array}{c} (\exists\, v_1', v_2'\colon \Delta^{+}_{st}[\varphi_1](v_1', v_2')) \\ p(v_1, v_1') \\ \wedge\ \Delta^{+}_{st}[\varphi_1](v_1', v_2') \\ \wedge\ p(v_2', v_2) \end{array} \right) \wedge \neg p(v_1, v_2) \right) \{w_1, w_2\}$	if $\psi_p \equiv$ (**RTC** $v_1', v_2'\colon \varphi_1)(v_1, v_2)$
	$\Delta^{+}_{st}[\psi_p]\{w_1, \ldots, w_k\}$	otherwise

φ	$\Delta^{-}_{st}[\varphi]$	
$p(w_1, \ldots, w_k)$, $p \in \mathcal{I}$	$((\exists\, v\colon \Delta^{-}_{st}[\varphi_1]) \wedge p)\{w_1, \ldots, w_k\}$	if $\psi_p \equiv \forall\, v\colon \varphi_1$
	$\wedge \left(\neg \left(\begin{array}{c} (\exists\, v_1', v_2'\colon \Delta^{-}_{st}[\varphi_1](v_1', v_2')) \\ T[p, \varphi_1](v_1, v_1') \\ \wedge\ T[p, \varphi_1](v_1', v_2') \\ \wedge\ T[p, \varphi_1](v_2', v_2) \end{array} \right) \wedge p(v_1, v_2) \right) \{w_1, w_2\}$	if $\psi_p \equiv$ (**RTC** $v_1', v_2'\colon \varphi_1)(v_1, v_2)$
	$\Delta^{-}_{st}[\psi_p]\{w_1, \ldots, w_k\}$	otherwise

Figure 3.10 Extension of the finite-differencing method from Fig. 3.5 to cover **RTC** formulas, for unit-sized changes to an acyclic graph defined by φ_1

relation p		$\Delta^{+}_{st}[\psi_p]$
$t_n(v_3, v_4)$		$\Delta^{+}_{st}[t_n(v_3, v_4)]$
	$=$	$(t_n(v_3, v_4) \vee (\exists\, v_1, v_2\colon t_n(v_3, v_1) \wedge \Delta^{+}_{st}[n(v_1, v_2)] \wedge t_n(v_2, v_4))) \wedge \neg t_n(v_3, v_4)$
$r_{n,z}(v)$		$\Delta^{+}_{st}[r_{n,z}(v)]$
	$=$	$(\exists\, v_1\colon \Delta^{+}_{st}[z(v_1) \wedge t_n(v_1, v)]) \wedge \neg r_{n,z}(v)$
	$=$	$(\exists\, v_1\colon (\Delta^{+}_{st}[z(v_1)] \wedge \mathbf{F}_{st}[t_n(v_1, v)]) \vee (\mathbf{F}_{st}[z(v_1)] \wedge \Delta^{+}_{st}[t_n(v_1, v)])) \wedge \neg r_{n,z}(v)$
$c_n(v)$		$\Delta^{+}_{st}[c_n(v)]$
	$=$	$(\exists\, v_1\colon \Delta^{+}_{st}[n(v_1, v) \wedge t_n(v, v_1)]) \wedge \neg c_n(v)$
	$=$	$(\exists\, v_1\colon (\Delta^{+}_{st}[n(v_1, v)] \wedge \mathbf{F}_{st}[t_n(v, v_1)]) \vee (\mathbf{F}_{st}[n(v_1, v)] \wedge \Delta^{+}_{st}[t_n(v, v_1)])) \wedge \neg c_n(v)$

Figure 3.11 The formulas obtained via the finite-differencing scheme given in Figs. 3.5 and 3.10 for the positive changes in the values of the instrumentation relations defined in Fig. 3.8

relation p	$\Delta^-_{st}[p]$
$t_n(v_3, v_4)$	$\Delta^-_{st}[t_n(v_3, v_4)]$
	$= (\exists\, v_1, v_2 \colon T[t_n, n](v_3, v_1) \wedge T[t_n, n](v_1, v_2) \wedge T[t_n, n](v_2, v_4)) \wedge t_n(v_3, v_4)$
	$= \left(\exists\, v_1, v_2 \colon \begin{array}{l} (t_n(v_3, v_1) \wedge \neg S[t_n, n](v_3, v_1) \vee \mathbf{F}_{st}[n](v_3, v_1)) \wedge \\ (t_n(v_1, v_2) \wedge \neg S[t_n, n](v_1, v_2) \vee \mathbf{F}_{st}[n](v_1, v_2)) \wedge \\ (t_n(v_2, v_4) \wedge \neg S[t_n, n](v_2, v_4) \vee \mathbf{F}_{st}[n](v_2, v_4)) \end{array} \right) \wedge t_n(v_3, v_4)$
	$= \left(\exists\, v_1, v_2 \colon \begin{array}{l} (t_n(v_3, v_1) \wedge \neg(\exists\, v_1', v_2' \colon t_n(v_3, v_1') \wedge \Delta^-_{st}[n](v_1', v_2') \wedge t_n(v_2', v_1)) \vee \mathbf{F}_{st}[n](v_3, v_1)) \wedge \\ (t_n(v_1, v_2) \wedge \neg(\exists\, v_1', v_2' \colon t_n(v_1, v_1') \wedge \Delta^-_{st}[n](v_1', v_2') \wedge t_n(v_2', v_2)) \vee \mathbf{F}_{st}[n](v_1, v_2)) \wedge \\ (t_n(v_2, v_4) \wedge \neg(\exists\, v_1', v_2' \colon t_n(v_2, v_1') \wedge \Delta^-_{st}[n](v_1', v_2') \wedge t_n(v_2', v_4)) \vee \mathbf{F}_{st}[n](v_2, v_4)) \end{array} \right)$
	$\wedge\, t_n(v_3, v_4)$
$r_{n,z}(v)$	$\Delta^-_{st}[r_{n,z}(v)]$
	$= \Delta^-_{st}[\exists\, v_1 \colon x(v_1) \wedge t_n(v_1, v)]$
	$= (\exists\, v_1 \colon \Delta^-_{st}[z(v_1) \wedge t_n(v_1, v)]) \wedge \neg(\exists\, v_1 \mathbf{F}_{st}[z(v_1) \wedge t_n(v_1, v)])$
	$= \left\{ \begin{array}{l} (\exists\, v_1 \colon ((\Delta^-_{st}[z(v_1)] \wedge t_n(v_1, v)) \vee (z(v_1) \wedge \Delta^-_{st}[t_n(v_1, v)]))) \\ \wedge \\ \neg \left(\exists\, v_1 \colon \left(\begin{array}{lll} (z(v_1) \wedge t_n(v_1, v)) & & \\ \quad ? & \neg \Delta^-_{st}[z(v_1) \wedge t_n(v_1, v)] & \\ \quad : & \Delta^+_{st}[z(v_1) \wedge t_n(v_1, v)] & \end{array} \right) \right) \end{array} \right.$
	$= \left\{ \begin{array}{l} (\exists\, v_1 \colon ((\Delta^-_{st}[z(v_1)] \wedge t_n(v_1, v)) \vee (z(v_1) \wedge \Delta^-_{st}[t_n(v_1, v)]))) \\ \wedge \\ \neg \left(\exists\, v_1 \colon \left(\begin{array}{lll} (z(v_1) \wedge t_n(v_1, v)) & & \\ \quad ? & \neg((\Delta^-_{st}[z(v_1)] \wedge t_n(v_1, v)) \vee (z(v_1) \wedge \Delta^-_{st}[t_n(v_1, v)])) & \\ \quad : & ((\Delta^+_{st}[z(v_1)] \wedge \mathbf{F}_{st}[t_n(v_1, v)]) \vee (\mathbf{F}_{st}[z(v_1)] \wedge \Delta^+_{st}[t_n(v_1, v)])) & \end{array} \right) \right) \end{array} \right.$
$c_n(v)$	$\Delta^-_{st}[c_n(v)]$
	$= \Delta^-_{st}[\exists\, v_1 \colon n(v_1, v) \wedge t_n(v, v_1)]$
	$= (\exists\, v_1 \colon \Delta^-_{st}[n(v_1, v) \wedge t_n(v, v_1)]) \wedge \neg \mathbf{F}_{st}[\exists\, v_1 \colon n(v_1, v) \wedge t_n(v, v_1)]$
	$= (\exists\, v_1 \colon (\Delta^-_{st}[n(v_1, v)] \wedge t_n(v, v_1)) \vee (n(v_1, v) \wedge \Delta^-_{st}[t_n(v, v_1)])) \wedge \neg \mathbf{F}_{st}[\exists\, v_1 \colon n(v_1, v) \wedge t_n(v, v_1)]$

Figure 3.12 The formulas obtained via the finite-differencing scheme given in Figs. 3.5 and 3.10 for the negative changes in the values of the instrumentation relations defined in Fig. 3.8

there is a unique satisfying assignment to the two free variables of $\Delta^{+}_{st}[\varphi_1]$ and no assignment satisfies $\Delta^{-}_{st}[\varphi_1]$, then the pair $\Delta^{+}_{st}[\varphi_1]$, $\Delta^{-}_{st}[\varphi_1]$ defines a change that adds exactly one edge to the graph. Similarly, if in any admissible 2-STRUCT$[\mathcal{R}]$ there is a unique satisfying assignment to the two free variables of $\Delta^{-}_{st}[\varphi_1]$ and no assignment satisfies $\Delta^{+}_{st}[\varphi_1]$, then the change is a deletion of exactly one edge from the graph.

Because answering (unique-)satisfiability questions in this logic is in general undecidable, we employ a conservative approximation based on a syntactic analysis of logical formulas. The analysis uses a heuristic to determine a set of variables V such that for each admissible structure, the variables in V have a single possible binding in the formula's satisfying assignments. We refer to such variables as *anchored* variables. For instance, if relation q has the attribute "unique", for each admissible structure there is a single possible binding for variable v in any assignment that satisfies $q(v)$; in a formula that contains an occurrence of $q(v)$, v is an anchored variable. (A conservative algorithm for identifying anchored variables appears later in this section.)

If both free variables of $\Delta^{+}_{st}[\varphi_1]$ are anchored and $\Delta^{-}_{st}[\varphi_1] = 0$, then the change adds one edge to the graph defined by φ_1. Similarly, if both free variables of $\Delta^{-}_{st}[\varphi_1]$ are anchored and $\Delta^{+}_{st}[\varphi_1] = 0$, then the change removes one edge from the graph. In these cases, the reflexive transitive closure of φ_1 can be updated using the method discussed above.

A Test for Anchored Variables Function *Anchored*, shown in Fig. 3.13, conservatively identifies anchored variables in a formula φ. It is invoked as *Anchored*$(\varphi, \emptyset)$. (In our application, at top-level φ is always either $\Delta^{+}_{st}[\varphi_1]$ or $\Delta^{-}_{st}[\varphi_1]$.) *Anchored* uses a handful of patterns to identify anchored variables. For example, if variable v_1 is anchored and binary relation p has the attribute "function",[3] then v_2 is anchored as well. In essence, negations are handled by pushing the negation deeper into the formula. In a disjunction, an anchored variable must be anchored in both subformulas. The conjunction rule accumulates anchored variables in A by a process of successive approximation, during which variables anchored in the left subformula are used to identify new anchored variables in the right subformula and vice versa; this process is iterated until a fixed point

[3]For instance, in program-analysis applications a relation $n(v_1, v_2)$ that records whether field `n` of v_1 points to v_2 is a function relation.

φ	*Anchored*(φ, A_0)
$\mathbf{0}, \mathbf{1}$	A_0
$v_1 = v_2$	$v_1 \in A_0 \to A_0 \cup \{v_2\} \llbracket v_2 \in A_0 \to A_0 \cup \{v_1\} \llbracket A_0$
$p()$	A_0
$p(v)$	*unique*$(p) \to A_0 \cup \{v\} \llbracket A_0$
$p(v_1, v_2)$	*function*$(p) \wedge v_1 \in A_0 \to A_0 \cup \{v_2\}$ $\llbracket$ *invfunction*$(p) \wedge v_2 \in A_0 \to A_0 \cup \{v_1\}$ $\llbracket$ A_0
$\neg\varphi_1$	$\varphi_1 \equiv \neg\varphi_2 \to$ *Anchored*(φ_2, A_0) $\llbracket$ $\varphi_1 \equiv \varphi_2 \vee \varphi_3 \to$ *Anchored*$(\neg\varphi_2 \wedge \neg\varphi_3), A_0)$ $\llbracket$ $\varphi_1 \equiv \varphi_2 \wedge \varphi_3 \to$ *Anchored*$(\neg\varphi_2 \vee \neg\varphi_3), A_0)$ $\llbracket$ $\varphi_1 \equiv \forall\, v\colon \varphi_2 \to$ *Anchored*$(\exists\, v\colon \neg\varphi_2), A_0)$ $\llbracket$ $\varphi_1 \equiv \exists\, v\colon \varphi_2 \to$ *Anchored*$(\forall\, v\colon \neg\varphi_2), A_0)$ $\llbracket$ A_0
$\varphi_1 \vee \varphi_2$	*Anchored*$(\varphi_1, A_0) \cap$ *Anchored*(φ_2, A_0)
$\varphi_1 \wedge \varphi_2$	$\mu A.$(*Anchored*$(\varphi_1, A \cup A_0) \cup$ *Anchored*$(\varphi_2, A \cup A_0))$
$\exists\, v\colon \varphi_1, \forall\, v\colon \varphi_1$	(*Anchored*$(\varphi_1, A_0 - \{v\}) - \{v\}) \cup A_0$
(**RTC** $v_1', v_2'\colon \varphi_1)(v_1, v_2)$	(*Anchored*$(\varphi_1, A_0 - \{v_1', v_2'\}) - \{v_1', v_2'\}) \cup A_0$

Figure 3.13 Function *Anchored* conservatively identifies anchored variables in φ. A_0 contains variables known to be anchored due to the surrounding context.

is reached. The rules for $\exists\, v\colon \varphi_1$ and $\forall\, v\colon \varphi_1$ contain recursive calls on *Anchored* with v removed from the second argument (because bound variable v refers to a different occurrence of v from an identically named v in A_0). If v is anchored in φ_1, it needs to be removed before this call returns, to avoid confusion with a v in the outer scope (note the second subtraction of $\{v\}$). Finally, the union of A_0 is performed because v may be in A_0, in which case it has to be included in the answer. (**RTC** $v_1', v_2'\colon \varphi_1)(v_1, v_2)$ is handled similarly to $\exists\, v\colon \varphi_1$ and $\forall\, v\colon \varphi_1$.

3.3.2 Transitive-Closure Maintenance in Tree-Shaped Graphs

Consider a binary instrumentation relation p, defined by $\psi_p(v_1, v_2) \equiv (\textbf{RTC}\ v_1', v_2'\colon \varphi_1)(v_1, v_2)$. If the graph defined by φ_1 is not only acyclic but is tree-shaped, it is possible to take advantage of this fact.[4] This fact has no bearing on the maintenance formula that reflects a positive unit-size change $\Delta^+[\varphi_1]$ to the relation φ_1 in the values of the relation p (see Formula (3.8)). However, it allows a negative unit-size change $\Delta^-[\varphi_1]$ to the relation φ_1 to be reflected in the values of the relation p in a more efficient manner. In a tree-shaped graph, there exists at most one path between a pair of nodes; if that path goes through the φ_1 edge to be deleted, it should be removed (cf. Formula (3.10)):

$$\mathbf{F}_{st}[p](v_1, v_2) = p(v_1, v_2) \wedge \neg(\exists\, v_1', v_2'\colon p(v_1, v_1') \wedge \Delta^-_{st}[\varphi_1](v_1', v_2') \wedge p(v_2', v_2)). \tag{3.11}$$

Fig. 3.14 extends the method for generating relation-maintenance formulas to handle instrumentation relations specified via the **RTC** of a binary formula that defines a tree-shaped graph. Fig. 3.14 recasts Eqn. (3.11) as a finite-difference expression $\Delta^-_{st}[\psi_p]$.

When comparing the techniques of Sect. 3.3.1 for the maintenance of the **RTC** of a binary formula φ_1 with those presented in this section, we will refer to the method of Sect. 3.3.1 as *acyclic-φ_1 maintenance* and the method of this section as *tree-shaped-φ_1 maintenance*.

[4] The special-case maintenance strategy that we describe in this section also applies only in the case that the change to the graph is a single edge addition or deletion (but not both). We rely on the test described in Sect. 3.3.1.1 to ensure that this is the case.

φ	$\Delta^{-}_{st}[\varphi]$	
$p(w_1, \ldots, w_k)$, $p \in \mathcal{I}$	$((\exists\, v\colon \Delta^{-}_{st}[\varphi_1]) \wedge p)\{w_1, \ldots, w_k\}$	if $\psi_p \equiv \forall\, v\colon \varphi_1$
	$(\exists\, v'_1, v'_2\colon p(v_1, v'_1) \wedge \Delta^{-}_{st}[\varphi_1](v'_1, v'_2) \wedge p(v'_2, v_2))\{w_1, w_2\}$	if $\psi_p \equiv$ (**RTC** $v'_1, v'_2\colon \varphi_1)(v_1, v_2)$
	$\Delta^{-}_{st}[\psi_p]\{w_1, \ldots, w_k\}$	otherwise

Figure 3.14 Extension of the finite-differencing method from Fig. 3.5 to cover **RTC** formulas, for unit-sized changes to a tree-shaped graph defined by φ_1. The finite-difference expression $\Delta^{+}_{st}[\psi_p]$ is as defined in Fig. 3.10.

3.3.3 Reachability Maintenance in Deterministic Graphs

A *deterministic graph* is a graph in which every node has outdegree at most one. If the graph defined by φ_1 is deterministic, it is possible to give first-order formulas that maintain reachability information in the graph in response to the addition or deletion of a single φ_1-edge.

3.3.3.1 Abstractions of Possibly-Cyclic Linked Lists

The class of deterministic graphs corresponds exactly to the set of possibly-cyclic linked lists. In particular, we will illustrate our techniques on *panhandle lists*, i.e., linked lists that contain a cycle but in which at least the head of the list is not part of the cycle. (The lists shown in Fig. 3.15 are examples of panhandle lists.) Fig. 2.3 gives the definition of a C linked-list datatype, and lists the core relations that would be used to represent the stores manipulated by programs that use type `List`, such as the stores in Fig. 3.15.

Fig. 3.16(a) shows 2-valued structure $S_{3.16}$, which represents the store of Fig. 3.15(a) using the relations of Fig. 2.3.[5] Fig. 3.16(b) shows 2-valued structure $S_{3.16}$, which represents the store of Fig. 3.15(a) using the core relations of Fig. 2.3, as well as the instrumentation relations of Fig. 2.6.

If all unary relations are abstraction relations ($\mathcal{A} = \mathcal{R}_1$), the canonical abstraction of 2-valued logical structure $S_{3.16}$ is $S_{3.17}$, shown in Fig. 3.17, with list nodes corresponding to u_2 and u_3 in $S_{3.16}$ represented by the summary individual u_2 of $S_{3.17}$ and list nodes corresponding to u_5 and u_6

[5] We will not show the *dle* relation in the rest of this chapter because it is not relevant to the problem of reachability maintenance.

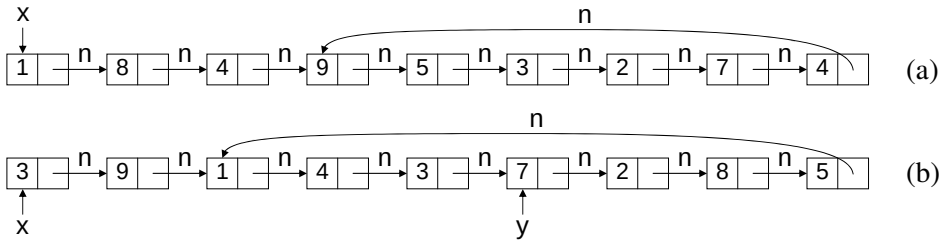


Figure 3.15 Possible stores for *panhandle* linked lists. (a) A panhandle list pointed to by x. We will refer to lists of this shape as type-X lists. (b) A panhandle list pointed to by x with y pointing into the middle of the cycle. We will refer to lists of this shape as type-XY lists.

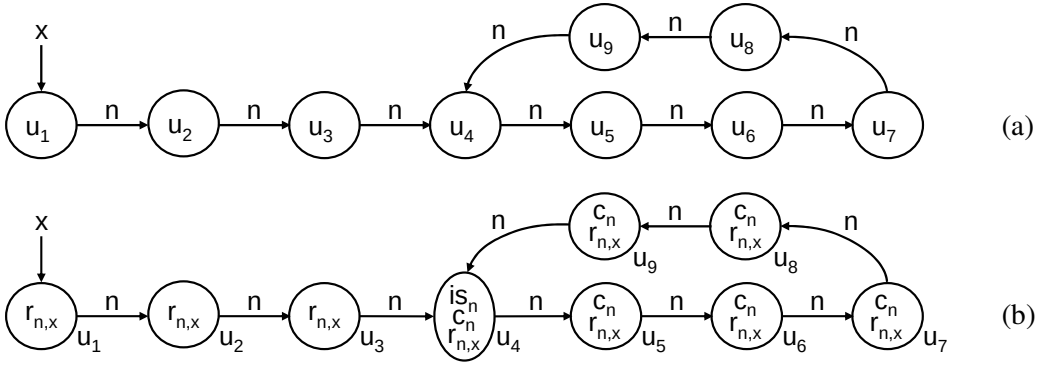


Figure 3.16 A logical structure $S_{3.16}$ that represents the store shown in Fig. 3.15(a) in graphical form: (a) $S_{3.16}$ with relations of Fig. 2.3; (b) $S_{3.16}$ with relations of Figs. 2.3 and 2.6. (Transitive-closure relation t_n has been omitted to reduce clutter.)

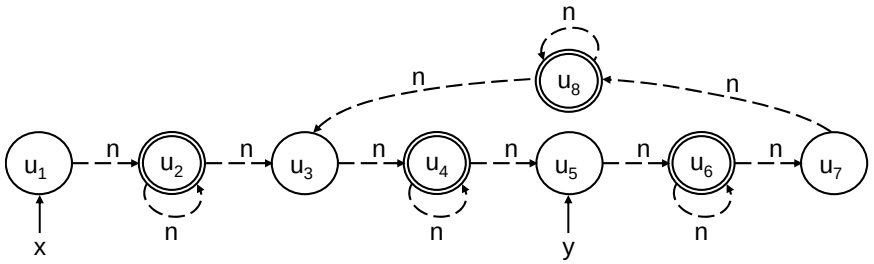


Figure 3.18 Logical structure $S_{3.18}$ that represents type-XY panhandle lists, such as the store of Fig. 3.15(b). The relations of Fig. 2.6 have been omitted to reduce clutter. Their values are as expected for a type-XY list: $r_{n,x}$ holds for all nodes, $r_{n,y}$ and c_n hold for all nodes on the cycle, and is_n holds for u_3.

in $S_{3.16}$ represented by the summary individual u_4 of $S_{3.17}$. $S_{3.17}$ represents any type-X panhandle list with at least two nodes in the panhandle and at least two nodes in the cycle.

3.3.3.2 Reachability Maintenance in Possibly-Cyclic Linked Lists

Unfortunately, the relations defined in Figs. 2.3 and 2.6 do not permit precise maintenance of reachability information, such as relation $r_{n,x}$, in possibly-cyclic lists. A difficulty arises when reachability information has to be updated to reflect the deletion of an n edge on a cycle (e.g., as a result of statement `y->n = NULL`). With the relations defined in Figs. 2.3 and 2.6, such an update requires the recomputation of a transitive-closure formula, which generally results in a drastic loss of precision in the presence of abstraction.

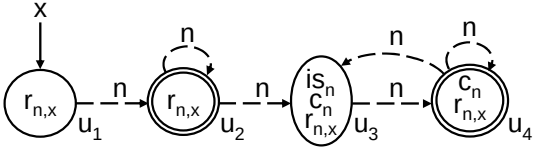


Figure 3.17 A 3-valued structure $S_{3.17}$ that is the canonical abstraction of structure $S_{3.16}$. In addition to $S_{3.16}$, $S_{3.17}$ represents any type-X panhandle list with at least two nodes in the panhandle and at least two nodes in the cycle.

We demonstrate the issue on panhandle lists represented by the abstract structure $S_{3.18}$ shown in Fig. 3.18, i.e., lists of type XY. Statement `y->n = NULL` has the effect of deleting the `n` edge leaving u_5, thus making the nodes represented by u_6, u_7, and u_8 unreachable from `x`.[6] Note that a first-order-logic formula over the relations of Figs. 2.3 and 2.6 cannot distinguish the list nodes represented by u_4 from those represented by u_6, u_7, and u_8: all of those nodes are reachable

[6]Clearly, all nodes except u_5 also become unreachable from `y`.

from both x and y, none of those nodes are shared, and all of them lie on a cycle. Our inability to characterize the group of nodes represented by u_4 via a first-order formula requires the maintenance formula for the reachability relation $r_{n,x}$ to recompute some transitive-closure information, e.g., the transitive-closure subformula of the definition of $r_{n,x}$, namely, $n^*(v_1, v)$. However, in the presence of abstraction, recomputing transitive-closure formulas often yields $1/2$. For instance, in $S_{3.18}$, formula $n^*(v_1, v)$ evaluates to $1/2$ under assignment $[v_1 \mapsto u_1, v \mapsto u_4]$ because of the many $1/2$ values of relation n (see the dashed edges connecting u_1 with u_2, for example).

The essence of a solution that enables maintaining reachability relations for possibly-cyclic lists in first-order logic is to find a way to break the symmetry of each cycle. The basic idea for a solution was suggested to us by William Hesse and Neil Immerman. It consists of maintaining a spanning-tree representation of a possibly-cyclic list. Reachability in such a representation can be maintained using first-order-logic formulas. Reachability in the actual list can be expressed in first-order logic based on the spanning-tree representation. We now explain our approach and highlight some differences with the approach taken by Hesse [37].

Our approach relies on the introduction of additional core and instrumentation relations. We extend the set of core relations (Fig. 2.3) with unary relation roc_n, which designates one node on each cycle to be the **r**epresentative **o**f the **c**ycle. (We refer to such a node as a roc_n node.) Relation roc_n is used for tracking a unique *cut edge* on each cycle, which allows the maintenance of a spanning tree. Fig. 3.20(a) shows 2-valued structure $S_{3.20}$, which represents the store of Fig. 3.15(a) using the extended set of core relations. Here, we let u_7 be the roc_n node. In general, we simply require that exactly one node on each cycle be designated as a roc_n node. Later in this section we describe how we ensure this.

Fig. 3.19 lists the extended set of instrumentation relations. We divide our description of the abstraction based on the new set of relations into three parts, which describe (i) how the relations of Fig. 3.19 define *directed* spanning forests, (ii) how we maintain precision on a cycle in the presence of abstraction, and (iii) how we generate maintenance formulas for instrumentation relations *automatically*. The three parts highlight the differences between our approach and that of Hesse.

p	**Intended Meaning**	**Defining Formula**
$is_n(v)$	Do `n` fields of two or more list nodes point to v?	$\exists\, v_1, v_2 \colon n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$
$sfe_n(v_1, v_2)$	Is there an `n` edge from v_2 to v_1 (assuming that v_2 is not a roc_n node)	$n(v_2, v_1) \wedge \neg roc_n(v_2)$
$sfp_n(v_1, v_2)$	Is v_2 reachable from v_1 along sfe_n edges?	$sfe_n^*(v_1, v_2)$
$t_n(v_1, v_2)$	Is v_2 reachable from v_1 along `n` fields?	$sfp_n(v_2, v_1) \vee \exists\, u, w \colon \begin{pmatrix} sfp_n(u, v_1) \wedge \\ roc_n(u) \wedge n(u, w) \\ \wedge\, sfp_n(v_2, w) \end{pmatrix}$
$r_{n,x}(v)$	Is v reachable from pointer variable `x` along `n` fields?	$\exists\, v_1 \colon x(v_1) \wedge t_n(v_1, v)$
$c_n(v)$	Is v on a directed cycle of `n` fields?	$\exists\, v_1, v_2 \colon roc_n(v_1) \wedge n(v_1, v_2) \wedge sfp_n(v, v_2)$
$pr_x(v)$	Does v lie on an sfe_n path from `x` (does v **pr**ecede `x` on an `n`-path to a roc_n node)?	$\exists\, v_1 \colon x(v_1) \wedge sfp_n(v_1, v)$
$pr_{is}(v)$	Does v lie on an sfe_n path from a shared node (does v **pr**ecede a shared node on an `n`-path to a roc_n node)?	$\exists\, v_1 \colon is_n(v_1) \wedge sfp_n(v_1, v)$

Figure 3.19 Defining formulas of instrumentation relations. The sharing relation is_n is defined as in Fig. 2.6. Relations t_n, $r_{n,x}$, and c_n are redefined via first-order-logic formulas in terms of other relations.

Defining Directed Spanning Forests Instrumentation relation sfe_n—*sfe* stands for **s**panning-**f**orest **e**dge—is used to maintain the set of edges that form a spanning forest of list nodes. In Hesse's work, the spanning-forest edges retain the direction of the n edges. As a result, he maintains spanning forests, in which the edges lead to the roots of the spanning forest, which are designated as roc_n nodes in our abstraction. For clarity of presentation, we define sfe_n to be the reverse of n edges (all but the edges leaving roc_n nodes). The graph defined by the sfe_n relation then defines a *directed* spanning forest with roc_n nodes as spanning-forest roots and with the usual orientation of spanning-forest edges.

Instrumentation relation sfp_n—*sfp* stands for **s**panning-**f**orest **p**ath—is used to maintain the set of paths in the spanning forest of list nodes. Binary reachability in the actual lists (see relation t_n in Fig. 3.19) can be defined in terms of n, roc_n, and sfp_n using a first-order-logic formula: v_2 is reachable from v_1 if there is a spanning-forest path from v_2 to v_1 or there is a pair of spanning-forest paths, one from the source of a cut edge (a roc_n node) to v_1 and the other from v_2 to the target of the cut edge (the `n`-successor of the same roc_n node).

Unary reachability relations $r_{n,x}$ and the cyclicity relation c_n can be defined via first-order formulas, as well. We defined $r_{n,x}$ in terms of binary reachability relation t_n. While we could define c_n in terms of t_n, as well, we chose another simple definition by observing that a node lies on a cycle if and only if there is a spanning-forest path from it to the target of a cut edge (the `n`-successor of a roc_n node).

Fig. 3.20(b) shows 2-valued structure $S_{3.20}$, which represents the store of Fig. 3.15(a) using the extended set of core and instrumentation relations. The relations pr_x and pr_{is} will be explained shortly.

Preserving Node Ordering on a Cycle in the Presence of Abstraction The fact that our techniques need to be applicable in the presence of abstraction introduces a complication that is not present in the setting studied by Hesse. His concern was with the expressibility of certain properties within the confines of a logic with certain syntactic restrictions. Our concern is with the ability to maintain precision in the framework of canonical abstraction.

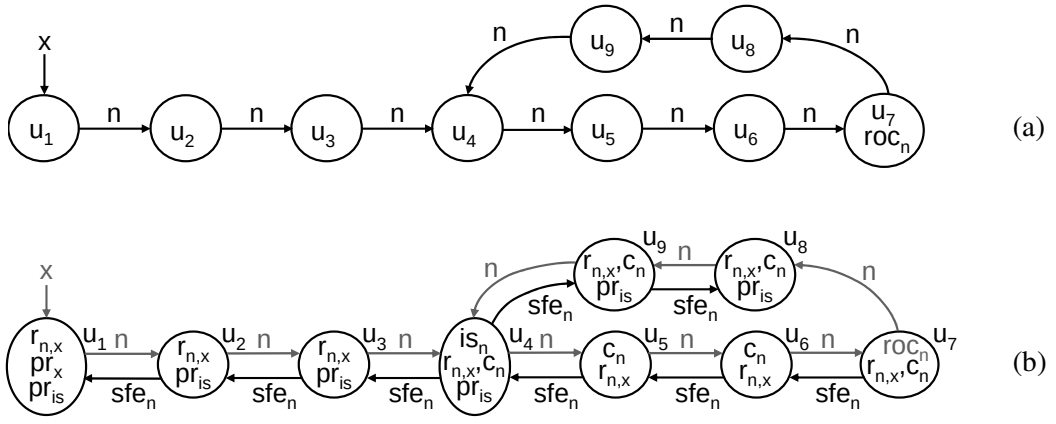


Figure 3.20 A logical structure $S_{3.20}$ that represents the store shown in Fig. 3.15(a) in graphical form: (a) $S_{3.20}$ with the extended set of core relations.(b) $S_{3.20}$ with the extended set of core and instrumentation relations (core relations appear in grey). Transitive-closure relations sfp_n and t_n have been omitted to reduce clutter. The values of the transitive-closure relations can be readily seen from the graphical representation of relations sfe_n and n. For instance, node u_5 is related via the sfp_n relation to itself and all nodes appearing to the left or above it in the pictorial representation.

Unary reachability relations $r_{n,x}$ (one for every program variable x) play a crucial role in the analysis of programs that manipulate acyclic linked lists. In addition to keeping disjoint lists summarized separately, they keep list nodes that have been visited during a traversal summarized separately from nodes that have not been visited: if x is the pointer used to traverse the list, then the nodes that have been visited will have value 0 for relation $r_{n,x}$, while the nodes that have not been visited will have value 1. If a list contains a cycle, then all nodes on the cycle are reachable from the same set of variables, namely, all variables that point to any node in that list. As a result, the instrumentation relations discussed thus far cannot prevent nodes u_4, u_6, and u_8 of $S_{3.18}$ shown in Fig. 3.18 from being summarized together. Thus, assuming that u_7 is the roc_n node, the canonical abstraction of $S_{3.18}$ is the 3-valued structure $S_{3.21}$ shown in Fig. 3.21. The nodes represented by u_4, u_6, and u_8 of $S_{3.18}$ are represented by the single summary individual u_6 in $S_{3.21}$. The symmetry hides all information about the order of traversal via pointer variable y. Moreover, the values of the sfp_n relation (not shown in Fig. 3.21) lose precision because ancestors of the shared node in the spanning tree are summarized together with its descendants in the spanning tree.

We break the symmetry of the nodes on a cycle using a general mechanism via unary properties akin to unary reachability relations $r_{n,x}$. In the definitions of relations pr_x of Fig. 3.19, full reachability (relation t_n) has been replaced with spanning-forest reachability (relation sfp_n). The relations pr_x distinguish nodes according to whether or not they are reachable from program variable x along spanning-forest edges. The relation pr_{is} is defined similarly but using instrumentation relation is_n; pr_{is} partitions the nodes of a panhandle list into ancestors and descendants of the shared node in the spanning tree. Fig. 3.22 shows structure $S_{3.22}$ that is the canonical abstraction of $S_{3.18}$

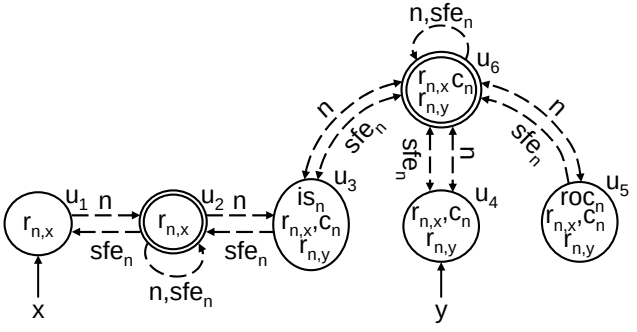


Figure 3.21 A 3-valued structure $S_{3.21}$ that is the canonical abstraction of structure $S_{3.18}$ if relations pr_x and pr_{is} are not added to $\mathcal{A}$ and node u_7 is the roc_n node.

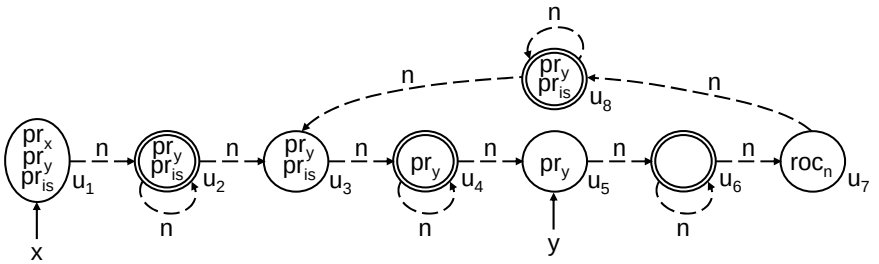


Figure 3.22 A 3-valued structure $S_{3.22}$ that is the canonical abstraction of structure $S_{3.18}$ if node u_7 is the roc_n node. $S_{3.22}$ represents panhandle lists of type XY, such as the store of Fig. 3.15(b). The only instrumentation relations shown in the figure are pr_x, pr_y, and pr_{is}. As in structure $S_{3.18}$ shown in Fig. 3.18, $r_{n,x}$ holds for all nodes, $r_{n,y}$ and c_n hold for all nodes on the cycle, and is_n holds for u_3.

of Fig. 3.18, assuming that u_7 is the roc_n node. In $S_{3.22}$, each of the nodes u_4, u_6, and u_8 has a distinct vector of values for the relations pr_y and pr_{is}, thus breaking the symmetry.

Automatic Generation of Maintenance Formulas for Instrumentation Relations In his thesis, Hesse gives hand-specified update formulas for a collection of relations that are used for maintaining a spanning-forest representation of possibly-cyclic linked lists. Instead of specifying them by hand, we rely on finite differencing, as described in previous sections of this chapter, to generate relation-maintenance formulas for all instrumentation relations. Finite-differencing-generated maintenance formulas have been effective in maintaining all relations defined via first-order-logic formulas, i.e., all relations of Fig. 3.19 except sfp_n. Additionally, under certain conditions, finite-differencing-generated maintenance formulas have been effective in maintaining relations defined via the reflexive transitive closure of binary relations. The necessary conditions for this technique to be applicable for the maintenance of relation sfp_n are:

Graph-shape condition: the graph defined by sfe_n needs to be acyclic or tree-shaped;

Unit-size-change condition: the change to the graph effected by any program statement needs to be a single-edge addition or deletion (but not both).

The graph-shape condition applies in our setting because the graph defined by sfe_n defines a spanning forest (which is both acyclic and tree-shaped). The unit-size-change condition requires some discussion.

The relation sfe_n is defined in terms of n and roc_n. While we have not yet discussed the relation-transfer formulas for core relation roc_n, it should be clear that the value of the relation roc_n should only change in response to a change in the value of a node's `n` field. There are two types of statements that change the value of the `n` field and thus may have an effect that should be reflected in the value of the sfe_n relation, namely, statements of the forms `x->n = NULL` and `x->n = y`. The former destroys the n edge leaving the node pointed to by `x`, and the latter creates a new n-connection from the node pointed to by `x` to the node pointed to by `y`. While both of these statements add or remove a single edge of the n relation, it is not necessarily the case that they add or remove a single edge of the sfe_n relation. When interpreted on logical structure $S_{3.22}$ of Fig. 3.22, statement `y->n = NULL` has the effect of deleting the n edge leaving u_5, an action that should result in the deletion of the sfe_n edge entering u_5 (not shown in the figure). However, to preserve the spanning-forest representation, we need to ensure that roc_n holds only for nodes that lie on a cycle and that sfe_n represents spanning-forest edges. This requires setting the value of roc_n for u_7 to 0 and adding an sfe_n edge from u_8 to u_7. Because, as this example illustrates, a language statement may result in the deletion of one sfe_n edge and the addition of another, neither of our techniques for maintaining instrumentation relations defined via the transitive-closure operator (Sects. 3.3.1 and 3.3.2) applies.

To work around this problem, we apply each transformer associated with statements `x->n = NULL` and `x->n = y` in two phases. In one phase, we apply the part of the transformer that corresponds to the relation n and reflect it in the values of all instrumentation relations. In the other phase, we apply the part of the transformer that corresponds to the relation roc_n and reflect it in the values of all instrumentation relations. As we explain below, each phase of the two transformers satisfies the requirement that the change add or delete a single edge of the sfe_n relation. Additionally, by paying attention to the order of phases, we ensure that the graph defined by the relation sfe_n remains acyclic and tree-shaped throughout the application of the transformers.

To preserve the graph-shape condition in the case of statement `x->n = NULL`, we apply the part of the transformer that corresponds to the relation n first:

$$\tau_{n,\texttt{x->n = NULL}}(v_1, v_2) = n(v_1, v_2) \wedge \neg x(v_1). \tag{3.12}$$

Unless `x` points to a roc_n node (or `x->n` is `NULL`), this phase results in the deletion of the sfe_n edge that enters the node pointed to by `x`. In the second phase, we apply the part of the transformer that corresponds to the relation roc_n:

$$\tau_{roc_n,\texttt{x->n = NULL}}(v) = roc_n(v) \wedge \exists\, v_1 \colon n(v, v_1) \wedge sfp_n(v, v_1). \tag{3.13}$$

This phase sets the roc_n property of the source n_s of a cut edge to 0, if there is no longer a spanning-forest path from n_s to the target n_t of the same cut edge. When this happens and `x` does not point to n_s, i.e., the cut edge is not being deleted, this phase results in the addition of an sfe_n edge from n_t to n_s.

To preserve the graph-shape condition in the case of statement `x->n = y`, we apply the part of the transformer that corresponds to the relation roc_n first:

$$\tau_{roc_n,\texttt{x->n = NULL}}(v) = roc_n(v) \vee (x(v) \wedge \exists\, v_1 \colon y(v_1) \wedge sfp_n(v, v_1)). \tag{3.14}$$

If there is a spanning-forest path from the node n_x, pointed to by `x`, to the node n_y, pointed to by `y`, the statement creates a new cycle in the data structure. The update of Formula (3.14) sets the roc_n property of n_x to 1, thus making n_x the source of a new cut edge and n_y the target of the cut edge. Because there was no n edge from n_x to n_y prior to the execution of this statement,[7] this phase results in no change to the sfe_n relation. In the second phase, we apply the part of the transformer that corresponds to the relation n:

$$\tau_{n,\texttt{x->n = NULL}}(v_1, v_2) = n(v_1, v_2) \vee (x(v_1) \wedge y(v_2)). \tag{3.15}$$

Unless the node pointed to by `x` became a roc_n node in the first phase, this phase results in the addition of an sfe_n edge from n_y to n_x.

[7] By normalizing procedures to include a statement of the form `x->n = NULL` prior to a statement of the form `x->n = y`, we ensure that `x->n` is always `NULL` prior to the latter assignment.

The break-up of the transformers corresponding to statements `x->n = NULL` and `x->n = y` into two phases, as described above, ensures that the sfe_n relation remains acyclic and tree-shaped throughout the analysis (the graph-shape condition) and that the change to the sfe_n relation effected by each phase is a unit-size change (the unit-size-change condition).[8] Thus, it is sound to maintain sfp_n ($= sfe_n^*$) via the techniques described in either Sect. 3.3.1 or 3.3.2. Additionally, it is also sound to maintain the remaining instrumentation relations via the techniques of Sect. 3.2 because the remaining relations are defined by first-order-logic formulas. Soundness guarantees that the stored values of instrumentation relations agree with the relations' defining formulas throughout the analysis. However, the stored values may not agree with the relations' *intended meanings*. For instance, if the `n`-transfer phase of the transformer for statement `x->n = NULL` removes a non-cut `n` edge on a cycle, the sfe_n relation will temporarily not span the entire list. However, as long as we do not query the results of abstract interpretation between the phases of a two-phase transformer, the stored values of instrumentation relations agree with the relations' intended meanings, as well as their defining formulas.

3.4 Experimental Evaluation

To evaluate the techniques presented in this chapter, we extended TVLA to generate relation-maintenance formulas, and applied it to a test suite of 5 existing analysis specifications, involving 24 programs (see Fig. 3.23).

The test programs consisted of various operations on acyclic singly-linked lists, doubly-linked lists, binary trees, and binary-search trees, plus several sorting programs [53]. The system was used to verify some partial-correctness properties of the test programs. For instance, Reverse, an in-situ list-reversal program, must preserve list properties and lose no elements; InsertSorted and DeleteSorted must preserve binary-search-tree properties; InsertSort must return a sorted list; Good Flow must not allow high-security input data to flow to a low-security output channel. Chapter 5 discusses the verification of stronger properties, such as the *partial correctness* of several of the

[8] The test described in Sect. 3.3.1.1 validates our reasoning about the unit-size-change condition.

Category	Test Program	# of non-identity maintenance formulas				Performance				
						Analysis Time (sec.)			% increase	
		schemas			# inst.	Ref.	FD		FD	
		total	TC	non-TC			acyc.	tree	acyc.	tree
SLL Shape Analysis	Search	2	0	2	2	0.30	0.30	0.31	1.10	1.90
	GetLast	3	0	3	4	0.31	0.32	0.32	2.23	2.22
	DeleteAll	11	2	9	15	0.30	0.32	0.30	4.97	-0.13
	Reverse	12	2	10	16	0.43	0.49	0.44	12.69	1.99
	Create	11	2	9	21	0.28	0.31	0.28	9.61	-0.60
	Delete	12	2	10	39	1.13	2.13	1.23	87.90	7.76
	Merge	11	2	9	64	1.77	3.67	1.96	107.27	10.42
	Insert	12	2	10	72	1.19	2.03	1.31	70.43	9.67
DLL Shape Analysis	Append	15	2	13	50	1.76	1.78	1.77	1.13	0.57
	Delete	16	2	14	74	8.35	8.78	8.38	5.15	0.36
	Splice	15	2	13	96	1.06	1.69	1.10	59.70	3.79
Binary Tree Shape Analysis	InsertSorted	13	2	11	43	1.25	1.28	1.28	1.97	1.54
	Lindstrom	10	2	8	43	40.44	82.29	41.48	103.47	2.57
	DSW	10	2	8	52	101.30	180.20	109.51	77.89	8.15
	DeleteSorted	13	2	11	554	75.26	409.31	97.71	443.85	29.69
SLL Sorting	ReverseSorted	18	2	16	23	0.47	0.54	0.49	13.05	2.58
	BubbleSort	18	2	16	80	5.74	8.91	6.42	55.32	11.77
	BubbleSortBug	18	2	16	80	5.41	7.61	6.01	40.75	11.14
	InsertSortBug2	18	2	16	87	5.19	17.57	6.09	238.55	17.04
	InsertSort	18	2	16	88	5.65	18.55	6.66	228.26	17.95
	InsertSortBug1	18	2	16	88	18.94	32.93	20.25	73.84	7.27
	MergeSorted	18	2	16	91	2.26	4.22	2.53	86.35	11.46
Information Flow	Good Flow	12	2	10	66	13.59	23.28	15.37	71.30	13.59
	Bad Flow	12	2	10	86	78.05	180.85	94.92	131.70	21.79

Figure 3.23 Results from using hand-crafted vs. automatically generated maintenance formulas for instrumentation relations

algorithms. Lindstrom and DSW are two variants of Deutsch-Schorr-Waite, a constant-space tree-traversal algorithm that uses destructive pointer rotation. For Lindstrom and DSW, we verified that the algorithms have no unsafe pointer operations or memory leaks, and that the data structure produced at the end is, in fact, a binary tree. Chapter 6 discusses the verification of the *total correctness* of Deutsch-Schorr-Waite, i.e., that the binary tree produced at the end is identical to the input tree and that the algorithm terminates.

A few of the programs contained bugs: for instance, InsertSortBug2 is an insert-sort program that ignores the first element of the list; BubbleBug is a bubble-sort program with an incorrect condition for swapping elements, which causes an infinite loop if the input list contains duplicate data values. (See [26, 53, 54] for more details.)

In TVLA, the operational semantics of a programming language is defined by specifying, for each kind of statement, an *action schema* to be used on outgoing CFG edges. Action schemas are instantiated according to a program's statement instances to create the CFG. For each combination of action schema and instrumentation relation, a *maintenance-formula schema* must be provided. The number of non-identity maintenance-formula schemas is reported in columns 3–5 of Fig. 3.23, broken down in columns 4–5 into those whose defining formula contains an occurrence of **RTC**, and those that do not. Relation-maintenance formulas produced by finite differencing are generally larger than the hand-crafted ones. Because this affects analysis time, the number of instances of non-identity maintenance-formula schemas is a meaningful size measure for our experiments. These numbers appear in column 6. The number of instances of non-identity schemas for `DeleteSorted` is high because `DeleteSorted` includes three inline expansions of the routine that finds the tree node that takes the place of the deleted node.[9]

The data structures manipulated by all programs in our test suite are acyclic and tree-shaped, thus acyclic reachability maintenance (i.e., the techniques of Sect. 3.3.1), as well as tree-shaped reachability maintenance (i.e., the techniques of Sect. 3.3.2), apply for the maintenance of reachability relations. In the absense of hand-crafted maintenance formulas for reachability relations

[9] Work on interprocedural shape analysis provides a solution that does not require inline-expanded programs [42, 83].

in possibly-cyclic linked lists, we could not extend our experiments to cover the techniques of Sect. 3.3.3.2. Instead, we validate those techniques as part of the verification of properties of `Reverse` when applied to possibly-cyclic linked lists (see Sect. 5.4.1).

For each program in the test suite, we first ran the analysis using hand-crafted maintenance formulas, to obtain a reference answer in which CFG nodes were annotated with their final sets of logical structures. We then ran the analysis using automatically generated maintenance formulas with acyclic reachability maintenance and compared the result against the reference answer. For all 24 test programs, the analysis using automatically generated formulas yielded answers identical to the reference answers. Finally, we ran the analysis using automatically generated maintenance formulas with tree-shaped reachability maintenance and compared the result against the reference answer. Again, for all 24 test programs, the analysis using automatically generated formulas yielded answers identical to the reference answers.

Columns 7–11 show performance data, which were collected on a 3GHz PC with 3.7GB of RAM running CentOS 4 Linux. The column labeled "Ref." gives the references times. Columns labeled "acyc." give the data for the analyses that used automatically generated maintenance formulas with acyclic reachability maintenance. Columns labeled "tree" give the data for the analyses that used automatically generated maintenance formulas with tree-shaped reachability maintenance. In each case, five runs were made; the longest and shortest times were discarded from each set, and the remaining three averaged. The geometric mean of the slowdowns when using the automatically generated formulas with acyclic reachability maintenance was approximately 60%, with a median of 55%, mainly due to the fact that the automatically generated formulas are larger than the hand-crafted ones. The maximum slowdown was 444%. The highest slowdowns occured in analyses of programs that involved deletions of edges in a data structure's graph.

Because the edge-deletion maintenance formulas produced by the tree-shaped reachability-maintenance technique are much smaller than those that are produced by acyclic reachability maintenance, our expectation was that the use of tree-shaped reachability-maintenance formulas would cause a much smaller slowdown. This expectation was confirmed: the geometric mean of the

slowdowns when using the automatically generated formulas with tree-shaped reachability maintenance was approximately 8%, with a median of 7%. The maximum slowdown was 30%.[10] A few analyses were actually faster with the automatically generated formulas; these speedups are either due to random variation or are accidental benefits of subformula orderings that are advantageous for short-circuit evaluation.

These results are encouraging. At least for abstractions of several common data structures, they suggest that the algorithm for generating relation-maintenance formulas from Sects. 3.2 and 3.3 is capable of automatically generating formulas that (i) are as precise as the hand-crafted ones, and (ii) have a tolerable effect on runtime performance.

The extended version of TVLA also uncovered several bugs in the hand-crafted formulas. A maintenance formula of the form $\mu_{p,st}(v_1, \ldots, v_k) = p(v_1, \ldots, v_k)$ is called an *identity relation-maintenance formula*. For each identity relation-maintenance formula in the hand-crafted specification, we checked that (after simplification) the corresponding generated relation-maintenance formula was also an identity formula. Each inconsistency turned out to be an error in the hand-crafted specification. We also found one instance of an incorrect non-identity hand-crafted maintenance formula. (The measurements reported in Fig. 3.23 are based on corrected hand-crafted specifications.)

3.5 Related Work

A weakness of past incarnations of TVLA has been the need for the user to define relation-maintenance formulas that specify how each statement affects each instrumentation relation. Recent criticisms of TVLA based on this deficiency are no longer valid [3, 66], at least for analyses that can be defined using formulas that define acyclic relations (and also for some classes of formulas that define cyclic relations). With the algorithm presented in Sects. 3.2 and 3.3, the user's responsibility is merely to write the ψ_p formulas; appropriate relation-maintenance formulas are created automatically.

[10] We expect that some simple optimizations, such as caching the results from evaluating subformulas, could reduce the slowdown further.

Graf and Saïdi [32] showed that theorem provers can be used to generate best abstract transformers [21] for abstract domains that are fixed, finite, Cartesian products of Boolean values. (The use of such domains is known as *predicate abstraction*; predicate abstraction is also used in SLAM [3] and other systems [23].) In contrast, the abstract transformers created using the algorithm described in Sects. 3.2 and 3.3 are not best transformers; however, this algorithm uses only very simple, linear-time, recursive tree-traversal procedures, whereas the theorem provers used in predicate abstraction are not even guaranteed to terminate. Moreover, our setting makes available much richer abstract domains than the ones offered by predicate abstraction, and experience to date has been that very little precision is lost (using only *good* abstract transformers) once the right instrumentation relations have been identified.

Paige studied how finite-differencing transformations of applicative set-former expressions could be exploited to optimize loops in very-high-level languages, such as SETL [71]. Liu et al. used related program-transformation methods in the setting of a functional programming language to derive incremental algorithms for various problems from the specifications of exhaustive algorithms [57, 58]. In their work, the goal is to maintain the value of a function $F(x)$ as the input x undergoes small changes. The methods described in Sects. 3.2 and 3.3 address a similar kind of incremental-computation problem, except that the language in which the exhaustive and incremental versions of the problem are expressed is first-order logic with reflexive transitive closure.

The finite-differencing operators defined in Sects. 3.2 and 3.3 are most closely related to a number of previous papers on logic and databases: finite-difference operators for the propositional case were studied by Akers [1] and Sharir [89]. Previous work on incrementally maintaining materialized views in databases [33], "first-order incremental evaluation schemes (FOIES)" [24], and "dynamic descriptive complexity" [73] has also addressed the problem of maintaining one or more auxiliary relations after new tuples are inserted into or deleted from the base relations. In databases, view maintenance is solely an optimization; the correct information can always be obtained by reevaluating the formula. In the abstract-interpretation context, where abstraction has been performed, this is no longer true: reevaluating a formula in the local (3-valued) state can lead to a drastic loss of precision. Thus, one aspect that sets our work apart from previous work is the

φ	$\Delta_{st}[\varphi]$
$\mathbf{1}$	$\mathbf{0}$
$\mathbf{0}$	$\mathbf{0}$
$p(w_1, \ldots, w_k), p \in \mathcal{C}$	$(\tau_{p,st} \oplus p)\{w_1, \ldots, w_k\}$
$p(w_1, \ldots, w_k), p \in \mathcal{I}$	$\Delta_{st}[\psi_p]\{w_1, \ldots, w_k\}$
$\varphi_1 \oplus \varphi_2$	$\Delta_{st}[\varphi_1] \oplus \Delta_{st}[\varphi_2]$
$\varphi_1 \wedge \varphi_2$	$(\Delta_{st}[\varphi_1] \wedge \varphi_2) \oplus (\varphi_1 \wedge \Delta_{st}[\varphi_2]) \oplus (\Delta_{st}[\varphi_1] \wedge \Delta_{st}[\varphi_2])$
$\forall v \colon \varphi_1$	$(\forall v \colon \varphi_1)\ ?\ (\exists v \colon \Delta_{st}[\varphi_1]) : (\forall v \colon \varphi_1 \oplus \Delta_{st}[\varphi_1])$

Figure 3.24 An alternative finite-differencing scheme for first-order formulas

goal of developing a finite-differencing transformation suitable for use when abstraction has been performed.

Not all finite-differencing transformations that are correct in 2-valued logic (i.e., satisfy Theorem 3.5), are appropriate for use in 3-valued logic. For instance, Fig. 3.24 presents an alternative finite-differencing scheme for first-order formulas. In this scheme, $\Delta_{st}[\varphi]$ captures both the negative and positive changes to φ's value. With Fig. 3.24, the maintenance formula for instrumentation relation p is

$$\mu_{p,st} \stackrel{\text{def}}{=} p \oplus \Delta_{st}[\psi_p], \tag{3.16}$$

where $\oplus$ denotes exclusive-or. However, in 3-valued logic, we have $1/2 \oplus V = 1/2$, regardless of whether V is 0, 1, or $1/2$. Consequently, Eqn. (3.16) has the unfortunate property that if $p(u) = 1/2$, then $\mu_{p,st}$ evaluates to $1/2$ on u, and $p(u)$ becomes "pinned" to the indefinite value $1/2$; it will have the value $1/2$ in all successor structures S', in all successors of S', and so on. With Eqn. (3.16), $p(u)$ can *never* reacquire a definite value.

In contrast, the maintenance formulas created using the finite-differencing scheme of Fig. 3.5 do not have this trouble because they have the form $p\ ?\ \neg\Delta^{-}_{st}[\psi_p] : \Delta^{+}_{st}[\psi_p]$. The use of if-then-else allows $p(u)$ to reacquire a definite value after it has been set to $1/2$: if $p(u)$ is $1/2$, $\mu_{p,st}$ evaluates to a definite value on u if $[\![\Delta^{-}_{st}[\psi_p(v)]]\!]^S_3([v \mapsto u])$ is 1 and $[\![\Delta^{+}_{st}[\psi_p(v)]]\!]^S_3([v \mapsto u])$ is 0, or vice versa.

In Sect. 3.3.3.2, we compared our work with that of William Hesse, which is closest in spirit to our techniques for maintaining reachability information in possibly-cyclic linked lists. Below, we discuss a few approaches that bear resemblance to ours in that they attempt to translate or simulate a data structure that cannot be handled by some core techniques into one that can.

The idea of using spanning-tree representations for specifying or reasoning about data structures that are "close to trees" is not new. Klarlund and Schwartzbach introduced *graph types*, which can be used to specify some common non-tree-shaped data structures in terms of a spanning-tree *backbone* and regular expressions that specify where non-backbone edges occur within the backbone [44]. Examples of data structures that can be specified by graph types are doubly-linked lists and threaded trees. A panhandle list cannot be specified by a graph type because in a graph type the location of each non-backbone edge has to be defined in terms of the backbone using a regular expression, and a regular expression cannot be used to specify the existence of a backedge to *some* node that occurs earlier in the list. In the PALE project [66], which incorporates work on graph types, automated reasoning about programs that manipulate data structures specified as graph types can be carried out using a decision procedure for monadic second-order logic. Unfortunately, the decision procedure has non-elementary complexity. An advantage of our approach over that of PALE is that we do not rely on the use of a decision procedure.

Immerman et al. presented *structure simulation*, a technique that broadens the applicability of decision procedures to a larger class of data structures [39]. Under certain conditions, it allows data structures that cannot be reasoned about using decidable logics to be translated into data structures that can, with the translation expressed as a first-order-logic formula. Unlike graph types, structure simulation is capable of specifying panhandle lists. However, this technique shares a limitation of graph types because it relies on decision procedures for automated reasoning about programs.

In [60], Manevich et al. specified abstractions (in canonical-abstraction and predicate-abstraction forms) for showing safety properties of programs that manipulate possibly-cyclic linked lists. By maintaining reachability within list segments that are not *interrupted* by nodes that are shared or pointed to by a variable, they are able to break the symmetry of a cycle. The definition of several key instrumentation relations in that work makes use of transitive-closure formulas that

cannot be handled precisely by finite differencing. As a result, a drawback of that work is the need to define some relation-maintenance formulas by hand. Another drawback is the difficulty of reasoning about reachability (in a list) from a program variable (see reachability relations $r_{n,x}$ of Fig. 3.19). Because in [60] reachability in a list has to be expressed in terms of reachability over a sequence of uninterrupted segments, a formula that expresses the reachability of node v from the program variable x in a list has to enumerate all permutations of other program variables that may act as interruptions on a path from x to v in the list.

A number of past approaches to the analysis of programs that manipulate linked lists relied on first-order axiomatizations of reachability information. All of these approaches involved the use of first-order-logic decision procedures. While our approach does not have this limitation, it is instructive to compare our work with those approaches that included mechanisms for breaking the symmetry on a cycle. Nelson defined a set of first-order axioms that describe the ternary reachability relation $r_n(u, v, w)$, which has the meaning: w is reachable from u along n edges without encountering v [70]. The use of this relation alone is not sufficient in our setting because in the presence of abstraction we require unary distinctions (such as the relations pr_x and pr_{is} of Fig. 3.19) to break the symmetry. Additionally, the maintenance of ternary relations is more expensive than the maintenance of binary relations. In [48], Lahiri and Qadeer specify a collection of first-order axioms that are sufficient to verify properties of procedures that perform a single change to a cyclic list, e.g., the removal of an element. They also verify properties of in-situ list reversal, albeit under the assumption that the input list is acyclic. (In Sect. 5.4.1, we describe a case study in which we use the techniques developed in Sect. 3.3.3.2 to verify properties of `Reverse` when applied to any linked list, including cyclic and panhandle lists.) They break the symmetry of cycles in a similar fashion to how it is done in [60]: the *blocking cells* of [48] are a subset of the interruptions of [60]. The blocking cells include only the set of *head variables*—program variables that act as heads of lists used in the program. This set has to be carefully maintained by the user to (i) satisfy the system's definition of acceptable (*well-founded*) lists, (ii) allow the system to verify useful postconditions, and (iii) avoid falling prey to the difficulty—that arises in [60]—of expressing reachability in the list. The current mechanism of [48] is insufficient for reasoning about

panhandle lists because the set of blocking cells does not include shared nodes. This limitation can be partially addressed by generalizing the set of blocking cells to mimic interruptions of [60] more faithfully. However, this may make it more difficult to satisfy points (ii) and (iii) stated above. As in our work, Lahiri and Qadeer rely on the insight that reachability information can be maintained in first-order logic. They use a collection of manually-specified update formulas that define how their relations are affected by the statements of the language and the (user-inserted) statements that manage the set of head variables.

Chapter 4

Inductive Logic Programming (ILP)

The present chapter discusses inductive logic programming (ILP), a known machine-learning technique that provides a key ingredient for our abstraction-refinement method, which is discussed in Chapter 5. The present chapter starts by defining the problem that is solved by ILP. Sect. 4.1 discusses an existing algorithm that implements the technique. Sect. 4.2 discusses a modification of the algorithm that is more suitable for learning from 3-valued logical structures. Sect. 4.3 presents some extensions that we implemented while adapting the algorithm to be used as part of our abstraction-refinement method. Finally, Sect. 4.4 presents an extension that allows the algorithm to learn nullary formulas. While we do not currently employ the capability of learning nullary formulas in our abstraction-refinement method, this capability is noteworthy because it provides a new technique for predicate abstraction: ILP can be used to identify nullary relations that distinguish a structure S from the other structures arising at a program point. The use of the techniques and algorithms discussed in this chapter for abstraction refinement is described in Sect. 5.2.6.

The goal of an ILP algorithm is the following: given

- an arity $k > 1$,
- a logical structure S,
- a set of positive example assignments of $v_1, \ldots, v_k$ to individuals of S, and
- a set of negative example assignments of $v_1, \ldots, v_k$ to individuals of S,

find a logical formula ψ, with free variables $v_1, \ldots, v_k$, such that (i) ψ is defined in terms of the relations in the vocabulary of S, and (ii) ψ agrees with the classification of input examples, i.e.,

- ψ evaluates to 1 in S on all positive examples, and
- ψ evaluates to 0 in S on all negative examples.

(In this thesis, we describe what is called the *example setting* of ILP; it is the setting employed by the large majority of ILP systems [69]. In this setting, some number of example assignments of the logical formula to be learned have been labeled as positive or negative examples. The semantics of ILP—sometimes referred to as the *model-theory* of ILP—that we assume in this thesis is what is called the *normal semantics*. An alternative semantics of ILP, *non-monotonic semantics*, was introduced by Helft [34] and Flach [27]. ILP systems based on non-monotonic semantics generally learn more conservative properties than do systems based on normal semantics. The reader is referred to the survey by Muggleton and De Raedt for more details on the theory and methods of ILP [69].)

Standard ILP algorithms produce the answer in the form of a logic program (thus the name of the technique). (Non-recursive) logic programs correspond to a subset of first-order logic.[1] A logic program can be thought of as a disjunction over the program rules, with each rule corresponding to a conjunction of literals. Variables not appearing in the head of a rule are implicitly existentially quantified.

Definition 4.1 (ILP) Given (1) a list of variables $v_1, \ldots, v_k$, (2) a set of assignments E^+ of the v_i to individuals (positive examples), (3) a set of assignments E^- of the v_i to individuals (negative examples), and (4) a logical structure S, the goal of ILP is to find a formula $\psi_E(v_1, \ldots, v_k)$ such that all $e \in E^+$ are satisfied (or *covered*) by ψ_E in S and no $e \in E^-$ is satisfied by ψ_E in S. □

Formula ψ_E defines an arity-k relation E.

[1] Some ILP algorithms are capable of producing recursive programs, which correspond to first-order logic plus a least-fixpoint operator (which is more general than transitive closure).

Example 4.2 Consider learning a unary formula (over variable v) that holds for linked-list elements that are pointed to by the n fields of more than one element, i.e., the defining formula of the sharing relation is_n shown in Fig. 2.6. (The importance of the concept of sharing in heap data structures was recognized in [13, 43].) We let $E^+ = \{[v \mapsto u_3], [v \mapsto u_5]\}$, $E^- = \{[v \mapsto u_1], [v \mapsto u_4]\}$, and $S =$ the 2-valued logical structure of Fig. 4.1. The formula $\psi_E(v) \stackrel{\text{def}}{=} \exists\, v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge \neg eq(v_1, v_2)$ meets the objective, as it covers all positive and no negative example assignments. □

4.1 An Implementation of ILP for Learning in 2-Valued Logical Structures

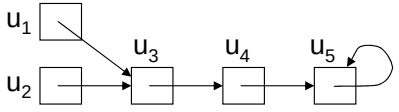


Figure 4.1 A linked list with shared elements

Fig. 4.2 presents the ILP algorithm used by systems such as FOIL [75, 76], modified to construct the answer as a first-order-logic formula in disjunctive normal form. Given the input described in the previous paragraph, this algorithm learns the formula $\psi_E(v) \stackrel{\text{def}}{=} \exists\, v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge \neg eq(v_1, v_2)$ (by performing one iteration of the outer loop and three iterations of the inner loop to successively choose literals $n(v_1, v)$, $n(v_2, v)$, and $\neg eq(v_1, v_2)$). It is a sequential covering algorithm parameterized by the function $Gain$, which characterizes the usefulness of adding a particular literal (generally, in some heuristic fashion). The algorithm creates a new disjunct as long as there are positive examples that are not covered by existing disjuncts. The disjunct is extended by conjoining a new literal until it covers no negative examples. Each literal uses a relation symbol from the vocabulary of structure S; valid arguments to a literal are the variables $v_1, \ldots, v_k$, which are E's formal parameters, as well as additional variables, as long as at least one of the arguments is a variable already used in the current disjunct.[2] In FOIL, a single literal is chosen on each iteration of the inner loop (lines [6]–[13]) using a heuristic value based on the information gain (see line [8]). FOIL uses information gain to find the literal that best distinguishes between positive and negative examples. After the algorithm chooses a literal $Best(v_{i_1}, \ldots, v_{i_m})$,

[2]The latter condition prevents unconstrained searches that cannot yield the simplest formula that agrees with the classification of input examples.

Input: Target-relation name E and the list of variables
$EVars = (v_1, \ldots, v_k)$ to be used as E's formal parameters,
Logical structure $S \in$ 2-STRUCT$[\mathcal{R}]$,
Set of assignments to $EVars$ E^+,
Set of assignments to $EVars$ E^-

[1] $\psi_E := \mathbf{0}$
[2] while $(E^+ \neq \emptyset)$ do
[3] $NewDisjunct := \mathbf{1}$
[4] $NewE^+ := E^+$
[5] $NewE^- := E^-$
[6] while $(NewE^- \neq \emptyset)$ do
[7] $Cand :=$ candidate literals using $\mathcal{R}$
[8] $Best(v_{i_1}, \ldots, v_{i_m}) := L(v_{i_1}, \ldots, v_{i_m}) \in Cand$ with max
$Gain\big(L(v_{i_1}, \ldots, v_{i_m}), NewDisjunct, NewE^+, NewE^-\big)$
[9] $NewVars := \{v_{i_1}, \ldots, v_{i_m}\} \setminus variables(NewDisjunct)$
[10] Extend $NewE^+$ and $NewE^-$ with assignments
for $NewVars$ that satisfy $Best(v_{i_1}, \ldots, v_{i_m})$
[11] $NewE^- :=$ subset of $NewE^-$ satisfying $Best(v_{i_1}, \ldots, v_{i_m})$
[12] $NewDisjunct := NewDisjunct \wedge Best(v_{i_1}, \ldots, v_{i_m})$
[13] end
[14] $E^+ :=$ subset of E^+ *not* satisfying $NewDisjunct$
[15] $\exists$-quantify each $v \in variables(NewDisjunct) \setminus \{v_1, \ldots, v_k\}$
[16] $\psi_E := \psi_E \vee NewDisjunct$
[17] end

Figure 4.2 Pseudo-code for FOIL

the current sets of example assignments, $NewE^+$ and $NewE^-$, need to be extended with mappings for each variable $v_j \in \{v_{i_1}, \ldots, v_{i_m}\}$ that does not already occur in the current disjunct (line [10]). The updated sets $NewE^+$ and $NewE^-$ should consist of assignments that are still to be handled, i.e., still to be covered in the case of $NewE^+$ and still to be excluded in the case of $NewE^-$. Each assignment $a \in NewE^+$ (or $NewE^-$) is extended (potentially to multiple assignments) with mappings for new variables from each assignment b that satisfies $Best(v_{i_1}, \ldots, v_{i_m})$ and agrees with a in its mapping of variables that are common to a and b. Using relational-algebra terminology, the updated assignment set $NewE^+$ ($NewE^-$) is the natural join of the previous value of $NewE^+$ ($NewE^-$) and the relation defined by the literal $Best(v_{i_1}, \ldots, v_{i_m})$.

4.2 An Implementation of ILP for Learning in 3-Valued Logical Structures

In the ILP literature, the logical structure S that serves as input to the algorithm is generally a *partial* 2-valued structure: for a given relation $r \in \mathcal{R}_l$ and a complete assignment a for $r(v_1, \ldots, v_l)$, the value $[\![r(v_1, \ldots, v_l)]\!]_2^S(a)$ may be 0, 1, or unspecified. The structure S can be extended to a structure in 2-STRUCT$[\mathcal{R}]$ (in many ways) by filling in unspecified entries with 0 or 1 values. (In general, some of these structures represent inadmissible stores.)

Given a partial logical structure S, a formula ψ_E learned by the algorithm shown in Fig. 4.2 is guaranteed to be satisfied by any assignment $e^+ \in E^+$ (i.e., to evaluate to 1 on e^+) in a structure $S_M \in$ 2-STRUCT$[\mathcal{R}]$ (defined below) and not to be satisfied by any assignment $e^- \in E^-$ (i.e., to evaluate to 0 on e^-) in S_M [69]. S_M is the logical structure that corresponds to *a minimal Herbrand model* of the values of background relations in S, the classification of input examples, and ψ_E. In other words, S_M is an extension of S such that no 1 value in S_M can be changed to 0 while maintaining S_M as an extension of S that satisfies ψ_E with the input classification of examples.[3] The guarantee with respect to other extensions $S' \in$ 2-STRUCT$[\mathcal{R}]$ of S is weaker. The formula ψ_E is guaranteed to be satisfied by any assignment $e^+ \in E^+$ in S' but it may or may not be satisfied by an assignment $e^- \in E^-$ in S'.

[3] Because the algorithm shown in Fig. 4.2 may introduce negated literals, there may not be a unique such extension.

The above statement is corroborated by the code on lines [10] and [11] of the algorithm. Because in code on those lines the algorithm continues to process only those negative examples (or their extensions) that satisfy $Best(v_{i_1}, \ldots, v_{i_m})$ in S, the algorithm makes a kind of *closed-world assumption*: literal values not known to be true are assumed to be false.

For learning from a structure $S^{\#} \in$ 3-STRUCT$[\mathcal{R}]$, it seems natural to define a partial logical structure S that consists of the definite entries of $S^{\#}$ but in which the entries corresponding to the $1/2$ relation values of $S^{\#}$ are unspecified. With S as input, the algorithm shown in Fig. 4.2 returns a formula ψ_E that is guaranteed to be satisfied by any assignment $e^+ \in E^+$ (i.e., to evaluate to 1 on e^+) in *all* concrete structures $S^{\natural} \sqsubseteq S^{\#}$, and for each assignment $e^- \in E^-$ not to be satisfied (i.e., to evaluate to 0 on e^-) in *some* $S^{\natural} \sqsubseteq S^{\#}$. The values of ψ_E in $S^{\#}$ are as follows: ψ_E is guaranteed to be (definitely) satisfied by any assignment $e^+ \in E^+$ (i.e., to evaluate to 1 on e^+) in $S^{\#}$ and it is guaranteed to be (possibly) not satisfied by any assignment $e^- \in E^-$ (i.e., to evaluate to 0 or $1/2$ on e^-) in $S^{\#}$.

However, in our intended setting of program analysis (or, more generally, transition systems that define the evolution of a logical structure), it is important that the learned formulas come with guarantees for *all* $S^{\natural} \sqsubseteq S^{\#}$, namely, that ψ_E yield 0 on all negative examples in all $S^{\natural} \sqsubseteq S^{\#}$ (and in $S^{\#}$ itself). To achieve this, we change the operations shown on lines [10] and [11] of Fig. 4.2 to process all *negative* examples that *potentially satisfy* $Best(v_{i_1}, \ldots, v_{i_m})$, rather than only those that *satisfy* $Best(v_{i_1}, \ldots, v_{i_m})$. The change ensures that the algorithm retains the (extended) negative-example assignments that have not been (definitely) excluded (i.e., for which $NewDisjunct$ does not evaluate to 0). Fig. 4.3 shows the ILP algorithm of Fig. 4.2 with the above modifications appearing in bold. Henceforth, we assume that learning from 3-valued logical structures is performed via the algorithm shown in Fig. 4.3. (The differences between the algorithms shown in Figs. 4.2 and 4.3 have no effect on learning from (non-partial) 2-valued logical structures.)

```
Input:  Target-relation name E and the list of variables
        EVars = (v_1, ..., v_k) to be used as E's formal parameters,
        Logical structure S ∈ 3-STRUCT[R],
        Set of assignments to EVars E+,
        Set of assignments to EVars E−
[1]   ψ_E := 0
[2]   while (E+ ≠ ∅) do
[3]     NewDisjunct := 1
[4]     NewE+ := E+
[5]     NewE− := E−
[6]     while (NewE− ≠ ∅) do
[7]       Cand := candidate literals using R
[8]       Best(v_{i_1}, ..., v_{i_m}) := L(v_{i_1}, ..., v_{i_m}) ∈ Cand with max
            Gain(L(v_{i_1}, ..., v_{i_m}), NewDisjunct, NewE+, NewE−)
[9]       NewVars := {v_{i_1}, ..., v_{i_m}} \ variables(NewDisjunct)
[10]      Extend NewE+ with assignments for NewVars
            that satisfy Best(v_{i_1}, ..., v_{i_m})
[11]      Extend NewE− with assignments for NewVars
            that potentially satisfy Best(v_{i_1}, ..., v_{i_m})
[12]      NewE− := subset of NewE−
            potentially satisfying Best(v_{i_1}, ..., v_{i_m})
[13]      NewDisjunct := NewDisjunct ∧ Best(v_{i_1}, ..., v_{i_m})
[14]    end
[15]    E+ := subset of E+ not satisfying NewDisjunct
[16]    ∃-quantify each v ∈ variables(NewDisjunct) \ {v_1, ..., v_k}
[17]    ψ_E := ψ_E ∨ NewDisjunct
[18]  end
```

Figure 4.3 Pseudo-code for FOIL modified to learn from 3-valued logical structures

4.3 Extensions of the Algorithm of Fig. 4.3 for Abstraction Refinement

The context in which we use ILP necessitates two modifications to the basic algorithm of Fig. 4.3. First, we changed the algorithm to learn multiple formulas in one invocation. Our motivation is not to find a single instrumentation relation that explains something about the input logical structure, but rather to find all instrumentation relations that help the analysis establish the property of interest. Whenever we find multiple literals whose quality metric (see line [8] of Fig. 4.3) lies within a certain threshold of the highest-quality metric observed so far, we extend distinct copies of the current disjunct using each of the literals, and then we extend distinct copies of the current formula using the resulting disjuncts. This extension is similar to the concept of a *beam search*, e.g., as implemented in the CN2 system [15]. However, while a beam search returns only the n answers with the highest quality metric (for some fixed n), our extension puts no bound on the number of answers. As mentioned above, this is in line with our motivation for finding all instrumentation relations that may help the analysis establish the property of interest. To cope with the potentially large number of answers returned by the algorithm, our abstraction-refinement method includes heuristics for pruning the set of answers returned by ILP.

As explained in Sect. 5.2.6.2, we sometimes invoke ILP with only positive or only negative examples. The second change is needed to enable the algorithm of Fig. 4.3 to return meaningful answers in the absence of negative or positive examples. By changing the inner loop (lines [6]–[14] of Fig. 4.3) from a `while` loop to a `do-while`, we obtain non-trivial formulas in the absence of negative examples. Similarly, by changing the outer loop (lines [2]–[18] of Fig. 4.3) from a `while` loop to a `do-while`, we obtain non-trivial formulas in the absence of positive examples. When the algorithm is invoked with the empty set of negative (or positive) examples, in place of FOIL's information-gain heuristic (see line [8] of Fig. 4.3) we use a simpler heuristic based on the percentage of positive examples covered (or negative examples excluded) by the new disjunct.

4.4 An Extension of the Algorithm of Fig. 4.3 for Learning Nullary Relations

The algorithm shown in Fig. 4.3 expects the arity of the target relation E (as well as the arities of the example assignments in sets E^+ and E^-) to be greater than zero. The algorithm learns relation(s) of the requested arity that satisfy the given classification of examples in the unique input structure.

In program-analysis applications, e.g., when the abstraction is based on predicate-abstraction domains, which use only nullary relations to express properties of memory configurations, it is desirable to find nullary relations that distinguish between two memory configurations that can arise at a given program point.

A conceptually simple change to the algorithm shown in Fig. 4.3 enables it to learn a nullary relation that evaluates to 1 on one set of logical structures and to 0 on another set of logical structures. We lift all operations that expect a set of assignments of individuals to variables to apply to a set of pairs of the form (S_i, T_i), where S_i is a logical structure and T_i is a set of assignments of individuals to variables. (We also change the signature of the algorithm; we will state the new signature shortly.) For instance, when finding the subset of the remaining negative examples that satisfy $Best(v_{i_1}, \ldots, v_{i_m})$ (see line [12] of Fig. 4.3), the computation

$$\{t \in NewE^- \mid [\![Best(v_{i_1}, \ldots, v_{i_m})]\!]_3^S(t) = 1\}$$

is replaced with

$$\big\{(S_i, T) \mid (S_i, T_i) \in NewE^- \text{ and } T = \{t \in T_i \mid [\![Best(v_{i_1}, \ldots, v_{i_m})]\!]_3^{S_i}(t) = 1\}\big\}$$

The new signature of the algorithm consists of the target-relation name E, the list of variables $(v_1, \ldots, v_k)$ to be used as E's formal parameters, a positive-example set of the form $\{(S_1^+, T_1^+), \ldots, (S_p^+, T_p^+)\}$, and a negative-example set of the form $\{(S_1^-, T_1^-), \ldots, (S_n^-, T_n^-)\}$. The S_i^+ and S_i^- entries are structures, and the T_i^+ and T_i^- entries are sets of arity-k assignments.

Now, when attempting to learn a logical relation that evaluates to 1 on a set of structures $\{S_1^+, \ldots, S_p^+\}$ and to 0 on a set of structures $\{S_1^-, \ldots, S_n^-\}$, we invoke the modified algorithm

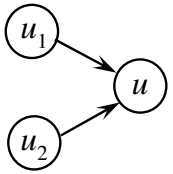


Figure 4.4 A logical structure $S_{4.4}$ in which u is shared

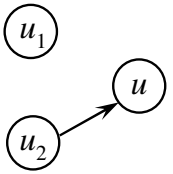


Figure 4.5 A logical structure $S_{4.5}$ in which no element is shared

with the following arguments: the target-relation name E, the empty variable list, the positive-example set $\{(S_1^+, \emptyset), \ldots, (S_p^+, \emptyset)\}$, and the negative-example set $\{(S_1^-, \emptyset), \ldots, (S_n^-, \emptyset)\}$. For instance, if we invoke the modified algorithm with arguments $E()$ (with the empty variable list), $E^+ = \{(S_{4.4}, \emptyset)\}$, and $E^- = \{(S_{4.5}, \emptyset)\}$, where $S_{4.4}$ and $S_{4.5}$ are as shown in Figs. 4.4 and 4.5, respectively, the algorithm identifies the presence or the absence of sharing to be the key distinction between the structures; it returns the formula

$$\psi_E() \stackrel{\text{def}}{=} \exists\, v_1, v_2, v_3 : n(v_1, v_2) \wedge n(v_3, v_2) \wedge \neg eq(v_1, v_3).$$

When attempting to learn a k-ary logical relation (for $k > 0$) that evaluates to 1 on the set of positive-example assignments E^+ and to 0 on the set of negative-example assignments E^- in structure S, we invoke the modified algorithm with the following arguments: the target-relation name E, the variables list $(v_1, \ldots, v_k)$, the single-element positive-example set $\{(S, E^+)\}$, and the single-element negative-example set $\{(S, E^-)\}$.

Given an input of the general form, i.e., the target-relation name E, the list of variables $(v_1, \ldots, v_k)$ to be used as E's formal parameters, the positive-example set $\{(S_1^+, T_1^+), \ldots, (S_p^+, T_p^+)\}$, and the negative-example set $\{(S_1^-, T_1^-), \ldots, (S_n^-, T_n^-)\}$, the modified algorithm learns a k-ary relation that evaluates to 1 in S_i^+ on all assignments in T_i^+ (for $i \in [1..p]$) and to 0 in S_j^- on all assignments in T_j^- (for $j \in [1..n]$).

Chapter 5

Automatic Abstraction Refinement

The present chapter addresses an instance of the following fundamental challenge in applying abstract interpretation:

> Given a program and a query of interest, how does one create an abstraction that is sufficiently precise to verify that the program satisfies the query?

The chapter presents an approach to creating abstractions automatically that, as in some previous work, involves the successive refinement of the abstraction in use. Unlike previous work, the work presented in this chapter is aimed at the analysis of programs that manipulate pointers and heap-allocated data structures (i.e., shape analysis). However, while we demonstrate our approach on shape-analysis problems, the approach is applicable in any program-analysis setting that uses first-order logic.

Refinement is performed by introducing new instrumentation relations (defined via logical formulas over core relations). Our abstraction-refinement method uses two refinement strategies. The first strategy, *subformula-based refinement*, analyzes the sources of imprecision in the evaluation of the query, and chooses how to define new instrumentation relations using subformulas of the query. The second strategy, *ILP-based refinement*, employs inductive logic programming (ILP) to learn new instrumentation relations that can stave off imprecision due to abstraction.

As will be explained in Sect. 5.2.6.2, ILP-based refinement invokes the ILP algorithm described in Sect. 4.3 to learn formulas for three kinds of relations that can be used to refine abstractions in our analysis framework, which uses abstractions that generalize predicate-abstraction domains. A fourth use of ILP provides a new technique for predicate abstraction itself: ILP can be used to

identify nullary relations that distinguish a positive-example structure S from the other structures arising at a program point. Sect. 4.4 described a version of the ILP algorithm that provides this capability.

The steps of ILP go beyond merely forming Boolean combinations of existing relations (as in many refinement techniques based on predicate abstraction); ILP can create new relations by introducing quantifiers during the learning process.

The chapter is organized as follows: Sect. 5.1 illustrates our goals on the problem of verifying the partial correctness of a sorting routine. Sect. 5.2 presents our abstraction-refinement method. (Sect. 5.2.2 describes subformula-based refinement. Sect. 5.2.6 discusses a shortcoming of subformula-based refinement and describes ILP-based refinement, which uses ILP for learning an abstraction.) Sects. 5.3 and 5.4 present experimental results. Sect. 5.5 discusses related work.

5.1 Example: Specifying and Verifying Sortedness

Given the static-analysis algorithm defined in Sect. 2.2, to demonstrate the partial correctness of a procedure, the user must supply the following program-specific information:

- The procedure's control-flow graph.
- A *data-structure constructor* (DSC): a code fragment that non-deterministically constructs all valid inputs.
- A query; i.e., a formula that identifies the intended outputs.

The analysis algorithm is run on the DSC concatenated with the procedure's control-flow graph; the query is then evaluated on the structures that are generated at exit.

Consider the problem of establishing that the version of `InsertSort` shown in Fig. 5.1 is partially correct. Fig. 5.2 shows the three structures that characterize the valid inputs to `InsertSort` (they represent the set of stores in which program variable `x` points to an acyclic linked list). To verify that `InsertSort` produces a *sorted* permutation of the input list, after running the analysis of `InsertSort`, we would check to see whether, for all of the structures that arise at the procedure's

```
[1]  void InsertSort(List *x) {
[2]    List *r, *pr, *rn, *l, *pl;
[3]    r = x;
[4]    pr = NULL;
[5]    while (r != NULL) {
[6]      l = x;
[7]      rn = r->n;
[8]      pl = NULL;
[9]      while (l != r) {
[10]       if (l->data > r->data) {
[11]         pr->n = rn;
[12]         r->n = l;
[13]         if (pl == NULL) x = r;
[14]         else pl->n = r;
[15]         r = pr;
[16]         break;
[17]       }
[18]       pl = l;
[19]       l = l->n;
[20]     }
[21]     pr = r;
[22]     r = rn;
[23]   }
[24] }
```

Figure 5.1 A stable version of insertion sort

exit node, the following formula evaluates to 1:

$$\forall v_1 \colon r_{n,x}(v_1) \Rightarrow (\forall v_2 \colon n(v_1, v_2) \Rightarrow dle(v_1, v_2)). \tag{5.1}$$

If the formula evaluates to 1, then the nodes reachable from `x` must be in non-decreasing order.[1]

Abstract interpretation collects 3-valued structure $S_{2.8}$ shown in Fig. 2.8 at line [24]. Note that Formula (5.1) evaluates to $1/2$ on $S_{2.8}$. While the first list element is guaranteed to be in correct order with respect to the remaining elements—note the definite dle edge between the first node and the summary node—there is no guarantee that all list nodes represented by the summary node are in correct order. In particular, because $S_{2.8}$ represents $S_{2.7}$, shown in Fig. 2.7, the analysis admits the possibility that the (correct) implementation of insertion sort of Fig. 5.1 can produce the store shown in Fig. 2.2. Thus, the abstraction that we used was not fine-grained enough to establish the partial correctness of `InsertSort`. In fact, the abstraction is not fine-grained enough to separate the set of sorted lists from the set of lists not in sorted order.

In [53], Lev-Ami et al. used TVLA to establish the partial correctness of `InsertSort`. The key step was the introduction of instrumentation relation $inOrder_{dle,n}(v)$, which holds for nodes whose `data`-components are less than or equal to those of their `n`-successors; $inOrder_{dle,n}(v)$ was defined by:

$$inOrder_{dle,n}(v) \stackrel{\text{def}}{=} \forall v_1 \colon n(v, v_1) \Rightarrow dle(v, v_1). \tag{5.2}$$

The sortedness property was then stated as follows (cf. Formula (5.1)):

$$\forall v \colon r_{n,x}(v) \Rightarrow inOrder_{dle,n}(v). \tag{5.3}$$

After the introduction of relation $inOrder_{dle,n}$, the 3-valued structures that are collected by abstract interpretation at the end of `InsertSort` describe all stores in which variable `x` points to an acyclic, *sorted* linked list. In all of these structures, Formulas (5.3) and (2.1) evaluate to 1. Consequently, `InsertSort` is guaranteed to work correctly on all valid inputs.

[1] A second property required of a correct sorting procedure (as well as of many other procedures that manipulate linked lists) is that the output list must be a permutation of the input list. This can be established by also checking Formula (2.1) from Chapter 2.

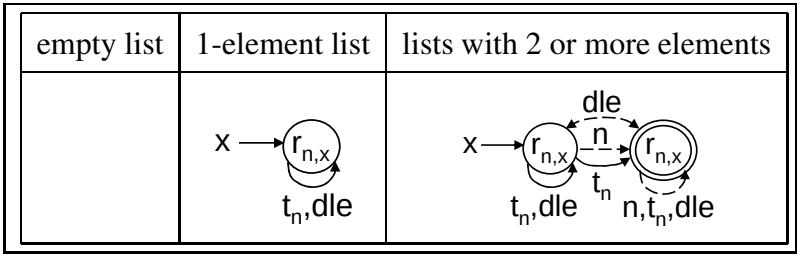


Figure 5.2 The structures that describe possible inputs to `InsertSort`

5.2 Iterative Abstraction Refinement

In [53], instrumentation relation $inOrder_{dle,n}$ was defined explicitly (by the TVLA user). Heretofore, there have really been two burdens placed on the TVLA user:

(i) he must have insight into the behavior of the program, and

(ii) he must translate this insight into appropriate instrumentation relations (e.g., Formula (5.2)).

The goal of the present chapter is to automate the identification of appropriate instrumentation relations, such as $inOrder_{dle,n}$. In the case of `InsertSort`, the goal is to obtain definite answers when evaluating Formula (5.1) on the structures collected by abstract interpretation at line [24] of Fig. 5.1. Fig. 5.3 gives pseudo-code for our method, the steps of which can be explained as follows:

- (Line [1]; Sect. 5.2.4) Use a *data-structure constructor* to compute the abstract input structures that represent all valid inputs to the program.
- Perform an abstract interpretation to collect a set of structures at each program point, and evaluate the query on the structures at exit. If a definite answer is obtained on all structures, terminate. Otherwise, perform abstraction refinement.
- (Line [6]; Sects. 5.2.2 and 5.2.6) Identify formulas to be used to define new instrumentation relations.

```
Input:  the program's transition relation,
        a data-structure constructor,
        a query φ (a closed formula)
[1]  Construct abstract input
[2]  do
[3]    Perform abstract interpretation
[4]    Let S_1, ..., S_k be the set of
         3-valued structures at exit
[5]    if for all S_i, [[φ]]_3^{S_i}([]) ≠ 1/2 break
[6]    Find formulas ψ_{p_1}, ..., ψ_{p_k} for new
         instrumentation relations p_1, ..., p_k
[7]    Refine the actions that define
         the program's transition relation
[8]    Refine the abstract input
[9]  while (true)
```

Figure 5.3 Pseudo-code for iterative abstraction refinement

- (Line [7]; Sect. 5.2.3) Replace all occurrences of these formulas in the query and in the definitions of other instrumentation relations with the use of the corresponding new instrumentation relation symbols, and apply finite differencing to generate relation-maintenance formulas for the newly introduced instrumentation relations, as well as for those instrumentation relations whose definitions have been changed.

- (Line [8]; Sect. 5.2.4) Obtain the most precise possible values for the newly introduced instrumentation relations in abstract structures that define the valid inputs to the program. This is achieved by "reconstructing" the valid inputs by performing abstract interpretation of the data-structure constructor.

In the next five sections, we will use the example of verifying partial correctness of `InsertSort` to illustrate the steps of iterative abstraction refinement.

5.2.1 Instrumentation-Relation Discovery

A first attempt at abstraction refinement could be the introduction of the query itself as a new instrumentation relation. However, this usually does not lead to a definite answer to the query. For instance, with `InsertSort`, introducing the query as a new instrumentation relation is ineffective because no statement of the program has the effect of changing the value of such an instrumentation relation from $1/2$ to 1.

In contrast, as we saw in Sect. 5.1, the introduction of unary instrumentation relation $inOrder_{dle,n}$ allows the sortedness query to be established. When $inOrder_{dle,n}$ is present, there are several statements of the program where abstract interpretation results in new definite entries for $inOrder_{dle,n}$. For instance, because of the comparison in line [10] of Fig. 5.1, the insertion in lines [12]–[14] of the node pointed to by `r` (say u) before the node pointed to by `l` results in a new definite entry $inOrder_{dle,n}(u)$.

An algorithm to generate new instrumentation relations should take into account the sources of imprecision. Sect. 5.2.2 describes subformula-based refinement; in this method, query subformulas that are responsible for an indefinite answer are used to generate new instrumentation relations.

Sect. 5.2.6 discusses a shortcoming of subformula-based refinement and describes ILP-based refinement, which uses ILP for learning an abstraction.

5.2.2 Subformula-Based Refinement

The subformulas of the query that are responsible for the indefinite answer are good candidates for defining new instrumentation relations. Fig. 5.4 presents function *instrum*, a recursive-descent procedure to generate defining formulas for new instrumentation relations. The arguments to the function are formula φ, logical structure $S \in$ 3-STRUCT$[\mathcal{R}]$, and an assignment Z that is defined on all free variables of φ. In the top-level invocation, φ is the (nullary) query, Z is empty, and S is a structure collected at the exit node by the last run of abstract interpretation for which $[\![\varphi]\!]_3^S(Z) = 1/2$.

A precondition of *instrum* is that $[\![\varphi]\!]_3^S(Z) = 1/2$. Starting with this assumption, *instrum* attempts to find subformulas of φ that, if sharpened, would sharpen the value of the whole formula (see Figs. 5.5 (a) and (b)). If such subformulas are found, they will be used to define new instrumentation relations. Below are explanations of a few cases:

instrum$(\mathbf{1}, \ldots)$ This violates the precondition of *instrum*.

instrum$(\mathbf{1/2}, \ldots)$ Nothing can be done in this case.

instrum$(p \in \mathcal{C}, \ldots)$ If p is unary and is not in the set of abstraction relations, add it to the set of abstraction relations.

instrum$(p \in \mathcal{I}, \ldots)$ Examine ψ_p, the defining formula of p. Also, if p is unary and is not in the set of abstraction relations, add it to the set of abstraction relations.

instrum$(\varphi_1 \vee \varphi_2, \ldots)$ If φ (i.e., $\varphi_1 \vee \varphi_2$) does not define an instrumentation relation, it will be used as the definition of a new instrumentation relation. Also, examine φ_1 and φ_2 to find subformulas that can cause φ to evaluate to $1/2$.

φ	return value of ***instrum***(φ, S, Z)
$\mathbf{0}, \mathbf{1}$	*ERROR*
$\mathbf{1/2}$	$\emptyset$
$v_1 = v_2$	$\emptyset$
$p(v_1, \ldots, v_k)$	$(p \in \mathcal{C})\ ?\ \emptyset : $ ***instrum***(ψ_p, S, Z) if $(k = 1 \wedge p \notin \mathcal{A})\ \mathcal{A} := \mathcal{A} \cup \{p\}$
$\neg\varphi_1$	***instrum***(φ_1, S, Z)
$\varphi_1 \vee \varphi_2$ $\varphi_1 \wedge \varphi_2$	$(\varphi \in \{\psi_p \mid p \in \mathcal{I}\})\ ?\ \emptyset : \{\varphi\}$ $\cup\ ([\![\varphi_1]\!]_3^S(Z) = 1/2)\ ?$ ***instrum***$(\varphi_1, S, Z) : \emptyset$ $\cup\ ([\![\varphi_2]\!]_3^S(Z) = 1/2)\ ?$ ***instrum***$(\varphi_2, S, Z) : \emptyset$
$\exists v\colon \varphi_1$ $\forall v\colon \varphi_1$	$(\varphi \in \{\psi_p \mid p \in \mathcal{I}\})\ ?\ \emptyset : \{\varphi\}$ $\cup \bigcup_{u \in S} ([\![\varphi_1]\!]_3^S(Z[v \mapsto u]) = 1/2$ $?$ ***instrum***$(\varphi_1, S, Z[v \mapsto u])$ $: \emptyset)$
$p^*(v_1, \ldots, v_k)$	$(\varphi \in \{\psi_q \mid q \in \mathcal{I}\})\ ?\ \emptyset : \{\varphi\}$ $\cup \bigcup ([\![p]\!]_3^S(Z[v_1' \mapsto u_1', v_2' \mapsto u_2']) = 1/2)$ $u_1', u_2' \in S,\ ?$ ***instrum***$(p, S, Z[v_1' \mapsto u_1', v_2' \mapsto u_2'])$ $u_1' \neq u_2'\ : \emptyset)$

Figure 5.4 Function *instrum*, which looks for formulas to be used as definitions of new instrumentation relations

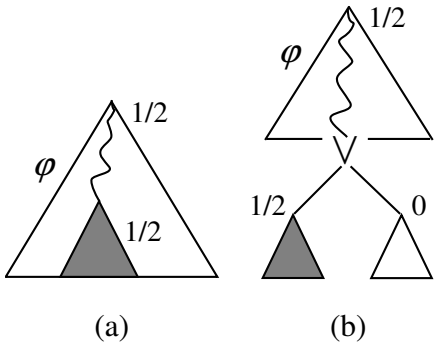


Figure 5.5 (a) Recursive-descent function *instrum* finds the subformulas of φ that can cause the $1/2$ answer. (b) Ex.: imprecision in an or-subformula.

instrum$(\exists\, v\colon \varphi_1, S, Z)$ If φ (i.e., $\exists\, v\colon \varphi_1$) does not define an instrumentation relation, it will be used as the definition of a new instrumentation relation. Also, examine φ_1 under different bindings $v \mapsto u$ to find subformulas of φ_1 that can cause φ to evaluate to $1/2$.

Each formula φ returned by *instrum* is given a name (say q) and used as the definition of a new instrumentation relation $q(v_1, \ldots, v_k)$, where $v_1, \ldots, v_k$ are the free variables of φ (in order of their appearance in the formula). All new unary instrumentation relations are added as non-abstraction relations. However, they may be added to the set of abstraction relations $\mathcal{A}$ on a subsequent iteration of abstraction refinement (see the second line of entry $p(v_1, \ldots, v_k)$ in Fig. 5.4, which handles core and instrumentation relations).

Example 5.1 Before the first abstract interpretation of `InsertSort`, the data-structure constructor of Fig. 5.7 (see Sect. 5.2.4) constructs all valid inputs to the procedure, which are the three structures shown in Fig. 5.2. As we saw in Sect. 5.1, abstract interpretation collects 3-valued structure $S_{2.8}$ of Fig. 2.8 at the exit node of `InsertSort`.[2] The sortedness query (Formula (5.1)) evaluates to $1/2$ on $S_{2.8}$, triggering a call to *instrum* with Formula (5.1), structure $S_{2.8}$, and empty assignment Z, as arguments.

Column 2 of Fig. 5.6 shows the instrumentation relations that are created as a result of the call to *instrum* on the first iteration of abstraction refinement. Note that $sorted_3$ is defined exactly as

[2]In our implementation, a given round of abstract interpretation is stopped as soon as imprecision is detected.

p	ψ_p (after call to *instrum*)	ψ_p (final version)
$sorted_1()$	$\forall\, v_1 : r_{n,x}(v_1) \Rightarrow (\forall\, v_2 : n(v_1, v_2) \Rightarrow dle(v_1, v_2))$	$\forall\, v_1 : sorted_2(v_1)$
$sorted_2(v_1)$	$r_{n,x}(v_1) \Rightarrow (\forall\, v_2 : n(v_1, v_2) \Rightarrow dle(v_1, v_2))$	$r_{n,x}(v_1) \Rightarrow sorted_3(v_1)$
$sorted_3(v_1)$	$\forall\, v_2 : n(v_1, v_2) \Rightarrow dle(v_1, v_2)$	$\forall\, v_2 : sorted_4(v_1, v_2)$
$sorted_4(v_1, v_2)$	$n(v_1, v_2) \Rightarrow dle(v_1, v_2)$	$n(v_1, v_2) \Rightarrow dle(v_1, v_2)$

Figure 5.6 Instrumentation relations created by subformula-based refinement during the verification of the partial correctness of `InsertSort`

$inOrder_{dle,n}$, which was the key insight for the results of [53]. Note also that *instrum* returns no subformulas of the definition of $r_{n,x}$. This is because $r_{n,x}(v)$ evaluates to a definite value (1) for both $v \mapsto u_{23}$ and $v \mapsto u_1$ (see Fig. 2.8). □

5.2.3 Refinement of the Actions that Define the Program's Transition Relation

The actions that define the program's transition relation need to be modified to gain precision improvements from storing and maintaining the new instrumentation relations. To this end, for each new instrumentation relation $p(v_1, \ldots, v_k)$, the query and all other instrumentation relations' defining formulas are scanned for occurrences of ψ_p. Every occurrence of $\psi_p\{w_1/v_1, \ldots, w_k/v_k\}$, i.e., ψ_p with w_i substituted for free variable v_i, is replaced with $p(w_1, \ldots, w_k)$, thus enabling the use of stored value $p(w_1, \ldots, w_k)$ in place of the evaluation of ψ_p.

To complete transition-relation refinement, finite differencing creates relation-maintenance formulas for the new instrumentation relations, as well as for those instrumentation relations whose definitions have been changed. This improves the precision with which relations' stored values are maintained during abstract interpretation.

Example 5.2 For `InsertSort`, the use of Formula (5.1) in the query is replaced with the use of the stored value $sorted_1()$. Then the definitions of all instrumentation relations are scanned for occurrences of $\psi_{sorted_1}, \ldots, \psi_{sorted_4}$ (in that order). These occurrences are replaced with the names of the four relations. In this case, only the new relations' definitions are changed, yielding the definitions given in Column 3 of Fig. 5.6. Transition-relation refinement is completed by invoking

finite differencing to generate relation-maintenance formulas for the new instrumentation relations. □

5.2.4 Refinement of the Abstract Input: Data-Structure Constructors

Before performing abstract interpretation of the refined transition system, we need to update the abstract structures that characterize the acceptable inputs to the procedure with values for the new instrumentation relations. To gain maximum benefit from maintaining $p(v_1, \ldots, v_k)$, abstract interpretation needs to start with the most precise possible values for p in abstract input structures. While simply evaluating ψ_p on abstract input structures for all assignments to free variables $v_1, \ldots, v_k$ results in safe values, these values are likely to be imprecise.

We illustrate the issue on the stability property. This property usually arises in the context of sorting procedures, but actually applies to list-manipulating programs in general: the stability query (Formula (5.5)) asserts that the relative order of elements with equal `data`-components remains the same.[3]

$$\forall\, v_1, v_2 \colon (dle(v_1, v_2) \wedge dle(v_2, v_1) \wedge t_n^0(v_1, v_2)) \Rightarrow t_n(v_1, v_2) \tag{5.5}$$

The first run of abstract interpretation on `InsertSort` does not result in a definite answer to the stability query. The first round of abstraction refinement then introduces the following subformula of Formula (5.5) as a new instrumentation relation, $stable_2(v_1, v_2)$:

$$(dle(v_1, v_2) \wedge dle(v_2, v_1) \wedge t_n^0(v_1, v_2)) \Rightarrow t_n(v_1, v_2) \tag{5.6}$$

Consider the rightmost structure of Fig. 5.2,[4] which includes one concrete and one summary individual; call them u_c and u_s, respectively. If we simply evaluate Formula (5.6) on the structure, we

[3] A related property, antistability, asserts that the order of elements with equal `data`-components is reversed:

$$\forall\, v_1, v_2 \colon (dle(v_1, v_2) \wedge dle(v_2, v_1) \wedge t_n^0(v_1, v_2)) \Rightarrow t_n(v_2, v_1) \tag{5.4}$$

Our test suite also includes program `InsertSort_AS`, which is identical to `InsertSort` except that it uses $\geq$ instead of $>$ in line [10] of Fig. 5.1 (i.e., when looking for the correct place to insert the current node). This implementation of insertion sort is antistable.

[4] In that structure, all history relations, such as t_n^0, have the same values as their active counterparts, but have been omitted from the figure for clarity.

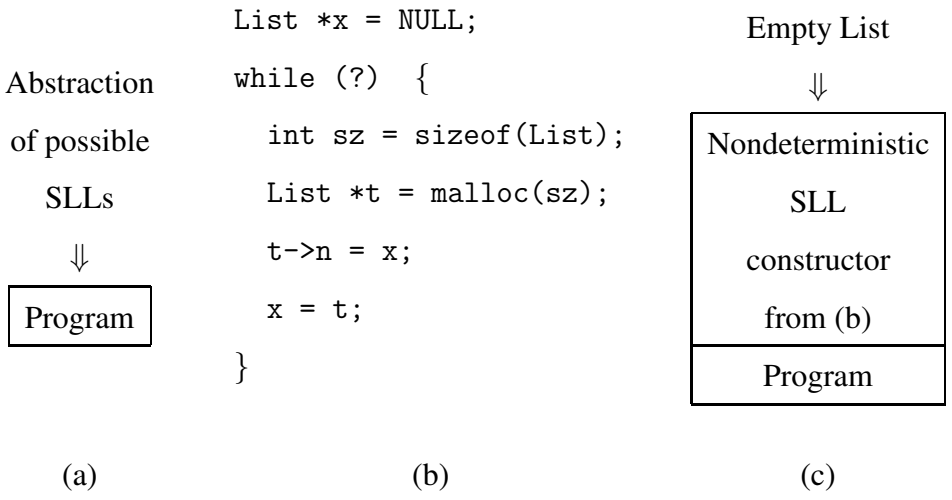


Figure 5.7 Illustration of input specifications for programs that manipulate singly-linked lists. (a) Traditional input specification in TVLA. (b) A fragment of code that nondeterministically constructs all possible singly-linked lists. (c) The use of loop (b) to specify the set of inputs.

obtain the definite value 1 for tuples (u_c, u_c), (u_c, u_s), and (u_s, u_c). However, the evaluation yields value $1/2$ for tuple (u_s, u_s) because $dle(u_s, u_s)$, $t_n^0(u_s, u_s)$, and $t_n(u_s, u_s)$ all equal $1/2$.

Our methodology for obtaining values for abstract input structures is to perform an abstract interpretation on a loop that constructs the family of all valid inputs to the program (we call such a loop a **D**ata-**S**tructure **C**onstructor, or DSC). This allows the values of instrumentation relations to be maintained (as input structures are manufactured from the empty store) rather than computed; in general, this results in more precise values for the instrumentation relations. Fig. 5.7 illustrates the idea. The left-hand side shows the traditional TVLA approach, in which the program is analyzed together with a specification of valid inputs. The loop in the middle nondeterministically constructs an acyclic linked list pointed to by `x`: a list is constructed from tail to head (i.e., most deeply nested node first); the loop exits after some number of nodes have been added at the front of the list. An example of our methodology is depicted on the right-hand side. The program to be analyzed is now composed of a DSC (here the nondeterministic loop constructing all singly-linked lists) together with the original procedure. The input specification now consists of just the empty list.

The abstract interpretation of the DSC is performed using an extended vocabulary that contains the new instrumentation relation symbols. The 3-valued structures collected at the exit node of the DSC become the abstract input to the original procedure, i.e., the abstract value at the procedure's entry point, for the subsequent abstract interpretation of the procedure.

Note that history relations (such as $r^0_{n,x}(v)$ from Chapter 2) are intended to record the state of the store at the entry point to the procedure or, equivalently, at the exit from the DSC. To make sure that these relations have appropriate values, they are maintained in tandem with their active counterparts during abstract interpretation of the DSC. When abstract input refinement is completed, values of history relations are frozen in preparation for the abstract interpretation that is about to be performed on the procedure proper.

The $stable_2$ instrumentation relation defined by Formula (5.6) exemplifies the benefits of the DSC methodology. The maintenance of $stable_2$, t_n, t^0_n, and other instrumentation relations, starting from the empty store, allows us to conclude that $stable_2$ has value 1 for every tuple of every abstract input structure to procedure `InsertSort` (and so the stability property holds initially).

Example 5.3 Fig. 5.7 shows the linked-list DSC used for specifying the inputs to `InsertSort`. Abstract interpretation collects three 3-valued structures at the exit node of the DSC. Two of these are the empty structure and the single element structure. The values of all four new instrumentation relations are definite in those structures, in particular $sorted_1() = 1$. The third structure is essentially the same as the rightmost structure of Fig. 5.2 but with values for the four new instrumentation relations. Because nothing is known about the `data`-values stored in list elements (and hence about the dle relationships between different elements), $sorted_1() = 1/2$ in this structure, and unary relations $sorted_2(v)$ and $sorted_3(v)$ have value $1/2$ for both individuals of the structure.

The three structures collected at the exit node of the DSC become the abstract input structures, i.e., the abstract value at the entry point of `InsertSort` for the next run of abstract interpretation on `InsertSort`. □

A DSC is also used to automatically construct the abstract input structures before the first run of abstract interpretation (line [1] in Fig. 5.3). This allows the user to specify the program's

inputs in the form of a program, which frees the user from having to know the details of the initial abstraction in use.

5.2.5 Success of Refinement for `InsertSort`

In all of the structures collected at the exit node of `InsertSort` by the second run of abstract interpretation, $sorted_1() = 1$. The permutation property also holds on all of the structures. These two facts establish the partial correctness of `InsertSort`. This process required one iteration of abstraction refinement, used the basic version of the specification (the vocabulary consisted of the relations of Figs. 2.3 and 2.6, together with the corresponding history relations), and needed no user intervention.

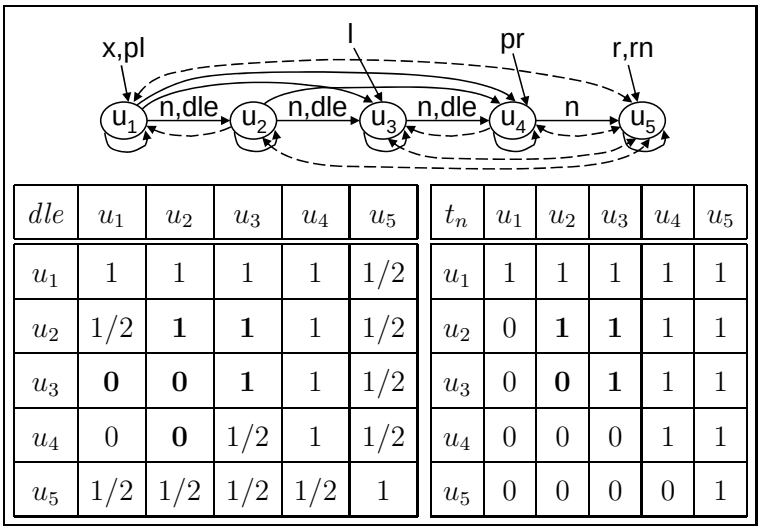
5.2.6 ILP-Based Refinement

5.2.6.1 Shortcomings of Subformula-Based Refinement

Procedure `InsertSort` consists of two nested loops (see Fig. 5.1). The outer loop traverses the list, setting pointer variable `r` to point to list nodes. For each iteration of the outer loop, the inner loop finds the correct place to insert `r`'s target, by traversing the list from the start using pointer variable `l`; `r`'s target is inserted before `l`'s target when `l->data > r->data`. Because `InsertSort` satisfies the invariant that all list nodes that appear in the list before `r`'s target are already in the correct order, the `data`-component of `r`'s target is less than the `data`-component of *all* nodes ahead of which `r`'s target is moved. Thus, `InsertSort` preserves the original order of elements with equal `data`-components, and `InsertSort` is a stable routine.

However, subformula-based refinement is not capable of establishing the stability of `InsertSort`. By considering only subformulas of the query (in this case, Formula (5.5)) as candidate instrumentation relations, the strategy is unable to introduce instrumentation relations that maintain information about the *transitive* successors with which a list node has the correct relative order.[5]

[5] In contrast, subformula-based refinement is capable of establishing the antistability of `InsertSort_AS`. When looking for a place to insert `r`'s target, this routine stops when `l->data >= r->data` and inserts `r`'s target before `l`'s target. The analysis need not establish anything about sortedness properties to observe that



dle	u_1	u_2	u_3	u_4	u_5
u_1	1	1	1	1	1/2
u_2	1/2	**1**	**1**	1	1/2
u_3	**0**	**0**	**1**	1	1/2
u_4	0	**0**	1/2	1	1/2
u_5	1/2	1/2	1/2	1/2	1

t_n	u_1	u_2	u_3	u_4	u_5
u_1	1	1	1	1	1
u_2	0	**1**	**1**	1	1
u_3	0	**0**	**1**	1	1
u_4	0	0	0	1	1
u_5	0	0	0	0	1

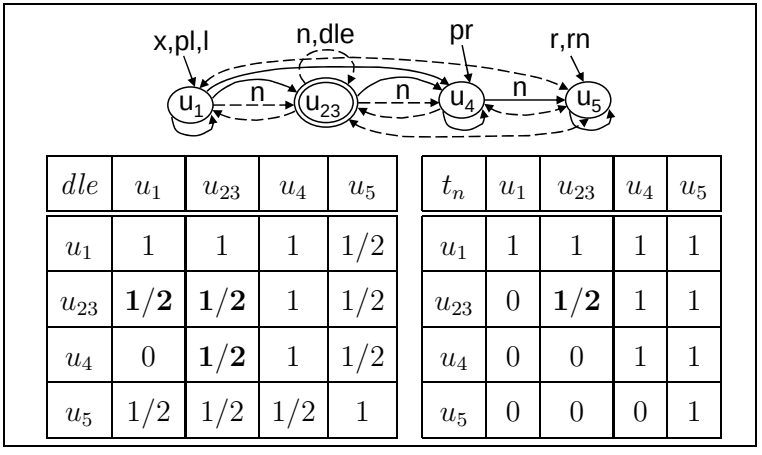
Figure 5.8 Structure $S_{5.8}$, which arises just before line [6] of Fig. 5.1. Unlabeled edges between nodes represent the dle relation.

5.2.6.2 Learning Instrumentation Relations

Fig. 5.8 shows the structure $S_{5.8}$, which arises during abstract interpretation just before line [6] of Fig. 5.1, together with a tabular version of relations t_n and dle. (We omit reachability relations from the figure for clarity.) After the assignment `l = x;`, nodes u_2 and u_3 have identical vectors of values for the unary abstraction relations. The subsequent application of canonical abstraction produces structure $S_{5.9}$, shown in Fig. 5.9. Bold entries of tables in Fig. 5.8 indicate definite values that are transformed into $1/2$ in $S_{5.9}$. Structure $S_{5.8}$ satisfies the sortedness invariant discussed above: every node among $u_1, ..., u_4$ has the dle relationship with all nodes appearing later in the list, except `r`'s target, u_5. However, a piece of this information is lost in structure $S_{5.9}$: $dle(u_{23}, u_{23}) = 1/2$, indicating that some nodes represented by summary node u_{23} might not be in sorted order with respect to their successors. We will refer to such abstraction steps as *information-loss points*.

An abstract structure transformer may temporarily create a structure S_1 that is not in the image of canonical abstraction [84]. The subsequent application of canonical abstraction transforms S_1

every list node is inserted before any other node with the same `data`-value. Once refinement introduces the appropriate instrumentation relations based on subformulas of the antistability query, TVLA is able to establish the antistability of `InsertSort_AS`.



dle	u_1	u_{23}	u_4	u_5
u_1	1	1	1	1/2
u_{23}	**1/2**	**1/2**	1	1/2
u_4	0	**1/2**	1	1/2
u_5	1/2	1/2	1/2	1

t_n	u_1	u_{23}	u_4	u_5
u_1	1	1	1	1
u_{23}	0	**1/2**	1	1
u_4	0	0	1	1
u_5	0	0	0	1

Figure 5.9 Structure $S_{5.9}$, corresponding to the transformation of $S_{5.8}$ by the statement on line [6] of Fig. 5.1. Unlabeled edges between nodes represent the dle relation.

into structure S_2 by grouping a set U_1 of two or more individuals of S_1 into a single summary individual of S_2. The loss of precision is due to one or both of the following circumstances:

- One of the individuals in U_1 possesses a property that another individual does not possess; thus, the property for the summary individual is $1/2$.
- Individuals in U_1 have a property in common, which cannot be recomputed precisely in S_2.

In both cases, the solution lies in the introduction of new instrumentation relations. In the former case, it is necessary to introduce a unary abstraction relation to keep the individuals of U_1 that possess the property from being grouped with those that do not. In the latter case, it is sufficient to introduce a non-abstraction relation of appropriate arity that captures the common property of individuals in U_1. The version of the ILP algorithm described in Sect. 4.3 can be used to learn formulas for the following three kinds of relations:[6]

Type I: Unary relation $r_1(v_1)$ with $E^+ = \{[v_1 \mapsto u]\}$ for one $u \in U_1$, and $E^- = \big\{[v_1 \mapsto u'] \;\big|\; u' \in U_1 \setminus \{u\}\big\}$.

Type II A: Unary relation $r_2(v_1)$ with $E^- = \big\{[v_1 \mapsto u] \;\big|\; u \in U_1\big\}$.

Type II B: Unary relation $r_2(v_1)$ with $E^+ = \big\{[v_1 \mapsto u] \;\big|\; u \in U_1\big\}$.

Type III A: Binary relation $r_3(v_1, v_2)$ with $E^- = \big\{[v_1 \mapsto u_1, v_2 \mapsto u_2] \;\big|\; u_1, u_2 \in U_1\big\}$.

Type III B: Binary relation $r_3(v_1, v_2)$ with $E^+ = \big\{[v_1 \mapsto u_1, v_2 \mapsto u_2] \;\big|\; u_1, u_2 \in U_1\big\}$.

Type I relations are intended to prevent the grouping of individuals with different properties, while Types II and III (II A, II B, III A, and III B) are intended to capture the common properties of individuals in U_1. Type III relations can be generalized to ternary and higher-arity relations.

[6]These are what are needed for our analysis framework, which uses abstractions that generalize predicate-abstraction domains. A fourth use of ILP provides a new technique for predicate abstraction itself: ILP can be used to identify nullary relations that distinguish a positive-example structure S from the other structures arising at a program point. Sect. 4.4 describes a version of the ILP algorithm that provides this capability. The steps of ILP go beyond merely forming Boolean combinations of existing relations (as in many refinement techniques based on predicate abstraction); ILP can create new relations by introducing quantifiers during the learning process.

Relations of Types II A and II B both capture the common properties of individuals in U_1. However, a relation of one type may be better suited for use in analysis than a relation of the other type. We will explain this shortly.

For the background logical structure that serves as input to ILP, we pass the structure S_1 identified at an information-loss point. We restrict the algorithm to use the relations of the structure that are used in the query and lose definite entries as a result of abstraction (e.g., t_n and dle in the above example). Definite entries (1 or 0) of those relations are then used to learn formulas that evaluate to 1 for every positive example and to 0 for every negative example.

We now describe how the variant of ILP described in Sect. 4.3 is able to learn a useful binary formula using structure $S_{5.8}$ of Fig. 5.8. (This formula will be used to define a binary relation $r_3(v_1, v_2)$ of Type III B.) The set of individuals of $S_{5.8}$ that are grouped by the abstraction is $U = \{u_2, u_3\}$, so the input set of positive examples is $\{[v_1 \mapsto u_2, v_2 \mapsto u_2], [v_1 \mapsto u_2, v_2 \mapsto u_3], [v_1 \mapsto u_3, v_2 \mapsto u_2], [v_1 \mapsto u_3, v_2 \mapsto u_3]\}$. The set of relations that lose definite values due to abstraction includes t_n and dle. Literal $dle(v_1, v_2)$ covers three of the four examples because it holds for assignments $[v_1 \mapsto u_2, v_2 \mapsto u_2]$, $[v_1 \mapsto u_2, v_2 \mapsto u_3]$, and $[v_1 \mapsto u_3, v_2 \mapsto u_3]$. The algorithm picks that literal and, because there are no negative examples, $dle(v_1, v_2)$ becomes the first disjunct. Literal $\neg t_n(v_1, v_2)$ covers the remaining positive example, $[v_1 \mapsto u_3, v_2 \mapsto u_2]$, and the algorithm returns the formula

$$\psi_{r_3}(v_1, v_2) \stackrel{\text{def}}{=} dle(v_1, v_2) \vee \neg t_n(v_1, v_2), \tag{5.7}$$

which can be re-written as $t_n(v_1, v_2) \Rightarrow dle(v_1, v_2)$.

Relation $r_3(v_1, v_2)$ allows the abstraction to maintain information about the transitive successors with which a list node has the correct relative order. In particular, although $dle(u_{23}, u_{23})$ is $1/2$ in $S_{5.9}$, $r_3(u_{23}, u_{23})$ is 1, which allows establishing the fact that all list nodes appearing prior to `r`'s target are in sorted order.

Other formulas, such as $dle(v_1, v_2) \vee t_n(v_2, v_1)$, are also learned using ILP (cf. Fig. 5.11). Not all of them are useful to the verification process, but introducing unnecessary instrumentation relations cannot harm the analysis, aside from increasing its cost.

Discussion

As explained in Sect. 5.2.3, the precision improvement from using a subformula of the query to define a new instrumentation relation p is mainly due to the use of the stored values for p in the query (or the definitions of other instrumentation relations) in place of p's defining formula. Picking the defining formulas of new instrumentation relations from among the subformulas of the query ensures that such reuse is always possible.

The relations learned by ILP do not necessarily match any of the subformulas of the query. In fact, assuming that subformula-based refinement fails to verify a property, other relations, such as $r_3(v_1, v_2)$, are needed to capture properties that hold at earlier points of the program but are lost due to abstraction. The precision improvement from introducing such an instrumentation relation p is due mainly to the sharpening of the relations in terms of which p is defined, i.e., the relations that occur in ψ_p. This sharpening is achieved through the application of the *coerce* operation to enforce additional integrity constraints that are generated automatically based on the definition of p. For instance, based on the definition of $r_3(v_1, v_2)$, the system generates the following two integrity constraints:

$$\forall\, v_1, v_2 \colon r_3(v_1, v_2) \wedge \neg dle(v_1, v_2) \Rightarrow \neg t_n(v_1, v_2) \tag{5.8}$$

$$\forall\, v_1, v_2 \colon r_3(v_1, v_2) \wedge t_n(v_1, v_2). \Rightarrow dle(v_1, v_2) \tag{5.9}$$

Because a relation p of Type II or III has the same value for all tuples that consist of elements of U_1 (that value is 0 in the case of Type II A and Type III A relations and 1 in the case of Type II B and Type III B relations), the abstraction step at an information-loss point does not cause a loss of precision in the value of p. At a later point during the analysis, the precise values of p allow the values of the relations that occur in p's defining formula to be sharpened via the automatically generated integrity constraints.

While the relations of Types II A and II B both capture the common properties of individuals in U_1, a relation of Type II A is generally better suited for generating integrity constraints. The defining formulas of the relations of Types II A and II B have the forms of Eqns. (5.10) and (5.11),

respectively:

$$\psi_{r_{2A}}(v_1) \stackrel{\text{def}}{=} (\exists\, v_2, v_3, \ldots, v_j \colon l_{1,1}(\cdot) \wedge l_{1,2}(\cdot) \wedge \ldots \wedge l_{1,k}(\cdot)) \tag{5.10}$$

$$\psi_{r_{2B}}(v_1) \stackrel{\text{def}}{=} (\exists\, v_2 \colon l_{2,1}(\cdot)) \vee (\exists\, v_2 \colon l_{2,2}(\cdot)) \vee \ldots \vee (\exists\, v_2 \colon l_{2,l}(\cdot)), \tag{5.11}$$

where the $l_{1,i}(\cdot)$ and the $l_{2,i}(\cdot)$ are logical literals. In our system, relation $r_{2B}(v_1)$ does not result in the automatic generation of integrity constraints, unless l (the number of disjuncts) is 1 or the defining formula contains no quantifiers. Relation $r_{2A}(v_1)$, on the other hand, results in the automatic generation of the following constraints (for each $i \in [1..j]$):

$$\forall\, v_1, v_2, \ldots, v_j \colon \neg r_{2A}(v_1) \wedge l_{1,1}(\cdot) \wedge \ldots \wedge l_{1,i-1}(\cdot) \wedge l_{1,i+1}(\cdot) \wedge \ldots \wedge l_{1,j}(\cdot) \Rightarrow \neg l_{1,i}(\cdot). \tag{5.12}$$

Thus, we can use ILP to learn relations of Type II A and II B, although generally only the Type II A relations have an effect on the precision of the analysis.

The above conclusions also hold for the distinction between the relations of Types III A and III B. However, when applied to examples of Type III B, ILP frequently learns defining formulas that do not contain quantifiers, e.g., the relation $r_3(v_1, v_2)$ discussed above. Such relations result in the generation of integrity constraints that can help improve the precision of the analysis, as we demonstrated using the relation $r_3(v_1, v_2)$.

5.2.6.3 ILP and the Refinement Loop

ILP gives us a powerful mechanism for learning new abstractions. At present, we generally employ subformula-based refinement first, because the cost of this strategy is reasonable (see Sect. 5.3) and the strategy is often successful. In this mode of operation, if the call to *instrum* on line [6] of Fig. 5.3 returns no formulas and adds no relations to the set of abstraction relations, we turn to the ILP strategy. Sect. 5.3 also describes our experiments with an alternative mode of operation, in which subformula-based refinement is turned off and only ILP-based refinement is applied.

During each iteration of subformula-based refinement, we save logical structures at information-loss points. Upon the failure of subformula-based refinement, we invoke the ILP

algorithm, as described in Sect. 5.2.6. To lower the cost of the analysis, we prune the returned set of formulas. For each learned formula φ, we use a conservative test to check whether the introduction of a new instrumentation relation p defined by φ can sharpen tuples of relations that occur in φ. (We call formulas that pass the test *effective*.) This test uses the results of the previous round of analysis to simulate the introduction of p. We then use effective formulas learned by ILP to define new instrumentation relations, and use these relations to refine the abstraction by performing the steps of lines [7] and [8] of Fig. 5.3, as done for subformula-based refinement. Our implementation can learn relations of all the types described in Sect. 5.2.6: unary, binary, as well as nullary. At present, the number of instrumentation relations used by an analysis in TVLA has a significant impact on the cost of the analysis. In the experiments reported below and in Sect. 5.3, we touch on the implications of learning some but not all types of relations described in Sect. 5.2.6.

Example 5.4 When invoking ILP to learn binary formulas only (i.e., of Type III) during the verification of the stability of `InsertSort`, thirteen effective binary formulas are learned using the ILP algorithm, among them Formula (5.7). Upon completion of the refinement steps, the subsequent run of the analysis successfully verifies the stability of `InsertSort`. □

5.3 Experimental Evaluation

To evaluate the method presented in this chapter, we extended TVLA to perform iterative abstraction refinement, and applied it to three queries and five programs (see Fig. 5.10). Besides `InsertSort`, the test programs included sorting procedures `BubbleSort` and `InsertSort_AS`, list-merging procedure `Merge`, and *in-situ* list-reversal procedure `Reverse`.

The DSC that we used in our tests is a procedure to generate unsorted lists of arbitrary length, in the case of all programs but `Merge`. For `Merge`, the DSC is a procedure to generate pairs of unsorted lists.

First, we describe the results of applying iterative abstraction refinement that invokes ILP to learn binary formulas only (i.e., of Type III). Fig. 5.10 shows that the method was able to generate the right instrumentation relations for TVLA to establish all properties that we expect to hold.

Test Program	*sorted* # instrum rels total/SF/ILP	*stable* # instrum rels total/SF/ILP	*antistable* # instrum rels total/SF/ILP
BubbleSort	31/4/0	32/5/0	32/5/0
InsertSort	39/4/0	53/5/13	40/5/0
InsertSort_AS	39/4/0	40/5/0	40/5/0
Merge	27/4/0	28/5/0	28/5/0
Reverse	23/4/0	24/5/0	24/5/0

Figure 5.11 The numbers of instrumentation relations used during the last iteration of abstraction refinement. The three numbers in each cell give the total number of relations, the number of relations introduced by subformula-based refinement, and the number of (effective) relations learned by ILP, respectively

Namely, TVLA succeeds in demonstrating that all three sorting routines produce sorted lists, that `BubbleSort`, `InsertSort`, and `Merge` are stable routines, and that `InsertSort_AS` and `Reverse` are antistable routines.

Indefinite answers are indicated by $1/2$ entries. *It is important to understand that all of the occurrences of* $1/2$ *in Fig. 5.10 are the most precise correct answers*. For instance, the result of applying `Reverse` to an unsorted list is usually an unsorted list; however, in the case that the input list happens to be in non-increasing order, `Reverse` produces a sorted list. Consequently, the most precise answer to the query is $1/2$, not 0.

Test Program	*sorted*	*stable*	*antistable*
BubbleSort	1	1	1/2
InsertSort	1	1	1/2
InsertSort_AS	1	1/2	1
Merge	1/2	1	1/2
Reverse	1/2	1/2	1

Figure 5.10 Results from applying iterative abstraction refinement to the verification of properties of programs that manipulate linked lists. Columns 2, 3, and 4 correspond to the queries stated in Formulas (5.1), (5.5), and (5.4), respectively

Fig. 5.11 shows the numbers of instrumentation relations used during the last iteration of abstraction refinement. The number of relations defined by subformulas of the query is small relative to the total number of instrumentation relations. The only verification test during which ILP learns effective relations is the verification of the stability of `InsertSort`.

Test Program	*sorted*	*stable*	*antistable*
BubbleSort	97	101	66
InsertSort	36	139	6.4
InsertSort_AS	36	6	37
Merge	13	25	16
Reverse	4.5	4.5	3.8

Figure 5.12 Total execution times for applying iterative abstraction refinement (in seconds)

Fig. 5.12 gives execution times that were collected on a 3GHz PC with 3.7GB of RAM running CentOS 4 Linux. The longest-running analysis, which verifies that `InsertSort` is stable, takes 2.3 minutes. Eleven of the analyses take under a minute. The rest take under 2 minutes. The total time for the 15 tests is less than 10 minutes. The maximum amount of memory used by TVLA to perform the analyses varied from just under 2 megabytes to 85 megabytes.[7]

Sortedness of `BubbleSort` and `InsertSort` are the only queries in our set to which TVLA has been applied before this work [53]. For these queries, the performance of iterative abstraction refinement is very close to the performance of the analysis when the user carefully chooses the right instrumentation relations.

Because ILP was necessary for the verification of only one of the eight properties that we expect to hold (namely, the stability of `InsertSort`), we devised two other sets of experiments:

- We tested the resiliency of ILP by varying the types of relations that ILP was allowed to learn during the verification of the stability of `InsertSort`. (See the discussion of Fig. 5.13.)
- We forced ILP to be invoked during the verification of the other seven properties that we expect to hold, by turning off subformula-based refinement. (See the discussion of Fig. 5.14.)

These are discussed below.

[7]TVLA is written in Java. Here we report the maximum of total memory minus free memory, as returned by Runtime.

Results From Varying the Types of Relations Learned

Fig. 5.13 shows the results of varying the types of relations that are learned by ILP during abstraction refinement when verifying the stability of `InsertSort`. In short, when ILP is invoked to learn relations of (a) only Type II, (b) only Type III, or (c) all types together, the resulting abstractions are sufficiently precise to establish the stability of `InsertSort`.

We were somewhat surprised that our system was capable of establishing the stability of `InsertSort` when allowed to learn only Type II relations. Moreover, when learning only relations of Type II, the total analysis time, as well as the analysis time for the last round of the analysis, is 14 seconds faster than when learning relations of Type III. (The total analysis time when learning only relations of Type II is just over 2 minutes.) The key relation that captures the same information as the relation $r_3(v_1, v_2)$ is the relation $r_2(v_1)$ (of Type II B) that is defined by

$$\psi_{r_2}(v_1) \stackrel{\text{def}}{=} \exists\, v_2 \colon\ t_n(v_2, v_1) \wedge \neg dle(v_2, v_1). \tag{5.13}$$

This relation has the meaning "v_1 has a transitive predecessor that is out of order". The introduction of $r_2(v_1)$ as an instrumentation relation results in the automatic generation of the following two integrity constraints:

$$\forall\, v_1, v_2 \colon\ \neg r_2(v_1) \wedge \neg dle(v_1, v_2) \Rightarrow \neg t_n(v_1, v_2) \tag{5.14}$$

$$\forall\, v_1, v_2 \colon\ \neg r_2(v_1) \wedge t_n(v_1, v_2) \Rightarrow dle(v_1, v_2). \tag{5.15}$$

Constraints (5.14) and (5.15) achieve the same effect as Constraints (5.8) and (5.9).

Incidentally, the value of $r_2(v_1)$ for every individual in every structure collected at the exit node of `InsertSort` is 0. Thus, the analysis also establishes that `InsertSort` does, in fact, produce a sorted list. Although, the query was the *stability* query, in this case the application of ILP learned a relation that allowed the analysis to establish something stronger, namely, that every output list is also *sorted*.

When learning only the relations of Type I, the ILP algorithm does not learn properties common to all nodes that are summarized together. As a result, it does not learn the relations $r_2(v_1)$, $r_3(v_1, v_2)$, or any other relation that is capable of "encoding" the sortedness invariant. When learning only the relations of Type I, the analysis is unable to establish the stability of `InsertSort`.

Test Program	Answer	# of ILP instrum rels	Execution time (sec.)		
			total	ILP	last iter
Type III	1	13	139	1.4	62
Type II	1	4	125	3.7	48
Type I	1/2	23	157	11.8	73
All Types	1	40	318	17.5	224

Figure 5.13 The results of using ILP to learn relations of different types during the verification of the stability of `InsertSort`. The third column gives the number of instrumentation relations introduced by ILP. The last three columns give the execution times: the total execution time, the execution time of ILP, and the execution time of the last round of the analysis (the DSC and `InsertSort`)

Intuitively, relations of Types II and III allow universal properties that hold for all individuals to be captured. (In the case of relations of Type II A or Type III A, the relations capture properties whose negations hold for all individuals.) Such relations appear to be more likely than relations of Type I to capture actual properties of programs, as opposed to coincidental properties that arise during an exploration of the program's reachable configurations.

The costs of the invocation of ILP (and the effectiveness tests) are between 1.4 and 17.5 seconds. These costs were incurred at 124 information-loss points. We consider these costs to be low given that our implementation of the ILP algorithm is unoptimized and we take no advantage of the fact that an ILP computation at one information-loss point often produces the same results as ILP computations at other information-loss points.

Fig. 5.13 shows that the use of a higher number of relations in the analysis comes at a cost. While the learning of relations of all types increases the chance of obtaining a definite answer, the resulting analysis suffers from a slowdown of nearly three times, when compared to the analysis that uses ILP to learn only the relations of Type II. This slowdown is concentrated in the last iteration of the analysis: the slowdown on the last iteration is more than fourfold. Thus, we believe that for better performance, the learning of relations of Type I should only be performed if the learning of relations of Types II and III does not lead to a definite answer to the query.

Results From Turning Off Subformula-Based Refinement

In an attempt to understand better the power of our ILP-based refinement approach, we performed an experiment in which we disabled the use of subformula-based refinement. At first, we were not able to verify automatically any of the eight properties that we expect to hold. (See the 1 entries of Fig. 5.10.) Subformula-based refinement appears to stave off imprecision sufficiently to help ILP learn useful formulas. In an effort to learn useful relations in the absence of subformula-based refinement, we made two changes to the basic refinement mechanism. First, we ensured that each round of analysis proceeds to completion instead of being terminated as soon as imprecision is detected. (This results in more information-loss points at which ILP can be applied.) Second, we turned off the effectiveness test for pruning the set of formulas returned by ILP. Both of these changes are intended to allow ILP to learn more formulas.

When attempting to verify that the sorting procedures do, in fact, produce sorted lists, ILP learns the Type II A relation $r_{21}(v_1)$ that is defined by

$$\psi_{r_{21}}(v_1) \stackrel{\text{def}}{=} \exists\, v_2 \colon\ n(v_1, v_2) \wedge \neg dle(v_1, v_2). \tag{5.16}$$

Note that the definition of $r_{21}(v_1)$ has the same meaning as the negation of the definition of $inOrder_{dle,n}(v_1)$. The steps of line [7] of Fig. 5.3 result in the replacement of the formula $\forall\, v_2 : n(v_1, v_2) \Rightarrow dle(v_1, v_2)$ in the sortedness query with the use of the stored value $\neg r_{21}(v_1)$. The modified query evaluates to 1 in all structures collected at the exit nodes of the three sorting procedures, thus establishing the sortedness property fully automatically. The total analysis times were 3.5 minutes for `InsertSort` and `InsertSort_AS`, and 7.5 minutes for `BubbleSort`.

When attempting to verify that `Merge` is stable, ILP learns the Type II A relation $r_{22}(v_1)$ that is defined by

$$\psi_{r_{22}}(v_1) \stackrel{\text{def}}{=} \exists\, v_2 \colon\ t_n^0(v_2, v_1) \wedge \neg t_n(v_2, v_1). \tag{5.17}$$

The definition of $r_{22}(v_1)$ has the meaning "v_1 had a transitive predecessor (in the input list) that is no longer a transitive predecessor". The introduction of the relation $r_{22}(v_1)$ does not result in a definite answer to the stability query because $\psi_{r_{22}}$ is not a subformula of the query. As a result, the stored values of $r_{22}(v_1)$ cannot be used in the evaluation of the query. However, the absence of

individuals with values 1 or $1/2$ for the relation $r_{22}(v_1)$ in the structures collected at the exit node of `Merge` implies that the procedure is stable. This establishes that `Merge` is stable, although not as mechanically as for the sortedness property—it required a manual examination of the results (and for us to recognize that stability follows from all individuals having the value 0 for r_{22} in the structures collected at the exit node). The total analysis time was 4.5 minutes.

When attempting to verify that `Reverse` is antistable, ILP learns the Type II A relation $r_{23}(v_1)$ that is defined by

$$\psi_{r_{23}}(v_1) \stackrel{\text{def}}{=} \exists\, v_2 \colon\, t_n^0(v_1, v_2) \wedge \neg t_n(v_2, v_1). \tag{5.18}$$

The definition of $r_{23}(v_1)$ has the meaning "v_1 had a transitive successor (in the input list) that is not a transitive predecessor currently". As in the case of `Merge`, the introduction of the learned relation allows the analysis to establish the property (here, the antistability of `Reverse`) in an indirect way. The total analysis time was 18 seconds.

When attempting to verify that `BubbleSort` and `InsertSort` are stable, ILP learns the Type II A relation $r_{24}(v_1)$ that is defined by

$$\psi_{r_{24}}(v_1) \stackrel{\text{def}}{=} \exists\, v_2 \colon\, t_n^0(v_2, v_1) \wedge dle(v_2, v_1) \wedge \neg t_n(v_2, v_1). \tag{5.19}$$

The definition of $r_{24}(v_1)$ has the meaning "v_1 had a transitive predecessor (in the input list) that is no longer a transitive predecessor even though it has a value that is less than or equal to that of v_1". The absence of individuals with values 1 or $1/2$ for the relation $r_{24}(v_1)$ in the structures collected at the exit nodes of the procedures would imply that the procedures are stable. Unfortunately, the relation $r_{24}(v_1)$ has some $1/2$ values in a structure collected at the exit nodes of `BubbleSort` and `InsertSort`. Thus, the present mechanism is unable to establish the stability of those routines without the use of subformula-based refinement. However, the introduction of the relation $r_{24}(v_1)$ can be qualified as a partial success.

Finally, when attempting to verify the antistability of `InsertSort_AS`, ILP does not learn one single formula that captures the property (although, a combination of two formulas does). In the absence of subformula-based refinement, the structures arising at information-loss points do not

have sufficient precision to enable the learning of a formula such as

$$\exists\, v_2 \colon\, t_n^0(v_1, v_2) \wedge dle(v_1, v_2) \wedge dle(v_2, v_1) \wedge \neg t_n(v_2, v_1),$$

which has the meaning "v_1 had a transitive successor (in the input list) that is not a transitive predecessor currently even though it has a value that is equal to that of v_1". The reason for ILP's inability to learn such a formula is that without the introduction of a new instrumentation relation that is defined by the equality subformula $dle(v_1, v_2) \wedge dle(v_2, v_1)$, the analysis will never have sufficient precision in the equality formula to enable it to become part of an ILP-learned relation. Thus, subformula-based refinement is sometimes needed to improve the precision of the evaluation of simple combinations of formulas that are relevant to the query (such as the combination of $dle(v_1, v_2)$ and $dle(v_2, v_1)$ that makes up the equality formula) to provide ILP with enough precision to learn relations based on such combinations.

Fig. 5.14 summarizes the above experiment. Non-empty cells correspond to properties that we expect to hold. The entries for the stability of `Merge` and the antistability of `Reverse` are labeled 1^* to distinguish these answers from the 1 answers that were obtained fully automatically in the case of the sortedness properties of the three sorting procedures. For the three remaining cases, we used $1/2$ to indicate that no definite conclusion about the property can be drawn based on the given analysis, although in the case of the stability of `BubbleSort` and `InsertSort`, we claim a partial success for the reasons given above.

Test Program	*sorted*	*stable*	*antistable*
BubbleSort	1	1/2	
InsertSort	1	1/2	
InsertSort_AS	1		1/2
Merge		1^*	
Reverse			1^*

Figure 5.14 Results from applying iterative abstraction refinement with subformula-based refinement disabled to the verification of properties of programs that manipulate linked lists. Empty cells indicate answers that are expected to be $1/2$

5.4 Additional Experiments: Beyond Acyclic Lists

We performed four additional experiments to test the applicability of our method to other queries and data structures. In the first experiment, subformula-based refinement successfully

verified that the *in-situ* list-reversal procedure `Reverse` indeed produces a list that is the reversal of the input list, assuming that the input list is acyclic. The query that expresses this property is $\forall\, v_1, v_2 \colon n(v_1, v_2) \Leftrightarrow n^0(v_2, v_1)$. This experiment took only 3.5 seconds and used less than 2 megabytes of memory. We will consider the application of `Reverse` to arbitrary lists in Sect. 5.4.1.

The second and third experiments involved two programs that manipulate binary-search trees. `InsertBST` inserts a new node into a binary-search tree, and `DeleteBST` deletes a node from a binary-search tree. For both programs, subformula-based refinement successfully verified the query that the nodes of the tree pointed to by variable `t` remain in sorted order at the end of the programs:

$$\begin{aligned} \forall\, v_1 \colon r_t(v_1) \Rightarrow (\forall v_2 \colon\ & (\mathit{left}(v_1, v_2) \Rightarrow dle(v_2, v_1)) \\ & \wedge\ \ (\mathit{right}(v_1, v_2) \Rightarrow dle(v_1, v_2))). \end{aligned} \tag{5.20}$$

The initial specifications for the analyses included only three standard instrumentation relations, similar to those listed in Fig. 2.6. Relation $r_t(v_1)$ from Formula (5.20), for example, distinguishes nodes in the (sub)tree pointed to by `t`. Sect. 6.1 discusses the relations employed in analyses of programs that use type `Tree` in more detail. The DSC used for the analyses nondeterministically constructs a binary-search tree by allocating one new node at a time and inserting it into the tree in the appropriate position according to its `data`-value. The `InsertBST` experiment took 9 seconds and used less than 3 megabytes of memory, while the `DeleteBST` experiment took approximately 3.3 minutes and used 63 megabytes of memory.

5.4.1 Properties of `Reverse` When Applied to Possibly-Cyclic Linked Lists

In the fourth experiment, subformula-based refinement successfully verified the expected properties of the transformations performed by `Reverse` on possibly-cyclic linked lists. Additionally, we used a simple *progress monitor* to establish the termination of the procedure on any input. We describe this experiment in detail in the remainder of this section.

Fig. 5.15 shows the list-reversal algorithm that we analyze. The algorithm performs the reversal in place using three pointer variables, `x`, `y`, and `t`. The `n` field of list nodes is reversed on lines [7] and [8]. During the execution of the statements on those lines, `x` points to the next node to be

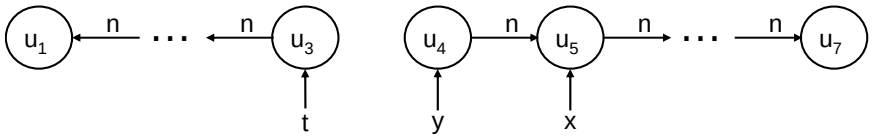


Figure 5.16 Logical structure $S_{5.16}$ that represents a store that arises prior to line [7] of `Reverse` when the algorithm is applied to an acyclic list

processed, `y` points to the node whose `n` field is reversed, and `t` points to the predecessor of that node.

First, let us consider how `Reverse` processes an acyclic list L_a with head u_1, pointed to by `x`. Fig. 5.16 shows a logical structure $S_{5.16}$ that represents a store that arises before line [7] during the application of `Reverse` to L_a. At this point the `n` edges of nodes $u_1, \ldots, u_3$ have been reversed, while the remaining edges retain their original orientation. The statements on lines [7] and [8] replace the `n` edge from u_4 to u_5 with an `n` edge from u_4 to u_3. The traversal continues until, on the last loop iteration, `t` is set to point to u_7's predecessor in the input list, `y` is set to point to u_7, and `x` is set to `NULL`. The subsequent execution of lines [7] and [8] reverses the remaining `n` edge. The head of the reversed list is u_7, pointed to by `y`. As in the input list, no node lies on a cycle. The last statement of the procedure (the assignment on line [10]) restores `x` as the head pointer. The transformation described above can be stated formally using history relations as follows:

```
[1]  void reverse(List *x)
[2]  { List *y = NULL;
[3]    while (x != NULL) {
[4]      t = y;
[5]      y = x;
[6]      x = x->n;
[7]      y->n = NULL;
[8]      y->n = t;
[9]    }
[10]   x = y;
[11] }
```

Figure 5.15 In-situ list reversal algorithm

$$same_{r_{n,x}}() \wedge same_{c_n}() \wedge \forall\, v_1, v_2 \colon n(v_1, v_2) \Leftrightarrow n^0(v_2, v_1). \tag{5.21}$$

Let us consider how `Reverse` processes a list L_c that consists of a single cycle without a panhandle, such as the acyclic list L_a discussed above, but with an additional `n` edge from u_7 to u_1. The behavior of `Reverse` on list L_c is nearly identical to its behavior on list L_a. The outgoing

n edges are reversed one at a time until, on the last iteration, t is set to point to u_7, y is set to point to u_1, and x is set to NULL. The subsequent execution of lines [7] and [8] reverses the remaining n edge from u_7 to u_1. The head of the reversed list remains u_1, pointed to by y. Every list node still lies on a cycle. The last statement of the procedure (the assignment on line [10]) restores x as the head pointer. The transformation of lists such as L_c also obeys the property specified in Formula (5.21).

Now, we discuss how Reverse processes a panhandle list L_p. Initially, the procedure advances the three pointer variables, x, y, and t, down the panhandle, reversing the n edges out of y. After the panhandle is processed, the algorithm proceeds with the processing of the cycle. Fig. 5.17(a) shows a logical structure that represents a store that arises prior to line [7] while Reverse processes nodes that lie on the cycle. Until Reverse completes the processing of the cycle, the steps are identical to the steps taken during the processing of lists L_a and L_c. Note that the orientation of the n edges in the panhandle is reversed when the loop body is executed with x pointing to u_5 (while reversing the backedge at the end of processing the cycle). As a result, the algorithm proceeds along the reversed n edges down the panhandle, reestablishing the original orientation of those edges. Fig. 5.17(b) shows a logical structure that represents a store that arises prior to line [7] while Reverse processes panhandle nodes for the second time. Instead of reversing every n edge in the list, as it does for lists L_a and L_c,[8] the algorithm reverses the direction of every n edge on the cycle but reestablishes the original direction of the n edges in the panhandle. The cyclicity property of all nodes remains as it was on input. The head of the output list remains u_1, pointed to by y. The last statement of the procedure (the assignment on line [10]) restores x as the head pointer. The transformation described above can be stated formally using history relations as follows:

$$
\begin{aligned}
& same_{r_{n,x}}() \wedge same_{c_n}() \wedge \\
\forall\, v_1, v_2 : \quad & (c_n^0(v_1) \wedge c_n^0(v_2)) \quad \wedge \quad (n(v_1, v_2) \Leftrightarrow n^0(v_2, v_1)) \\
\vee \quad & \neg(c_n^0(v_1) \wedge c_n^0(v_2)) \quad \wedge \quad (n(v_1, v_2) \Leftrightarrow n^0(v_1, v_2)).
\end{aligned}
\tag{5.22}
$$

[8] Reversing every n edge of a panhandle list is not possible because it requires the shared node (u_5 in Fig. 5.17) to have two outgoing n edges.

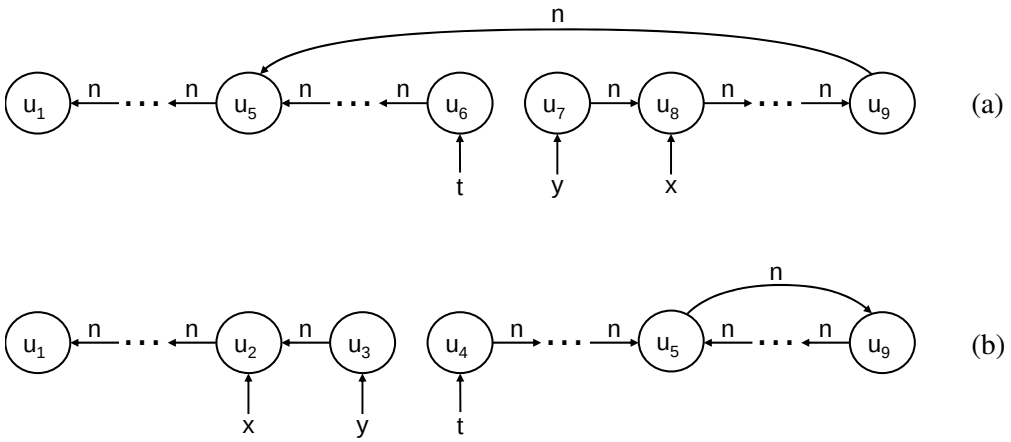


Figure 5.17 Logical structures that represents stores that arise prior to line [7] of `Reverse` when the algorithm is applied to a panhandle list. (a) Logical structure that represents a store that arises while `Reverse` processes nodes that lie on the cycle, i.e., after processing nodes that lie in the panhandle once. (b) Logical structure that represents a store that arises while `Reverse` processes nodes that lie on the panhandle for the second time, i.e., after processing nodes that lie on the cycle.

Note that while the behavior of `Reverse` on lists consisting of a cycle without a panhandle can be described by Formula (5.21), as we mentioned above, it can also be described by Formula (5.22). (The case described by formula $\neg(c_n^0(v_1) \wedge c_n^0(v_2))$ never arises.)

5.4.1.1 A DSC for Possibly-Cyclic Linked Lists

We use the methodology of Sect. 5.2.4 to construct the 3-valued structures that represent all valid inputs to the procedure (before each round of analysis for `Reverse`). Fig. 5.18(a) shows the DSC that constructs all acyclic lists; it is identical to the DSC shown in Fig. 5.7. The slight modification shown on the right nondeterministically constructs a (cyclic or acyclic) linked list pointed to by `x`. This is achieved by setting `y` to point to the last list node on line [7], nondeterministically setting `h` to point to some list node (or `NULL`) on line [9], and setting `y->n` to point to `h` on line [15] if `y` is non-`NULL` (possibly completing a cycle). If `h` is `NULL`, the DSC constructs an acyclic list. If `h` points to the head of the list, the DSC constructs a list consisting of a cycle with no panhandle. If `h` is neither `NULL` nor points to the head of the list, the DSC constructs a panhandle list.

Abstract interpretation of the DSC of Fig. 5.18(b) constructs an abstract representation of all linked lists pointed to by `x`. When testing the application of a procedure to acyclic lists, we select only those structures collected at the exit of the DSC that satisfy the following formula:

$$(\exists\, v\colon\, r_{n,x}(v)) \wedge (\forall v\colon\, r_{n,x}(v) \Rightarrow \neg c_n(v)) \tag{5.23}$$

We will refer to input abstractions satisfying Formula (5.23) as *type Acyclic*. When testing the application of a procedure to cyclic lists without a panhandle, we select only those structures collected at the exit of the DSC that satisfy the following formula:

$$(\exists\, v\colon\, r_{n,x}(v)) \wedge (\forall v\colon\, r_{n,x}(v) \Rightarrow c_n(v)) \tag{5.24}$$

We will refer to input abstractions satisfying Formula (5.24) as *type Cyclic*. When testing the application of a procedure to panhandle lists, we select only those structures collected at the exit of the DSC that satisfy the following formula:

$$(\exists\, v_1\colon\, r_{n,x}(v_1) \wedge \neg c_n(v_1)) \wedge (\exists\, v_2\colon\, r_{n,x}(v_2) \wedge c_n(v_2)) \tag{5.25}$$

We will refer to input abstractions satisfying Formula (5.25) as *type Panhandle*. Note that Formulas (5.23)–(5.25) ensure that each of the input types admits only non-empty lists. Note also that the three types represent disjoint collections of data structures. Additionally, the cross product of the set of lists represented by type *Acyclic* and the set of lists represented by type *Cyclic* is in a one-to-one correspondence with the set of lists represented by type *Panhandle:* the acyclic-list component corresponds to the panhandle of a panhandle list and the cyclic-list component corresponds to its cycle. We will make use of these facts in Sect. 5.4.1.4.

5.4.1.2 Abstraction-Refinement Steps

After an abstraction of the appropriate valid input is constructed by analyzing the DSC, the abstract interpretation collects all structures that arise at all program points of `Reverse`. To check if `Reverse` satisfies the expected properties, we check if all structures collected at the exit of `Reverse` satisfy the appropriate query (Formula (5.21) when testing the application of the procedure to lists

```
[1]  List *x = NULL;

[3]  int sz = sizeof(List);
[4]  while (?)  {
[5]    List *t = malloc(sz);

[10]   t->n = x;
[11]   x = t;
[12] }
```

(a)

```
List *x, *y, *h;                [1]
x = y = h = NULL;               [2]
int sz = sizeof(List);          [3]
while (?)  {                    [4]
  List *t = malloc(sz);         [5]
  // save the last node         [6]
  if (y == NULL) y = t;         [7]
  // save a node (or NULL)      [8]
  if (?)  h = t;                [9]
  t->n = x;                     [10]
  x = t;                        [11]
}                               [12]
// if y and h are non-NULL,     [13]
// this will create a cycle     [14]
if (y != NULL) y->n = h;        [15]
```

(b)

Figure 5.18 (a) The Data-Structure Constructor for acyclic linked lists. This DSC is identical to the one shown in Fig. 5.7. (b) The Data-Structure Constructor for possibly-cyclic linked lists (including acyclic and panhandle lists). The differences between the two versions appear in bold.

p	ψ_p (after call to *instrum*)	ψ_p (final version)
$rev_1()$	$same_{r_{n,x}}() \wedge same_{c_n}() \wedge$ $\forall v_1, v_2 \colon n(v_1, v_2) \Leftrightarrow n^0(v_2, v_1)$	$same_{r_{n,x}}() \wedge same_{c_n}() \wedge rev_2()$
$rev_2()$	$\forall v_1, v_2 \colon n(v_1, v_2) \Leftrightarrow n^0(v_2, v_1)$	$\forall v_1 : rev_3(v_1)$
$rev_3(v_1)$	$\forall v_2 \colon n(v_1, v_2) \Leftrightarrow n^0(v_2, v_1)$	$\forall v_2 : rev_4(v_1, v_2)$
$rev_4(v_1, v_2)$	$n(v_1, v_2) \Leftrightarrow n^0(v_2, v_1)$	$rev_5(v_1, v_2) \wedge rev_6(v_2, v_1)$
$rev_5(v_1, v_2)$	$n(v_1, v_2) \Rightarrow n^0(v_2, v_1)$	$n(v_1, v_2) \Rightarrow n^0(v_2, v_1)$
$rev_6(v_2, v_1)$	$n^0(v_2, v_1) \Rightarrow n(v_1, v_2)$	$n^0(v_2, v_1) \Rightarrow n(v_1, v_2)$

Figure 5.19 Instrumentation relations created by subformula-based refinement when the application of `Reverse` is checked against the query expressed in Formula (5.21) on an input abstraction of either type *Acyclic* or *Cyclic*.

represented by type *Acyclic* and Formula (5.22) when testing the application of the procedure to lists represented by type *Panhandle*; we can check either query when testing the application of the procedure to lists represented by type *Cyclic*).

Both queries (Formulas (5.21) and (5.22)) contain formula $n(v_1, v_2) \Leftrightarrow n^0(v_2, v_1)$ as a subformula. Because this formula evaluates to $1/2$ under any assignment that maps v_1 and v_2 to the same summary individual with a $1/2$-valued self-loop for the relation n, it should come as no surprise that the first run of abstract interpretation returns an indefinite answer, whether we are checking Formula (5.21) or Formula (5.22).

Column 2 of Fig. 5.19 shows the instrumentation relations that are created as a result of the call to *instrum* after Formula (5.21) evaluated to $1/2$ on a structure collected at the exit of `Reverse`, given an input abstraction of either type *Acyclic* or *Cyclic*. During transition-relation refinement of `Reverse`, the use of Formula (5.21) in the query is replaced with the use of the stored value $rev_1()$ and occurrences of the defining formulas for $rev_1, \ldots, rev_6$ are replaced with the use of the corresponding relation symbols. Column 3 of Fig. 5.19 shows the final version of the defining formulas for the new relations.

Column 2 of Fig. 5.20 shows the instrumentation relations that are created as a result of the call to *instrum* after Formula (5.22) evaluated to $1/2$ on a structure collected at the exit of `Reverse`, given an input abstraction of either type *Panhandle* or *Cyclic*. Note that subformulas of $acycSame(v_1, v_2)$, i.e., $\neg(c_n^0(v_1) \wedge c_n^0(v_2)) \wedge (n(v_1, v_2) \Leftrightarrow n^0(v_1, v_2))$ were not introduced. This is because refinement was triggered by imprecise evaluation on a structure that had a single concrete individual in the panhandle. However, relation rev_3 is capable of maintaining the key property of nodes in the panhandle with enough precision, so that another refinement iteration is not required. During transition-relation refinement of `Reverse`, the use of Formula (5.22) in the query is replaced with the use of the stored value $rev_1()$ and occurrences of the defining formulas for $rev_1, \ldots, rev_8$ are replaced with the use of the corresponding relation symbols. Column 3 of Fig. 5.20 shows the final version of the defining formulas for the new relations.

After the introduction of the new instrumentation relations (Fig. 5.19 or 5.20, depending on the query being verified), the abstract interpretation of the DSC is performed using an extended vocabulary that contains the new instrumentation-relation symbols. The subsequent abstract interpretation of `Reverse` succeeds: in all of the structures collected at the exit, $rev_1() = 1$.

5.4.1.3 Establishing that `Reverse` Terminates

We can establish that `Reverse` terminates using a few unary core relations and a simple progress monitor. We introduce a collection of unary core *state relations*, $state_0(v)$, $state_1(v)$, and $state_2(v)$.[9] Every time the reversal of the `n` pointer of the list node pointed to by `y` is completed (after line [8] of Fig. 5.15), the node's state is changed to the next state. (The state relations carry no semantics with respect to the pointer values of nodes; they simply record the "visit counts" for each node.) For each state relation s, we create a copy of s, which is used to save the values of relation s at the start of the currently-processed loop iteration (after line [3] of Fig. 5.15). We give the new relations the superscript lh to indicate that they hold the *loop-head* values. The first abstract operation of each iteration of the loop takes a snapshot of the current states of nodes: $state_i^{lh}(v) \leftarrow state_i(v)$, for each $i \in [0..2]$ and each assignment of v to an individual in the abstract

[9] The state relations are *not* added to the set of abstraction relations, $\mathcal{A}$.

p	ψ_p (after call to *instrum*)	ψ_p (final version)
$rev_1()$	$same_{r_{n,x}}() \wedge same_{c_n}() \wedge$ $\forall\, v_1, v_2 \colon cycRev(v_1, v_2) \vee acycSame(v_1, v_2)$	$same_{r_{n,x}}() \wedge same_{c_n}() \wedge rev_2()$
$rev_2()$	$\forall\, v_1, v_2 \colon cycRev(v_1, v_2) \vee acycSame(v_1, v_2)$	$\forall\, v_1 : rev_3(v_1)$
$rev_3(v_1)$	$\forall\, v_2 \colon cycRev(v_1, v_2) \vee acycSame(v_1, v_2)$	$\forall\, v_2 : rev_4(v_1, v_2)$
$rev_4(v_1, v_2)$	$cycRev(v_1, v_2) \vee acycSame(v_1, v_2)$	$rev_5(v_1, v_2) \vee acycSame(v_1, v_2)$
$rev_5(v_1, v_2)$	$cycRev(v_1, v_2)$	$(c_n^0(v_1) \wedge c_n^0(v_2)) \wedge rev_6(v_2, v_1)$
$rev_6(v_1, v_2)$	$n(v_1, v_2) \Leftrightarrow n^0(v_2, v_1)$	$rev_7(v_1, v_2) \wedge rev_8(v_2, v_1)$
$rev_7(v_1, v_2)$	$n(v_1, v_2) \Rightarrow n^0(v_2, v_1)$	$n(v_1, v_2) \Rightarrow n^0(v_2, v_1)$
$rev_8(v_2, v_1)$	$n^0(v_2, v_1) \Rightarrow n(v_1, v_2)$	$n^0(v_2, v_1) \Rightarrow n(v_1, v_2)$

Figure 5.20 Instrumentation relations created by subformula-based refinement when the application of `Reverse` is checked against the query expressed in Formula (5.22) on an input abstraction of either type *Panhandle* or *Cyclic*. For compactness, we refer to formula $(c_n^0(v_1) \wedge c_n^0(v_2)) \wedge (n(v_1, v_2) \Leftrightarrow n^0(v_2, v_1))$ as $cycRev(v_1, v_2)$ and to formula $\neg(c_n^0(v_1) \wedge c_n^0(v_2)) \wedge (n(v_1, v_2) \Leftrightarrow n^0(v_1, v_2))$ as $acycSame(v_1, v_2)$.

structure being processed. Additionally, it asserts that `x` does not point to a list node in state 2 at the head of the loop (at that point, `x` points to the node whose `n` edge is about to be reversed). The last operation of every loop iteration performs a progress test by asserting the following formula:

$$\exists\, v\colon\ \big(state_0^{lh}(v) \wedge state_1(v) \vee state_1^{lh}(v) \wedge state_2(v)\big)$$
$$\bigwedge \forall\, v_1 \neq v\colon\ \bigwedge_{i\in[0..2]}\ \big(state_i^{lh}(v_1) \Leftrightarrow state_i(v_1)\big)$$

The assertion ensures that one node's state makes forward progress (the first line of the assertion) and that no other node changes state (the second line of the assertion). Together with the assertion that `x` does not point to a list node in state 2 at the start of the loop, the above progress monitor establishes that each list node is visited at most twice, thus establishing that the algorithm terminates.

5.4.1.4 Performance

The tables shown in Fig. 5.21 give execution times that were collected on a 3GHz Linux PC. The rows indicate the type of data structures assumed as input, and the columns indicate the query to be verified. In each case, one round of abstraction refinement was required to obtain the definite answer 1 to the query. In other words, two rounds of analysis were performed for both the DSC and `Reverse`: the first analysis round of the DSC and `Reverse` used the initial abstraction (the core relations of Fig. 2.3, core relation roc_n, the instrumentation relations of Fig. 3.19, and the history relations of Sect. 2.2.2), while the second round used the final abstraction, which additionally included the relations of Fig. 5.19 or 5.20, depending on the query. For a given abstraction, the cost of the DSC analysis is nearly identical for all input types because the general DSC of Fig. 5.18(b) constructs an abstraction of all input types, from which structures that represent the chosen input type are selected at the end using Formula (5.23), (5.24), or (5.25). To gain a better understanding of the cost of verifying `Reverse` proper, the tables also include the execution times for the last analysis round (using the final abstraction) of `Reverse`, excluding the analysis time for the DSC.

The tables of Fig. 5.21 show that the use of tree-shaped-sfe_n maintenance techniques (see Sect. 3.3.2) in place of acyclic-sfe_n maintenance techniques (see Sect. 3.3.1) for maintaining the relation sfp_n results in a reduction of the total analysis time by a factor in the range of 2.8-4.8.

The highest-cost analyses are those that include type *Panhandle* as input. Using tree-shaped-sfe_n maintenance, the last iteration of the analysis of `Reverse` with the input abstraction of type *Panhandle*/*Cyclic* takes approximately 2.5 minutes (the total execution time is approximately 4.5 minutes). The last iteration of the analysis of `Reverse` when the input abstraction is of any other type takes under 13 seconds. The majority of the total analysis cost in those cases is due to the use of the general DSC, which could be specialized to produce input abstractions of type *Acyclic* or *Cyclic* more efficiently (e.g., using the DSC shown in Fig. 5.18(a)).

The two tables of Fig. 5.21 share many qualitative characteristics. Below we draw some conclusions from Fig. 5.21(b), but all of the conclusions can be drawn from Fig. 5.21(a) equally well. As expected, the cost of the last run of the analysis of `Reverse` when the input abstraction is of type *Acyclic*/*Cyclic* is close to the sum of the cost when the input abstraction is of type *Acyclic* and the cost when the input abstraction is of type *Cyclic*. Similarly, the cost of the last run of the analysis of `Reverse` when the input abstraction is of type *Panhandle*/*Cyclic* is close to the sum of the cost when the input abstraction is of type *Panhandle* and the cost when the input abstraction is of type *Cyclic*. Curiously, the total cost of the analysis when the input abstraction is of type *Panhandle* is slightly higher than the total cost when the input abstraction is of type *Panhandle*/*Cyclic*. The reason is that a structure of type *Cyclic* triggers the refinement process at an earlier point. The resulting shorter execution of the first run of the analysis of `Reverse` explains the counterintuitive relation of total execution times. The cost of the analysis when the input abstraction is of type *Cyclic* (both total cost and the cost of the last iteration of the analysis of `Reverse`) is similar for the two queries. The panhandle query (Formula (5.22)) results in the introduction of a more complex abstraction (cf. Figs. 5.20 and 5.19), so the costs in column 3 of Fig. 5.21(a) are slightly higher.

The cost of verifying that `Reverse` terminates is negligible (when compared to the cost of verifying the query) because the progress monitor does not increase the size of the reachable state space.

The three analyses represented by the right column of Fig. 5.21(a), i.e., analyses using the panhandle query (Formula (5.22)) and acyclic-sfe_n maintenance, used a maximum of approximately

Input Type	Query	
	acyclic total/last	panhandle total/last
Acyclic	161.2/11.8	
Cyclic	208.1/28.2	232.9/34.6
Acyclic/Cyclic	219.7/38.6	
Panhandle		1320.3/782.3
Panhandle/Cyclic		1249.3/810.3

(a)

Input Type	Query	
	acyclic total/last	panhandle total/last
Acyclic	57.1/4.6	
Cyclic	65.6/8.4	71.5/9.0
Acyclic/Cyclic	69.1/12.5	
Panhandle		277.1/147.8
Panhandle/Cyclic		268.1/154.9

(b)

Figure 5.21 Execution times in seconds using (a) acyclic-sfe_n maintenance for maintaining the relation sfp_n; (b) tree-shaped-sfe_n maintenance for maintaining the relation sfp_n. In row labels, input types "*Acyclic/Cyclic*" and "*Panhandle/Cyclic*" denote an abstraction that represents lists of either type. The label of column 2 (query "acyclic") denotes the query of Formula (5.21). The label of column 3 (query "panhandle") denotes the query of Formula (5.22). Empty cells indicate inappropriate input/query combinations. The first number in each column represents the total execution time for all iterations of the analysis (on both the DSC and `Reverse`). The second number represents the execution time for only the last iteration of the analysis of `Reverse` (and not the DSC).

170 MB of memory, as reported by the Java Runtime. All other analyses required significantly less memory.

As a sanity check, we studied the number of distinct 3-valued structures collected at all points of `Reverse` during the last run of the analysis. As we expected, that information is identical when the analysis relies on acyclic-sfe_n maintenance and when it relies on tree-shaped-sfe_n maintenance, thus providing a cross-validation of the implementation of the two methods. The structure counts are shown in Fig. 5.22. The figure shows that when the input abstraction is of type *Cyclic*, the same number of structures is collected with either query. Also, the number of structures collected when the input abstraction is of type *Acyclic*/*Cyclic* is the sum of the number when the input abstraction is of type *Acyclic* and the number when the input abstraction is of type *Cyclic*. Similarly, the number of structures collected when the input abstraction is of type *Panhandle*/*Cyclic* is the sum of the number when the input abstraction is of type *Panhandle* and the number when the input abstraction is of type *Cyclic*.

Input Type	Query	
	acyclic	panhandle
Acyclic	103	
Cyclic	162	162
Acyclic/*Cyclic*	265	
Panhandle		921
Panhandle/*Cyclic*		1083

Figure 5.22 The number of distinct 3-valued structures collected during the last iteration of the analysis of `Reverse` (and not the DSC). Rows and columns have the same meaning as in Fig. 5.21.

Additionally, we used the data collected in our experiments to answer an instance of the following general question: "Can we predict how much work needs to be done for analysis X when we know how much work is done for related analyses Y and Z?" Given the correspondence of lists represented by type *Panhandle* with combinations of lists represented by type *Acyclic* and lists represented by type *Cyclic*, we made a prediction about the number of structures collected during the analysis of `Reverse` when the input abstraction is of type *Panhandle* (using the panhandle query) based on the number of structures collected during the analyses of `Reverse` when the input abstraction is of types *Acyclic* and *Cyclic* (using the acyclic query). Let a_n, c_n, and p_n, represent the numbers of structures collected at CFG node n during the analysis of `Reverse` when the input

abstraction is of type *Acyclic*, *Cyclic*, and *Panhandle*, respectively. For a CFG node n that lies outside the loop of `Reverse`, we expect that $p_n = a_n * c_n$. For a CFG node n that lies inside the loop, we expect that

$$p_n = c_{entry} * a_n + a_{exit} * c_n + c_{exit} * a_n, \tag{5.26}$$

where c_{entry} is the number of structures at the entry node of `Reverse` when the input abstraction is of type *Cyclic*, a_{exit} is the number of structures collected at the exit of `Reverse` when the input abstraction is of type *Acyclic*, and c_{exit} is the number of structures collected at the exit of `Reverse` when the input abstraction is of type *Cyclic*. The intuition behind the first summand of Formula (5.26) is that every acyclic structure collected at n (when the input abstraction is of type *Acyclic*) can be extended to c_{entry} panhandle structures at n. These structures represent the states in which the panhandle is being reversed before the cycle is entered. The intuition behind the second summand of Formula (5.26) is that every cyclic structure collected at n (when the input abstraction is of type *Cyclic*) can be extended to a_{exit} panhandle structures. These structures represent the states in which the cycle is being reversed after the panhandle has been reversed. Finally, the intuition behind the third summand of Formula (5.26) is that every acyclic structure collected at n can be extended to c_{exit} panhandle structures. These structures represent the states in which the panhandle is being un-reversed after the cycle has been reversed. The summation of predicted values for p_n over the nodes n of `Reverse` gives 858 structures. This prediction is a little short of the actual number (921). This relatively small discrepancy is probably due to the fact that our prediction for a run using the panhandle query (which leads to the abstraction of Fig. 5.20) is based on numbers for the right-hand side quantities of Formula (5.26) gathered from runs that use a slightly different abstraction, namely, Fig. 5.19. The more complex abstraction introduced when verifying the panhandle query apparently creates a few additional intermediate structures.

Note that the sum of the numbers of structures collected during the analyses when the input abstraction is of types *Acyclic* and *Cyclic* is much lower than the number of structures collected during the analysis when the input abstraction is of type *Panhandle*. The sum of the execution times of the analyses when the input abstraction is of types *Acyclic* and *Cyclic* is also much lower than the execution time of the analysis when the input abstraction is of type *Panhandle*. A possible

extension of this work is to infer properties of `Reverse` when applied to input abstraction of type *Panhandle* from properties of `Reverse` when applied to input abstractions of types *Acyclic* and *Cyclic*. To make this possible, we need to find a way to infer properties of heap configurations from properties of components of those configurations. The concept of *local heaps* introduced by Rinetzky et al. is relevant in this line of research [82].

5.5 Related Work

The work reported in this chapter is similar in spirit to counterexample-guided abstraction refinement [4, 16, 22, 28, 36, 47, 49, 72]. A key difference between this work and prior work in the model-checking community is the abstract domain: prior work has used abstract domains that are fixed, finite, Cartesian products of Boolean values (i.e., predicate-abstraction domains), and hence the only relations introduced are nullary relations. (The use of such domains is known as *predicate abstraction*.) Our work applies to a richer class of abstractions—3-valued structures—that generalize predicate-abstraction domains. The abstraction-refinement algorithm described in this chapter can introduce unary, binary, ternary, etc. relations, in addition to nullary relations. While we demonstrated our approach using shape-analysis queries, this approach is applicable in any setting in which first-order logic is used to describe program states.

A second distinguishing feature of our work is that the method is driven not by counterexample traces, but instead by imprecise results of evaluating a query (in the case of subformula-based refinement) and by loss of information during abstraction steps (in the case of ILP-based refinement). There do not currently exist theorem provers for first-order logic extended with transitive closure capable of identifying infeasible error traces [38]; hence we needed to develop techniques different from those used in SLAM, BLAST, etc. SLAM identifies the shortest prefix of a spurious counterexample trace that cannot be extended to a feasible path; in general, however, the first information-loss point occurs before the end of the prefix. Information-loss-guided refinement can identify the earliest points at which information is lost due to abstraction, as well as what new instrumentation relations need to be added to the abstraction at those points. A potential advantage of counterexample-guided refinement over information-loss-guided refinement is that the former is

goal-driven. Information-loss-guided refinement can discover many relationships that do not help in establishing the query. To alleviate this problem, we restricted the ILP algorithm to only use relations that occur in the query.

Abstraction-refinement techniques from the abstract-interpretation community are capable of refining domains that are not based on predicate abstraction. In [41], for example, a polyhedra-based domain is dynamically refined. Our work is based on a different abstract domain, and led us to develop some new approaches to abstraction refinement, based on machine learning.

In the abstract-interpretation community, a strong (albeit often unattainable) form of abstraction refinement has been identified in which the goal is to make abstract interpretation complete (a.k.a. "optimal") [29]. In our case, the goal is to extend the abstraction just enough to be able to answer the query, rather than to make the abstraction optimal.

In [78], weakest preconditions are used to generate nullary instrumentation relations, which are then generalized manually. The technique presented there produces precise results if it terminates, but is not guaranteed to terminate for all cases. In contrast, our method is guaranteed to terminate, and automatically generates interesting non-nullary relations, such as the unary relation $inOrder_{dle,n}(v)$, which is crucial for showing sortedness, and the binary relation $r_3(v_1, v_2)$, defined by Formula (5.7), which allows the analysis to establish the stability of `InsertSort`.

The concept of a data-structure constructor, which non-deterministically constructs all valid inputs to the program, can be thought of as a mechanism for closing open programs, and hence is related to such work as [17] and [91].

Other work that relates machine-learning techniques and program analysis includes [2, 55, 80]. The Strauss tool [2] uses a machine-learning approach to discovering specifications of API protocols. The underlying premise is that even programs with bugs contain hints that can reveal correct protocols. The Cooperative Bug Isolation project [55] instruments programs and collects information about their executions. Statistical and machine-learning techniques are used to find bugs by mining the information about crashing and non-crashing runs. The technique for finding the most-precise abstract value for a set of concrete stores (expressed as a logical formula) successively approximates the result from below [80]; this technique is related to algorithm Find-S from

machine learning [65, §2.4]—they both search a space of hypotheses to find the most specific hypothesis that satisfies the positive examples (the input concrete stores).

Our work represents a new connection between program analysis and machine learning: it shows how ILP can be used as part of an abstraction-refinement loop to learn an appropriate abstraction.

Chapter 6

Total Correctness of the Deutsch-Schorr-Waite Tree-Traversal Algorithm

The present chapter discusses the automated verification of the *total correctness* (partial correctness and termination) of the Deutsch-Schorr-Waite (DSW) tree-traversal algorithm. Past approaches have involved hand-written proofs of complicated invariants to verify the partial correctness of the algorithm. Even with some automation, these efforts were usually laborious: a proof performed in 2002 with the help of the Jape proof editor took 152 pages [8]. The key advantage of our abstract-interpretation approach over proof-theoretic approaches is that a relatively small number of concepts are involved in defining an abstraction of the structures that can arise on any execution, and verification is then carried out automatically by symbolic exploration of all memory configurations that can arise.

The chapter is organized as follows: Sect. 6.1 presents the relations that we use to encode memory configurations that include binary trees. Sect. 6.2 shows how one can take advantage of the fact that the only kind of data structure that a program manipulates is a binary tree. Sect. 6.3 discusses the DSW algorithm in detail. Sect. 6.4 presents an extension of the abstraction defined in Sect. 6.1 that we employ for establishing the partial correctness of DSW. Sect. 6.5 explains the technique that we use for showing the termination of the algorithm. Sect. 6.6 presents experimental results. Sect. 6.7 makes observations in regards to some of the choices made in this work and discusses some future directions. Sect. 6.8 discusses related work.

```
typedef struct node {
  struct node *left;
  int data;
  struct node *right;
} *Tree;
```

(a)

Relation	Intended Meaning
$x(v)$	Does pointer variable `x` point to heap cell v?
$left(v_1, v_2)$	Does the `left` field of v_1 point to v_2? (Is v_2 the left child of v_1?)
$right(v_1, v_2)$	Does the `right` field of v_1 point to v_2? (Is v_2 the right child of v_1?)

(b)

Figure 6.2 (a) Declaration of a binary-tree datatype in C. (b) Core relations used for representing the stores manipulated by programs that use type `Tree`.

6.1 Binary-Tree Abstractions

Fig. 6.2 gives the definition of a C binary-tree datatype, and lists the core relations that would be used to represent the stores manipulated by programs that use type `Tree`, such as the store in Fig. 6.1. Unary relations represent pointer variables, and binary relations *left* and *right* represent the `left` and `right` fields of a `Tree` node. Fig. 6.4(a) shows 2-valued structure $S_{6.4}$, which represents the store of Fig. 6.1 using the relations of Fig. 6.2.

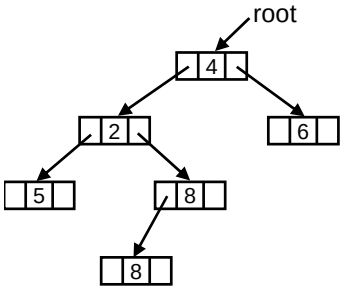


Figure 6.1 A possible store for a binary tree

Fig. 6.3 lists some instrumentation relations that are important for the analysis of programs that use type `Tree`. Instrumentation relations that involve reachability properties, such as relation $r_x(v)$, often play a crucial role in the definitions of abstractions. These relations have the effect of keeping disjoint subtrees summarized separately. Fig. 6.4(b) shows 2-valued structure $S_{6.4}$, which represents the store of Fig. 6.1 using the core relations of Fig. 6.2, as well as the instrumentation relations of Fig. 6.3.

If all unary relations are abstraction relations ($\mathcal{A} = \mathcal{R}_1$), the canonical abstraction of 2-valued logical structure $S_{6.4}$ is $S_{6.5}$, shown in Fig. 6.5, with all tree nodes not pointed to by `root` represented by the summary individual at the bottom. In $S_{6.4}$, nodes in the left subtree of `root`'s target are indistinguishable from those in its right subtree according to $\mathcal{A}$ (consisting of relations

p	**Intended Meaning**	**Defining Formula**
$down(v_1, v_2)$	Do the `left` or `right` fields of v_1 point to v_2? (Is v_2 a child of v_1?)	$left(v_1, v_2) \vee right(v_1, v_2)$
$t_{down}(v_1, v_2)$	Is v_2 reachable from v_1 along `left` and `right` fields?	$down^*(v_1, v_2)$
$r_x(v)$	Is v reachable from pointer variable `x` along `left` and `right` fields?	$\exists\, v_1 \colon x(v_1) \wedge t_{down}(v_1, v)$

Figure 6.3 Defining formulas of instrumentation relations commonly employed in analyses of programs that use type `Tree`. There is a separate reachability relation r_x for every program variable `x`. (Recall that $down^*(v_1, v_2)$ is a shorthand for (**RTC** $v_1', v_2' \colon down(v_1', v_2'))(v_1, v_2)$.)

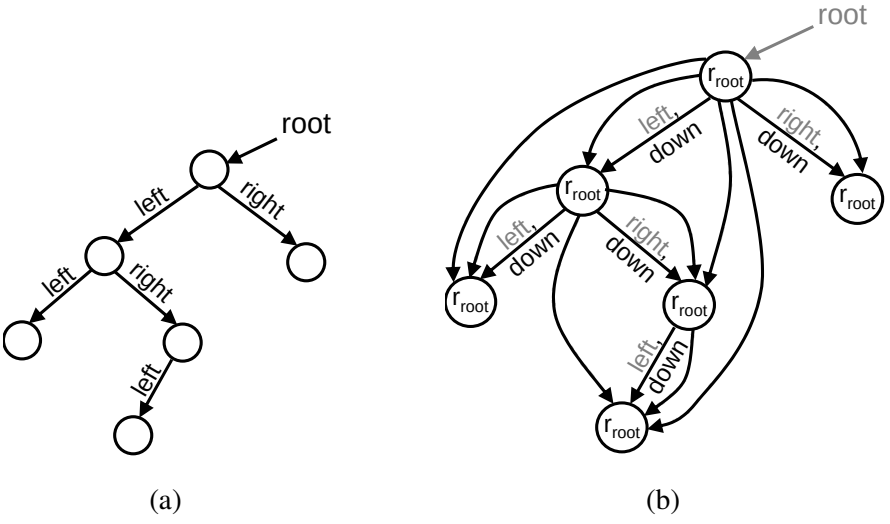


Figure 6.4 A logical structure $S_{6.4}$ that represents the store shown in Fig. 6.1 in graphical form: (a) $S_{6.4}$ with relations of Fig. 6.2. (b) $S_{6.4}$ with relations of Figs. 6.2 and 6.3 (relations of Fig. 6.2 appear in grey). Unlabeled (curved) arcs between nodes represent the t_{down} relation. Self-loops of the t_{down} relation (corresponding to the reflexive tuples) have been omitted to reduce clutter.

$x(v)$ and $r_x(v)$ for each program variable x). $S_{6.5}$ represents all trees with two or more elements, with the root node pointed to by program variable `root`.

6.2 Analyzing Programs that Manipulate (Only) Trees

When analyzing a program in which each data structure at every point is a tree (a property that we will call *treeness*), it is possible to take advantage of this fact to reduce the (abstract) state space that is explored. This is achieved by having the analysis perform a semantic reduction after each step to filter out non-trees that may have crept into the representation. When the analysis relies on the program to maintain treeness, to guarantee that the results are sound, the analysis must check that treeness is preserved at every step. We address the latter obligation first. The techniques described below are applicable whenever one wishes to analyze programs in which all input, output, and intermediate data structures are trees. We call such analyses *tree-specific shape analyses*; our DSW analysis is an example of a particular tree-specific shape analysis. (Other work in which tree-specific shape analyses have been developed includes [35, 51, 52].)

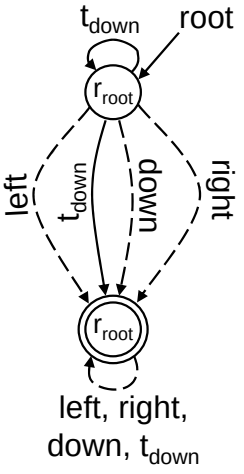


Figure 6.5 A 3-valued structure $S_{6.5}$ that is the canonical abstraction of structure $S_{6.4}$. In addition to $S_{6.4}$, $S_{6.5}$ represents any tree of size 2 or more that is pointed to by program variable `root`.

6.2.1 Checking that Treeness is Maintained.

The analyzer checks that treeness is maintained by asserting certain logical formulas that capture the conditions under which the execution of a program statement could result in a violation of treeness. Before the computation of a transfer function, the logical formulas of corresponding assertions are evaluated. If a formula *possibly fails to hold*, i.e., does not evaluate to 1, then an error report is issued and the analysis is terminated.

For purposes of this thesis, a binary tree is a structure containing no cycles and no nodes with multiple incoming `left` or `right` pointers. (Our definition disallows the sharing of subtrees, and thus is more restrictive than the traditional definition that merely requires there to be at most one

path between any pair of nodes. This is not an inherent limitation of TVLA; if the sharing of subtrees is to be permitted, the restriction on sharing can be relaxed—see footnote 2.)

Given a data structure that satisfies the data-structure invariants for a binary tree, only one type of statement has the potential to transform the data structure into one that violates some of those properties, namely, a statement of the form `x->sel = y` (where `sel` can be `left` or `right`), which creates a new `sel`-connection in the data structure. Two logical formulas capture the conditions that guarantee that the application of the transformer for a statement of the form `x->sel = y` maintains treeness. The first formula captures the precondition for $down$ to remain acyclic:

$$\forall v_1, v_2 \colon x(v_1) \wedge y(v_2) \Rightarrow \neg t_{down}(v_2, v_1) \tag{6.1}$$

The second formula captures the precondition for the statement to avoid introducing sharing:[1]

$$\forall v_1, v_2 \colon y(v_2) \Rightarrow \neg down(v_1, v_2)^2 \tag{6.2}$$

6.2.2 Semantic Reduction for Trees.

After each application of an abstract transformer, we perform a semantic reduction to filter out non-trees that may have crept into the abstract structures computed by the transformer. The reduction is implemented as an application of $coerce$ to enforce integrity constraints that express data-structure invariants.

For instance, relation $down$ is given the attributes "acyclic" and "invfunction". The "acyclic" attribute of $down$ results in the automatic generation of the following integrity constraint:

$$\forall v_1, v_2 \colon t_{down}(v_1, v_2) \wedge t_{down}(v_2, v_1) \quad \Rightarrow \quad v_1 = v_2 \tag{6.3}$$

[1] As explained in Sect. 6.3, we ensure that `x->sel` is `NULL` prior an assignment of the form `x->sel = y`, so the assignment indeed creates a new `sel`-connection.

[2] If we relaxed the restriction on the sharing of subtrees, then, in place of Formula (6.2), we would employ a slightly more complex formula that precludes the possibility of creating two paths between a pair of tree nodes v_1 and v_4 (one path that existed prior to the statement, and the other that was created due to the introduction of the new `sel` edge from `x` to `y`):
$\forall v_1, v_2, v_3, v_4 \colon t_{down}(v_1, v_4) \wedge t_{down}(v_1, v_2) \wedge x(v_2) \wedge y(v_3) \Rightarrow \neg t_{down}(v_3, v_4)$

The "invfunction" attribute of $down$ results in the automatic generation of the following integrity constraint:

$$\forall v_1, v_2 \colon (\exists v \colon down(v_1, v) \wedge down(v_2, v)) \quad \Rightarrow \quad v_1 = v_2 \tag{6.4}$$

Operation $coerce$ is applied at certain steps of the algorithm, e.g., after the application of an abstract transformer, to enforce Constraints (6.3) and (6.4), along with a few others, to help prevent the analysis from admitting non-trees, and thereby possibly losing precision.

6.3 Deutsch-Schorr-Waite Tree-Traversal Algorithm

The original Deutsch-Schorr-Waite algorithm reverses the direction of `left` and `right` pointers, as it traverses the tree [87]. It attaches two bits, `mark` and `tag`, to each node. The `mark` bit serves to prevent multiple visits to nodes on a cycle or in shared subtrees. The `tag` bit records whether, during the traversal of reversed pointers, a node was reached from its left or right child.

In [56], Lindstrom gave a variant that eliminated the need for both bits, provided the input data structure contains no cycles. His insight was that one could treat the visit step at an internal node as a kind of pointer-rotation operation, and that completion of the tree-traversal could be established by having the algorithm watch for a distinguished value that serves as a kind of sentinel. In this chapter, we actually consider the Lindstrom variant, but continue to refer to it as Deutsch-Schorr-Waite (DSW). Another connection between our analysis (of the Lindstrom variant) and the original version of DSW is discussed briefly in Sect. 6.7.

Fig. 6.6 shows two versions of the Deutsch-Schorr-Waite algorithm. The left-hand column shows a version adapted from [56], also known as Lindstrom scanning. The right-hand column shows a slightly modified version of the algorithm that we used in our work. There are two differences between the two versions.

First, the constant `-1` on lines [5] and [13] has been replaced with `SENTINEL`, where `SENTINEL` is assumed to be a reference to a distinguished node that is not part of the input tree. In TVLA, pointer values can either equal `NULL` (corresponding to the situation in which the pointer does not point to any heap object) or point to a heap object that was allocated by `malloc`. In this sense,

```
[1]  void traverse(Tree *root) {
[2]    Tree *prev, *cur, *next;

[3]    if (root == NULL)
[4]      return;
[5]    prev = -1;
[6]    cur = root;
[7]    while (1) {
         // Save the left subtree
[8]      next = cur->left;

         // Rotate pointers
[9]      cur->left = cur->right;
[10]     cur->right = prev;
         // Move forward
[11]     prev = cur;
[12]     cur = next;
[13]     if (cur == -1)
           // Traversal completed
[14]       break;
[15]     if (cur == NULL) {
           // Swap prev and cur
[16]       cur = prev;
[17]       prev = NULL;
[18]     }
[19]   }
[20] }
```

(a)

```
void traverse(Tree *root) {   [1]
  Tree *prev, *cur,           [2]
       *next, *tmp;           [3]
  if (root == NULL)           [4]
    return;                   [5]
  prev = SENTINEL;            [6]
  cur = root;                 [7]
  while (1) {                 [8]
    // Save the left subtree
    next = cur->left;         [9]
    // Rotate pointers
    tmp = cur->right;         [10]
    // Maintain treeness
    cur->right = NULL;        [11]
    cur->right = prev;        [12]
    cur->left = NULL;         [13]
    cur->left = tmp;          [14]
    // Move forward
    prev = cur;               [15]
    cur = next;               [16]
    if (cur == SENTINEL)      [17]
      // Traversal completed
      break;                  [18]
    if (cur == NULL) {        [19]
      // Swap prev and cur
      cur = prev;             [20]
      prev = NULL;            [21]
    }                         [22]
  }                           [23]
}                             [24]
```

(b)

Figure 6.6 (a) Original version of the Deutsch-Schorr-Waite algorithm (adapted from [56]). (b) Modified version of the Deutsch-Schorr-Waite algorithm that was analyzed using TVLA. (The differences appear in bold.)

TVLA follows the semantics of Java, in which new non-NULL pointer values can be generated only via memory-allocation operations.

Second, a purely local transformation (involving the introduction of one temporary variable tmp) has been applied to lines [9]–[10]:

```
                                          [10] tmp = cur->right;
                                          // Maintain treeness
[9] cur->left = cur->right;       ⟹      [11] cur->right = NULL;
[10] cur->right = prev;                   [12] cur->right = prev;
                                          [13] cur->left = NULL;
                                          [14] cur->left = tmp;
```

This really involved three transformations:

1. Assignment statements of the form `x->sel1 = y->sel2` have been normalized to statement sequences `tmp = y->sel2; x->sel1 = tmp` (see lines [10] and [14] of Fig. 6.6(b)).

2. Assignment statements of the form `x->sel = y` have been normalized to statement sequences `x->sel = NULL; x->sel = y` (see lines [11]–[12] and [13]–[14] of Fig. 6.6(b)). This ensures that statements of the form `x->sel = y` can never destroy existing `sel`-paths in the data structure, thus simplifying the task of maintaining information about the reachability of tree nodes from program variables.

3. Assignments `cur->right = NULL` and `cur->right = prev` have been moved to lines [11] and [12] (before assignments to `cur->left`). This change prevents the right child of `cur`'s target from temporarily having two incoming edges after the assignment to `cur->left` on line [14].[3] The resulting algorithm maintains the invariant that the nodes of the input tree always make up one or two data structures that satisfy the binary-tree properties: after the assignment on line [14] of Fig. 6.6(b), the nodes of the input tree make up two trees, one

[3]Only the assignment `cur->right = NULL` needs to be moved to achieve the desired effect. We moved both assignments for clarity.

rooted at `next`'s target, and the other rooted at `cur`'s target; the original root is a descendant of `cur`'s target.

Transformations 1 and 2 above are simple normalizations that one could expect to find in a translation of programs written in a high-level language into a lower-level intermediate representation. Transformation 3 prevents the temporary sharing of `cur`'s right subtree (it would otherwise briefly become `cur`'s left and `cur`'s right subtree). We could relax our restriction on sharing and analyze the version of the algorithm that does not include transformation 3 (Sect. 6.7 discusses how we would approach this task), but we chose to verify total correctness and preservation of treeness for the slightly modified version of the DSW algorithm shown in Fig. 6.6(b). Because of transformation 3, the techniques of Sect. 6.2 apply in the analysis of this version; we now describe this version in detail.

For each tree node n, the body of the `while` loop is executed three times with `cur` pointing to n. Each time that n is considered, its `left` and `right` pointers are rotated in a counter-clockwise fashion on lines [10]–[14] of Fig. 6.6(b) (cf. lines [9] and [10] of Fig. 6.6(a)). After the third such execution, the original values for the `left` and `right` pointers are re-established, as we explain below.

Before the first execution of lines [10]–[14] of Fig. 6.6(b) with `cur` pointing to n, no nodes in the subtrees rooted at l or r (n's left and right subtrees in the original tree) have been visited, and no `left` or `right` pointers of nodes in the subtrees rooted at l or r have been modified. In this situation, we say that n is in *state 0*. Fig. 6.7(a) illustrates this situation.

A pointer to node l, the left child of n prior to the rotation of n's `left` and `right` pointers, is saved in `next` on line [9]. After the rotation, the traversal continues by moving into the (sub)tree rooted at `next`, i.e., l (see lines [15] and [16]). When `cur` becomes null, the values of `cur` and `prev` are swapped on lines [20] and [21]. This causes the traversal to backtrack to the most recently visited node that had a right subtree in the original tree.

When the traversal backtracks to n, the algorithm reaches lines [10]–[14] of Fig. 6.6(b) for the second time with `cur` pointing to n. At this point, all nodes in l's subtree and no nodes in r's subtree have been visited. The `left` and `right` pointers of nodes in l's subtree have been rotated

three times and restored to their original values. No `left` or `right` pointers of nodes in r's subtree have been modified. In this situation we say that n is in *state 1*. Fig. 6.7(b) illustrates this situation.

A pointer to node r, the left child of n prior to the second rotation of n's pointers, is saved in `next`. After the rotation, the traversal continues by moving into the (sub)tree rooted at r (see lines [15] and [16]). Once again, the algorithm backtracks when `cur` is null. When the traversal backtracks to n, the algorithm reaches lines [10]–[14] of Fig. 6.6(b) for the third (and final) time with `cur` pointing to n. At this point, all nodes in l's and r's subtrees have been visited. The `left` and `right` pointers of nodes in both subtrees have been rotated three times and restored to their original values. In this situation we say that n is in *state 2*. Fig. 6.7(c) illustrates this situation.

After the subsequent execution of lines [10]–[14] of Fig. 6.6(b) with `cur` pointing to n, n's `left` and `right` pointers are restored to their original values. At this point, all nodes in the subtree rooted at n have been visited, and all `left` and `right` pointers in the subtree have been rotated three times and restored to their original values. In this situation we say that n is in *state 3*. Fig. 6.7(d) illustrates this situation.

The algorithm traverses the tree *in order*, visiting each node n three times: (1) while following the original `left` pointers from n's parent through n into l's subtree, (2) while backtracking from l's subtree to n

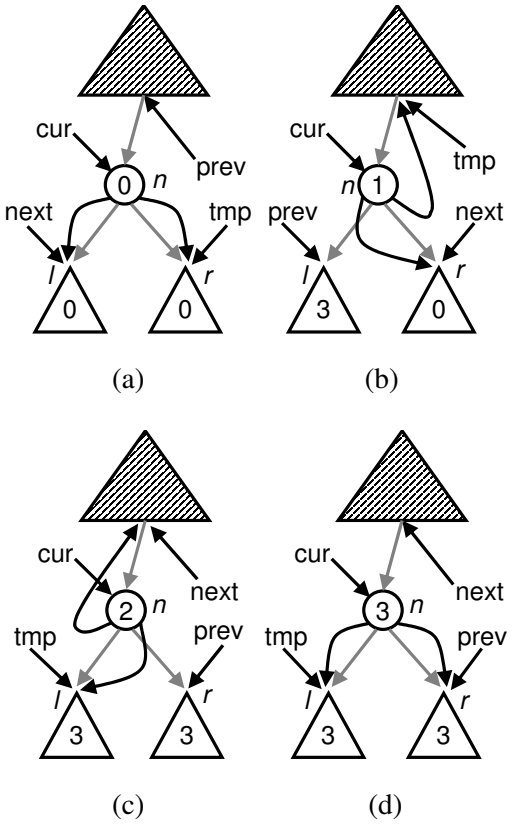


Figure 6.7 States of the subtree of n with `cur` pointing to n: (a) after the first execution of statement on line [10] of Fig. 6.6(b), n is in state 0; (b) after the second execution of statement on line [10] of Fig. 6.6(b), n is in state 1; (c) after the third execution of statement on line [10] of Fig. 6.6(b), n is in state 2; (d) after the third execution of statement on **line [14]** of Fig. 6.6(b), n is in state 3. Grey edges represent the original values of the `left` and `right` fields.

and then traversing r's subtree, and (3) while backtracking from r's subtree through n to n's parent in the original tree.

Fig. 6.8 depicts the states of the tree nodes that are not in the subtree pointed to by `cur`. All ancestors (in the original tree) of `cur`'s target are in state 1 or 2, indicating that the left (1) or right (2), subtree is currently being traversed. If `cur`'s target lies in the left subtree of an ancestor, then that ancestor must be in state 1, otherwise it must be in state 2. The triangular shapes at left represent all nodes that occur earlier than `cur`'s target in an in-order traversal of the tree. For each of these nodes there exists an ancestor of `cur`'s target, such that the node is in the left subtree of the ancestor, and `cur`'s target is in the right subtree of the ancestor. All nodes in that category are in state 3; they have been visited three times, and their `left` and `right` pointers have been reset to their original values. The triangular shapes at right represent all nodes that occur later than `cur`'s target in an in-order traversal of the tree. For each of these nodes there exists an ancestor of `cur`'s target, such that the node is in the right subtree of the ancestor, and `cur`'s target is in the left subtree of the ancestor. All nodes in that category are in state 0; they have not been visited, and their `left` and `right` pointers still have their original values.

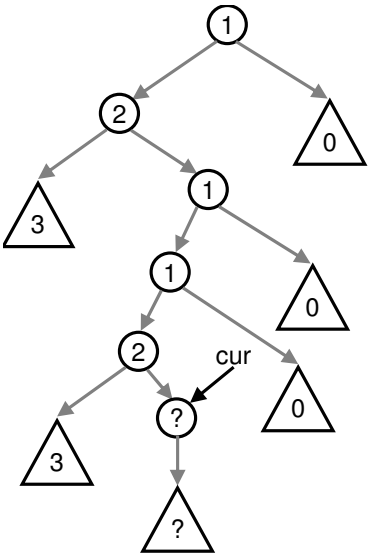


Figure 6.8 States of tree nodes that are outside of the subtree pointed to by `cur`. (Grey edges represent the original values of the `left` and `right` fields.)

6.4 A Shape Abstraction for Verifying DSW

Consider the problem of establishing that the version of the Deutsch-Schorr-Waite algorithm shown in Fig. 6.6(b) is partially correct. This is an assertion that compares the state of a store at the end of the procedure with its state at the start.

Partial correctness of DSW means (i) the tree produced at exit must be identical to the input tree, and (ii) every node must be visited. We will come back to property (ii) when we discuss the

total correctness of DSW in Sect. 6.5. Property (i) can be specified as follows:

$$\forall v_1, v_2 \colon \mathit{left}(v_1, v_2) \Leftrightarrow \mathit{left}^0(v_1, v_2) \tag{6.5}$$

$$\forall v_1, v_2 \colon \mathit{right}(v_1, v_2) \Leftrightarrow \mathit{right}^0(v_1, v_2), \tag{6.6}$$

where left^0 and right^0 denote the initial values of relations left and right, respectively. Additionally, a correct traversal routine must neither lose nodes of the input tree, nor gain new ones. However, this property is implied by properties (6.5) and (6.6).

The challenge is that the abstraction has to track the "unintended" use of pointers for stack simulation with sufficient precision to verify that at the end of the algorithm their correct usage has been reestablished. Canonical abstraction with just the properties listed in Figs. 6.2 and 6.3 is an insufficiently precise abstraction to demonstrate that the tree's edges are restored.

The key relations for establishing properties (6.5) and (6.6) at the end of the program are those that capture the relationships of pointers that arise between tree nodes during the traversal. The following set of unary relations capture properties of nodes in state 0 (before any changes to the nodes' `left` and `right` pointers) or state 3 (after the nodes' `left` and `right` pointer values have been restored):

$$eq_{l,l^0}(v_1) \stackrel{\text{def}}{=} \forall v_2 \colon \mathit{left}(v_1, v_2) \Leftrightarrow \mathit{left}^0(v_1, v_2) \tag{6.7}$$

$$eq_{r,r^0}(v_1) \stackrel{\text{def}}{=} \forall v_2 \colon \mathit{right}(v_1, v_2) \Leftrightarrow \mathit{right}^0(v_1, v_2) \tag{6.8}$$

Unary relations $eq_{l,l^0}(v_1)$ and $eq_{r,r^0}(v_1)$ distinguish individuals that represent tree nodes whose `left`, respectively `right`, pointers have their initial values. We can now use $\forall v \colon eq_{l,l^0}(v)$ in place of Formula (6.5) and $\forall v \colon eq_{r,r^0}(v)$ in place of Formula (6.6) when asserting the partial correctness of DSW.

The following set of unary relations capture properties of nodes in state 1, after one visit to those nodes, i.e., one rotation of the `left` and `right` pointers:

$$eq_{l,r^0}(v_1) \stackrel{\text{def}}{=} \forall v_2 \colon \mathit{left}(v_1, v_2) \Leftrightarrow \mathit{right}^0(v_1, v_2) \tag{6.9}$$

$$re_{r,l^0}(v_1) \stackrel{\text{def}}{=} \forall v_2 \colon \mathit{right}(v_1, v_2) \Leftrightarrow \mathit{left}^0(v_2, v_1) \tag{6.10}$$

$$re_{r,r^0}(v_1) \stackrel{\text{def}}{=} \forall v_2 \colon \mathit{right}(v_1, v_2) \Leftrightarrow \mathit{right}^0(v_2, v_1) \tag{6.11}$$

Unary relation $eq_{l,r^0}(v_1)$ distinguishes individuals that represent tree nodes whose `left` field points to their right (in the input tree) subtree. Unary relations $re_{r,l^0}(v_1)$ and $re_{r,r^0}(v_1)$ (re is a mnemonic for *reverse*) distinguish individuals that represent tree nodes n whose `right` fields point to their parents in the input tree (assuming that n is the left child in the case of $re_{r,l^0}(v_1)$ and right child, otherwise).

The following set of unary relations capture properties of nodes in state 2, after two visits to those nodes, i.e., two rotations of the `left` and `right` pointers:

$$eq_{r,l^0}(v_1) \stackrel{\text{def}}{=} \forall v_2 \colon right(v_1, v_2) \Leftrightarrow left^0(v_1, v_2) \tag{6.12}$$

$$re_{l,l^0}(v_1) \stackrel{\text{def}}{=} \forall v_2 \colon left(v_1, v_2) \Leftrightarrow left^0(v_2, v_1) \tag{6.13}$$

$$re_{l,r^0}(v_1) \stackrel{\text{def}}{=} \forall v_2 \colon left(v_1, v_2) \Leftrightarrow right^0(v_2, v_1) \tag{6.14}$$

Unary relation $eq_{r,l^0}(v_1)$ distinguishes individuals that represent tree nodes whose `right` field points to their left (in the input tree) subtree. Unary relations $re_{l,l^0}(v_1)$ and $re_{l,r^0}(v_1)$ distinguish individuals that represent tree nodes n whose `left` fields point to their parents in the input tree (assuming that n is the left child in the case of $re_{l,l^0}(v_1)$ and right child, otherwise).

Let us give the intuition behind the use of the relations defined by Formulas (6.7)–(6.14) for the partial-correctness verification of DSW, which involves establishing that all `left` and `right` pointers have their initial values at the end of DSW.

These relations maintain the relationship between the current and the original values of `left` and `right` pointers. Prior to the first rotation of pointers for node n, n has entries 1 for the state-0 relations (Formulas (6.7) and (6.8)), which say that there has been no change from n's starting pointer values. These entries allow the analysis to conclude that after the current iteration's rotation of n's pointers, n should have entry 1 for state-1 relations, Formula (6.9) and Formulas (6.10) or (6.11). Similarly, the 1 entries for the state-1 relations for node n help establish the 1 entries for its state-2 relations (Formula (6.12) and Formulas (6.13) or (6.14)) after the second rotation of n's pointers. Finally, the 1 entries for the state-2 relations for node n help establish the 1 entries for its state-3 relations Formulas (6.7) and (6.8) after the third rotation of n's pointers.

In our initial attempt to establish the partial correctness of DSW, we added all relations of Formulas (6.7)–(6.14) to the set of abstraction relations, $\mathcal{A}$. This attempt failed (we terminated the analysis after several days of computation) because of the vast abstract state space that needed to be explored. To pare down the abstract state space, we observed that not all node distinctions introduced by the relations of Formulas (6.7)–(6.14) were necessary. For instance, note that any leaf node in state 0 or state 3 satisfies (among other relations) Formula (6.9), which defines eq_{l,r^0}—nominally a state-1 relation—because it has no outgoing `left` or `right` pointers, while an internal tree node in state 0 or state 3 does not satisfy it. As a result, eq_{l,r^0} prevents canonical abstraction from summarizing a leaf node in state 0 or 3 with an internal node in one of those states. The resulting abstraction has a larger-than-necessary state space because we only need to ensure that tree nodes in state 1 have their `left` field pointing to their original right subtree, i.e., have the property defined by the relation eq_{l,r^0}.

To remove such unnecessary distinctions, we introduce the concept of a *state-dependent* abstraction. The first component of such an abstraction is a collection of unary core *state relations*, $state_0(v)$, $state_1(v)$, $state_2(v)$, and $state_3(v)$.[4] Every time the rotation of `left` and `right` pointers of the tree node pointed to by `cur` is completed (after line [14] of Fig. 6.6(b)), the node's state is changed to the next state. (The state relations carry no semantics with respect to the pointer values of nodes; they simply record the "visit counts" for each node.) As the second component of the abstraction, we introduce state-relation-guarded versions of the relations of Formulas (6.7)–(6.14):

$$s_0\_eq_{l,l^0}(v_1) \stackrel{\text{def}}{=} state_0(v_1) \wedge eq_{l,l^0}(v_1) \tag{6.15}$$

$$s_0\_eq_{r,r^0}(v_1) \stackrel{\text{def}}{=} state_0(v_1) \wedge eq_{r,r^0}(v_1) \tag{6.16}$$

$$s_1\_eq_{l,r^0}(v_1) \stackrel{\text{def}}{=} state_1(v_1) \wedge eq_{l,r^0}(v_1) \tag{6.17}$$

$$s_1\_re_{r,l^0}(v_1) \stackrel{\text{def}}{=} state_1(v_1) \wedge re_{r,l^0}(v_1) \tag{6.18}$$

$$s_1\_re_{r,r^0}(v_1) \stackrel{\text{def}}{=} state_1(v_1) \wedge re_{r,r^0}(v_1) \tag{6.19}$$

$$s_2\_eq_{r,l^0}(v_1) \stackrel{\text{def}}{=} state_2(v_1) \wedge eq_{r,l^0}(v_1) \tag{6.20}$$

$$s_2\_re_{l,l^0}(v_1) \stackrel{\text{def}}{=} state_2(v_1) \wedge re_{l,l^0}(v_1) \tag{6.21}$$

[4]The state relations are *not* added to the set of abstraction relations, $\mathcal{A}$.

$$s_2\_re_{l,r^0}(v_1) \stackrel{\text{def}}{=} state_2(v_1) \wedge re_{l,r^0}(v_1) \tag{6.22}$$

$$s_3\_eq_{l,l^0}(v_1) \stackrel{\text{def}}{=} state_3(v_1) \wedge eq_{l,l^0}(v_1) \tag{6.23}$$

$$s_3\_eq_{r,r^0}(v_1) \stackrel{\text{def}}{=} state_3(v_1) \wedge eq_{r,r^0}(v_1) \tag{6.24}$$

We replace the relations of Formulas (6.7)–(6.14) in the set of abstraction relations, $\mathcal{A}$, with Formulas (6.15)–(6.24). The resulting abstraction allows the grouping of nodes that have different values for the relation eq_{l,r^0}, for example, as long as these nodes are not in state 1.

6.5 Establishing that DSW Terminates

We established that DSW terminates using the unary state relations of Sect. 6.4 via a simple progress monitor, which we describe below.

For each state relation s, we create a copy of s, which is used to save the values of relation s at the start of the currently-processed loop iteration (after line [8] of Fig. 6.6(b)). We give the new relations the superscript lh to indicate that they hold the *loop-head* values. The first abstract operation of each iteration of the loop takes a snapshot of the current states of nodes: $state_i^{lh}(v) \leftarrow state_i(v)$, for each $i \in [0..3]$ and each assignment of v to an individual in the abstract structure being processed. Additionally, it asserts that `cur` does not point to a tree node in state 3 at the head of the loop.

The last operation of every loop iteration performs a progress test by asserting the following formula:

$$\begin{aligned} &\exists\, v\colon \ \big(state_0^{lh}(v) \wedge state_1(v) \vee state_1^{lh}(v) \wedge state_2(v) \vee state_2^{lh}(v) \wedge state_3(v)\big) \\ &\quad \bigwedge \forall\, v_1 \neq v\colon \ \bigwedge_{i\in[0..3]} \big(state_i^{lh}(v_1) \Leftrightarrow state_i(v_1)\big) \end{aligned}$$

The assertion ensures that one node's state makes forward progress (the first line of the assertion) and that no other node changes state (the second line of the assertion).

Together with the assertion that `cur` does not point to a tree node in state 3 at the start of the loop, the above progress monitor establishes that each tree node is visited at most three times, thus establishing that the algorithm terminates. As we show in the next section, in each structure collected by the analysis of DSW at the exit point every non-sentinel node is in state 3. Hence,

the analysis also establishes that every tree node is visited exactly three times, thus establishing property (ii) of partial correctness.

6.6 Experimental Evaluation

We applied TVLA to the DSW algorithm shown in Fig. 6.6(b) and analyzed it using the abstraction defined in Sect. 6.4. As input for the algorithm, we supplied the 3-valued structure $S_{6.9}$ shown in Fig. 6.9, which is essentially the structure $S_{6.5}$ from Fig. 6.5 refined with values for relations introduced in Sect. 6.4. Additionally, $S_{6.9}$ contains a special *sentinel* node that is not part of the input tree; it is referenced by program variable `SENTINEL`. In Fig. 6.9, as well as Fig. 6.10, history relations (e.g., $left^0$ and $right^0$) have been omitted to reduce clutter. Their values are identical to the values of their active counterparts. We have also omitted the values for state-1 and state-2 relations eq_{l,r^0}, re_{r,l^0}, re_{r,r^0}, eq_{r,l^0}, re_{l,l^0}, and re_{l,r^0}. They have value $1/2$ for the non-sentinel nodes of both figures and value 1 for the sentinel nodes. Because we are performing tree-specific shape analysis, both figures only represent concrete structures that satisfy the treeness integrity constraints (see Sect. 6.2).

Fig. 6.10 shows the unique structure $S_{6.10}$ collected by the analysis at the exit node. The definite 1 values for relations eq_{l,l^0} and eq_{r,r^0} (defined by Formulas (6.7) and (6.8)) for each individual of $S_{6.10}$ establish that the outgoing `left` and `right` pointers of every tree node are restored, thus establishing partial correctness property (i), i.e., that the tree produced at exit is identical to the input tree. The absence of violations of the progress monitor defined in Sect. 6.5 establishes that DSW terminates. The fact that every non-sentinel node is in state 3 establishes that every tree node is visited (partial correctness property (ii)).

The analysis took just under nine hours on a 3GHz Linux PC and used 150 MB of memory. While the authors have a number of ideas for performance optimizations for the research system, the main goal was to demonstrate the feasibility of automatic symbolic exploration of heap-manipulating programs with vast (abstract) state spaces.

The cost of verifying that DSW terminates is negligible (when compared to the cost that DSW is partially correct) because the progress monitor does not increase the size of the reachable state

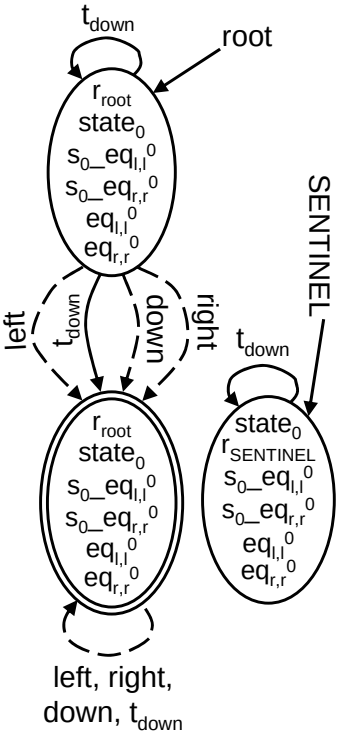


Figure 6.9 A 3-valued structure $S_{6.9}$ that represents all trees of size 2 or more

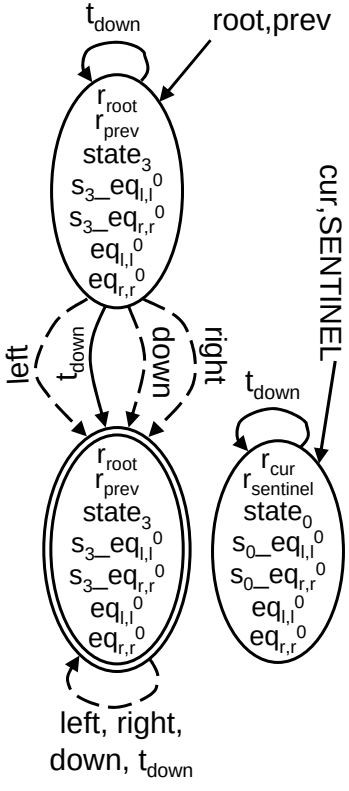


Figure 6.10 A 3-valued structure $S_{6.10}$ collected at exit of DSW

space. The number of distinct abstract structures that were collected at all program points exceeded 80,000. The number of structures at some program points exceeded 11,000. This number is not surprising, if we consider that some of these structures contained 15 individuals. (At intermediate steps, the analysis explored abstracts structures with up to 21 individuals!) However, 80,000 is well below the limit imposed by the number of distinct 3-valued structures, $2^{2^{20}}$, which represents the number of subsets of individuals with every possible vector of unary abstraction-relation values. (There are 20 unary abstraction relations: pointer relations $x(v)$ and reachability relations $r_x(v)$ for each of the five pointer-valued program variables, as well as ten relations of Formulas (6.15)–(6.24).) Fig. 6.11 shows a sample abstract structure $S_{6.11}$ that arises before line [11] of Fig. 6.6(b). In $S_{6.11}$, as in all other structures that arise at that point, the state relations and state-relation-guarded relations defined by Formulas (6.15)–(6.24), have precise values for all individuals.

In summary, our experiment showed that, using the abstraction defined in Sect. 6.4, an automatic analysis can maintain enough precision to identify sufficient invariants to demonstrate both partial correctness and termination of DSW.

6.7 Discussion and Future Work

The analysis carried out by TVLA performs fully-automatic state-space exploration. However, one has to bring to bear some expertise in specifying TVLA analyses. The concept of tree-specific shape analysis (see Sect. 6.2) is of general utility. It can be reused for any analysis in which all input, output, and intermediate data structures are trees. The instrumentation relations defined by Formulas (6.9)–(6.14), which capture pointer relationships of tree nodes, and core state relations $state_0(v), \ldots, state_3(v)$, which are used to control the precision of the abstraction, are specific to the problem of verifying the total correctness of DSW.

A key difference between our approach and theorem-prover-based approaches is that we do not need to specify loop invariants. Instead, we need to specify a collection of node distinctions (or node relationships), such as the relations $eq_{l,r^0}(v_1)$ and $re_{r,l^0}(v_1)$ of Formulas (6.9)–(6.14); these allow the node distinctions specified to be observable by the analysis. Given the appropriate node distinctions, abstract interpretation automatically infers the invariants satisfied by the program.

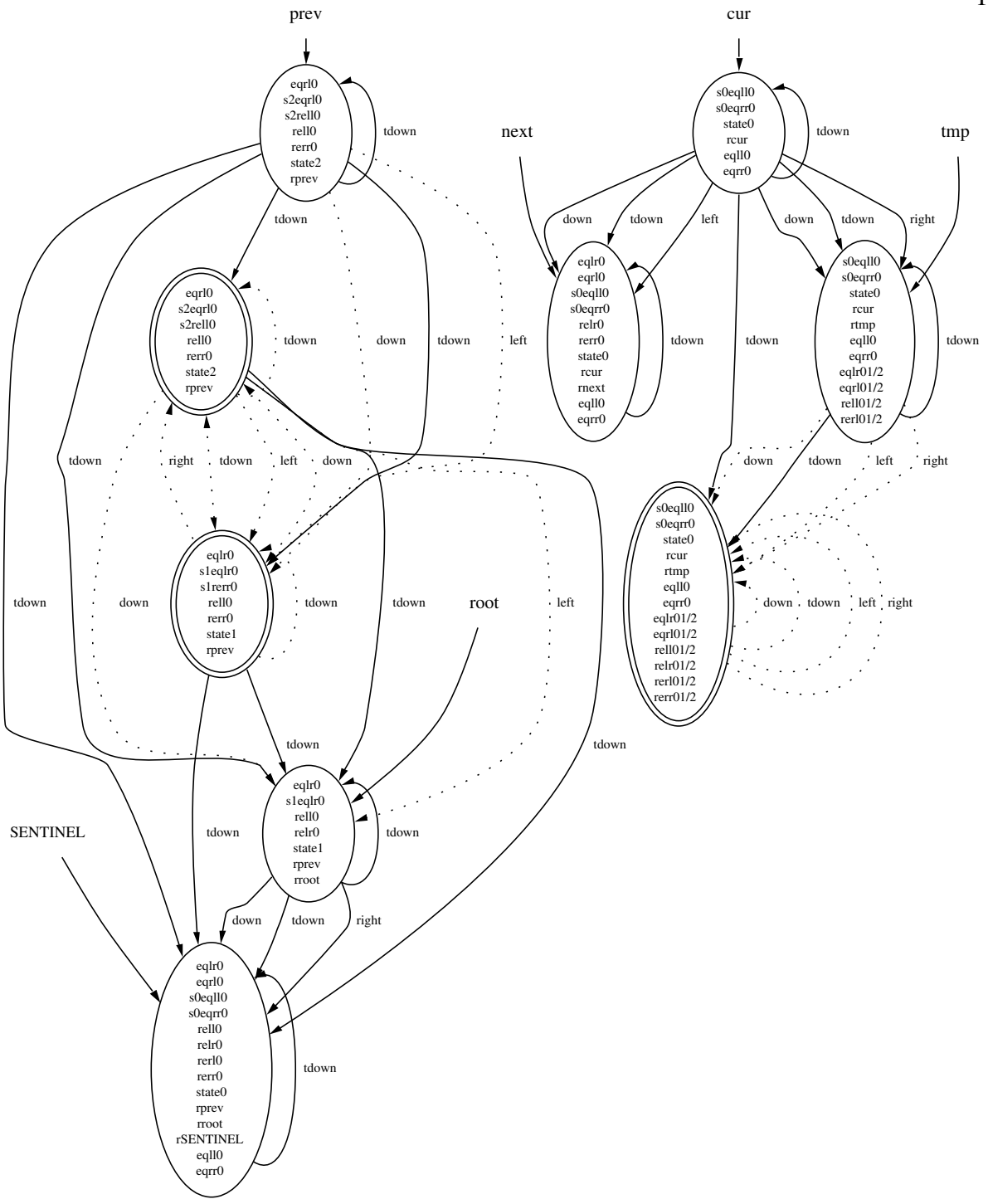


Figure 6.11 A 3-valued structure $S_{6.11}$ that arises prior to the first rotation of pointers of the node n pointed to by `cur` (before line [11] of Fig. 6.6(b)). History relations have been omitted from the figure. Initially, node n was the right child of the node pointed to by `prev`. The latter node is now the root of a tree with leaf `SENTINEL` (the original root is the parent of `SENTINEL`). No nodes in n's subtree have been visited; that subtree has not been modified from its initial state.

In Chapter 5, we showed how a machine-learning technique can be used to identify key instrumentation relations automatically. In the future, we would like to see if it can be used to identify the key relations for verifying DSW, namely the relations of Formulas (6.9)–(6.14).

Although the instrumentation relations introduced in Sect. 6.4 are tailored for establishing the correctness of DSW, the concept of state-dependent abstractions is likely to be of general utility. In fact, simpler versions of state-dependent abstractions have arisen in past work. For example, the unary relation $inOrder$ was used to establish the partial correctness of sorting [53]. The state-dependent abstractions defined in this chapter are prepared to deal with more than just two states (initial and final, as is the case for the relation $inOrder$), and use the value of the state as a guard to reduce the number of distinct properties recorded for individuals, thereby reducing the size of the (abstract) state space that is explored.

There is an interesting analogy between the explicit state-tracking that the original DSW algorithm performs via the `mark` and `tag` bits, and the state relations of our abstraction. (In some sense, the state relations introduced for purposes of analysis impose a DSW-like view of the world to track the actions of the Lindstrom variant of the algorithm.)

While we chose to apply a transformation that ensures that the algorithm maintains treeness (transformation 3 of Sect. 6.3), it is possible to verify the unmodified algorithm (Fig. 6.6(a)) by introducing the following instrumentation relation:

$$isLocallyShared(v) \stackrel{\text{def}}{=} \exists\, v_1 :\, left(v_1, v) \wedge right(v_1, v)$$

Relation $isLocallyShared$ (which has value 0 for all nodes in the input 3-valued structure, indicating that the input is a valid binary tree) allows us to relax the restriction on sharing by tracking where sharing occurs rather than requiring its absence. To be applicable to the version of the algorithm that does not include transformation 3, the tree-specific shape analysis of Sect. 6.2 can be generalized to handle the limited class of DAGs that arise in lines [9]–[10] of Fig. 6.6(a) as follows:

1. The precondition for the absence of sharing (Formula (6.2)) would be removed.

2. The integrity constraints that forbid structures that contain sharing would be modified to include an $isLocallyShared$ guard to permit the kind of local sharing that arises in Fig. 6.6(a).

E.g., Constraint (6.4) becomes:

$$\forall\, v_1, v_2 \colon (\exists\, v \colon \neg \mathit{isLocallyShared}(v) \wedge \mathit{down}(v_1, v) \wedge \mathit{down}(v_2, v)) \quad \Rightarrow \quad v_1 = v_2.$$

The DSW algorithm shown in Fig. 6.6(b) (as well as the algorithm shown in Fig. 6.6(a)) does not work correctly when applied to a data structure that contains a cycle: the traversal terminates prematurely and not all of the edges are properly restored. However, the algorithm works correctly when applied to a DAG: a node n with k paths from the root to n is visited $3k$ times, rather than 3 times. (Note, however, that k can be exponential in the size of the graph.) Given a bound on k, we may be able to verify the correctness of DSW for DAGs, if we relax the restriction on sharing and introduce $3k$ state relations and the corresponding state-relation-guarded relations. However, unless k is very small it is not likely that the reachable state space can be explored with our computing resources. In the general case, in which the input is a DAG with no bound on k, the partial-correctness result can be obtained by having the state relations of nodes wrap around: a visit to a node in state 3 results in changing the node's state to 1. While this change would be sufficient to establish that the outgoing `left` and `right` pointers of every DAG node are restored and that every node is visited, the analysis would no longer be able to establish termination using the simple progress monitor of Sect. 6.5.

In practice, one would rarely be interested in using such an algorithm to traverse a DAG because of the potentially exponential cost. In most applications, one is likely to want to process each node once (e.g., in depth-first order) and visit each node a constant number of times. This can be achieved by equipping the nodes with two bits to record the visit count (a number from 0 to 3). All nodes reachable from a node with visit count 3 must have been visited three times. If `cur` is set to point to a node with visit count 3, the direction of the traversal can be reversed by swapping the values of `cur` and `prev`, thus terminating the exploration of the node's subgraph. By relaxing the restriction on sharing, it should be possible to verify the total correctness of the modified algorithm.

When DSW is used to traverse a DAG, the algorithm may temporarily create a cycle in the data structure. (This happens when the input DAG contains forward edges, for example.) In this case, the techniques of Sect. 6.2, as well as the techniques of Sect. 3.3.2, which we use for the

maintenance of the relation t_{down}, do not apply. We believe that the general approach taken in Sect. 3.3.3 for the maintenance of reachability in possibly-cyclic deterministic graphs can be used to address this limitation.

6.8 Related Work[5]

The general form of the Deutsch-Schorr-Waite algorithm works correctly for arbitrary graphs [87]. (Unlike the algorithm we used in our work, which was taken from [56], the general form is not constant-space because it uses mark and tag bits.) We divide the discussion of related work according to the kind of data structures to which the analyzed algorithm can be applied.

6.8.1 DSW on Arbitrary Graphs.

The first formal proofs of the partial correctness of DSW were performed manually by Morris [67] and Topor [92]. In [90], Suzuki automated some steps of the partial-correctness verification of the algorithm by introducing decision procedures that could handle heap-manipulating programs. More recently, Bornat used the Jape proof editor [10] to construct a partial-correctness proof of DSW [9]. The resulting proof used 152 pages [8].

Our automated approach provides the obvious benefit of disposing with the need to provide manual proofs, which require significant investments of time and expertise. However, even in the presence of a powerful theorem prover, proof-based approaches rely on the user to provide loop invariants that are sufficient to establish the property being verified. For instance, the properties of nodes and their subtrees that are described in Sect. 6.3 (see Figs. 6.7 and 6.8 and the corresponding text) would have to be specified as loop invariants. As discussed in Sect. 6.7, our obligation is simpler: we have to specify instrumentation relations that act as *ingredients* for a loop invariant; the analysis automatically synthesizes a loop invariant—in the form of a collection of 3-valued structures that overapproximate the set of concrete structures that actually arise—by means of state-space exploration.

[5]The discussion of [67, 90, 92] relies on what is reported in [62, 99].

Yang [98] and Mehta and Nipkow [62] gave manually-constructed, but machine-checkable, proofs of the partial correctness of DSW. The two approaches share the goal of making formal reasoning about heap-manipulating programs more natural. The former approach uses the logic of Bunched Implications [40] (a precursor formalism to Separation Logic [81]), which permits the user to reason with Hoare triples in the presence of complicated aliasing relationships. The latter approach uses Isabelle/HOL to construct formal proofs that are human-readable. These approaches improve the usability of proof-based techniques. However, they still lack the automation of our approach.

6.8.2 DSW on Trees and DAGs.

Yelowitz and Duncan were the first to present a termination argument for the Deutsch-Schorr-Waite algorithm [99]. They analyzed Knuth's version of the algorithm [46], which uses tag bits but does not work correctly for graphs that contain a cycle. It does, however, work for DAGs, as does the version we used, taken from [56]. The termination argument involved the use of program invariants to prove bounds on the number of executions of statements in the loop. In Sect. 6.5, we showed how to use the *state relations* defined in Sect. 6.4 in a simple progress monitor for the algorithm's loop to establish that DSW terminates (on trees). As was the case for partial correctness, our task is reduced to establishing appropriate distinctions between nodes. Given the state relations, the complete state-space exploration shows no violation of the progress monitor and establishes a bound (namely, three) on the number of visits to each tree node; consequently, the algorithm must terminate.

Several previous papers reported on automatic verification of weaker properties of the Deutsch-Schorr-Waite algorithm, namely that the algorithm has no unsafe pointer operations or memory leaks, and that the data structure produced at the end is, in fact, a binary tree [51, 59, 79]. The authors first established these properties in [79]. ([59] contains a typo stating that that work establishes partial correctness; however, [59] reused the TVLA specification from [79], and establishes the same properties as [79].) Finally, [51] extended the framework of [84] with grammars, which provide convenient syntactic sugar for expressing shape properties of data structures. That work

relied on the use of grammars, instead of instrumentation relations, to express tree properties and the absence of memory leaks.

Chapter 7

Semantic Minimization of 3-Valued Propositional Formulas

While studying the question of precision in analyses based on 3-valued logic, we encountered a non-standard logic-minimization problem that arises in 3-valued propositional logic. The present chapter provides a formalization of the "semantic-minimization" problem and several methods for creating a "semantically minimal" formula.

Our interest in this problem is motivated by the possibility of obtaining better answers in applications that use 3-valued logic. An answer of 0 or 1 provides precise (definite) information; an answer of $1/2$ provides imprecise (indefinite) information. By replacing a formula φ with a semantically-minimal equivalent ψ, we may improve the precision of the answers obtained.

The chapter is organized as follows: Sect. 7.1 introduces some terminology and notation. Sect. 7.2 defines the problem of semantic minimization for 3-valued propositional logic. Sect. 7.3 presents a couple of different methods for performing semantic minimization. Sect. 7.4 defines a semantic-minimization algorithm that, for efficiency, uses Binary Decision Diagrams in certain stages. Sect. 7.5 discusses related work. Several proofs appear in App. B.

7.1 Terminology and Notation

In this section, we define a standard 2-valued propositional logic, together with a related 3-valued propositional logic with a semantics due to Kleene [45].

7.1.1 2-Valued Propositional Logic

We write propositional formulas over a set of propositional variables $\mathcal{V}$ using the propositional constants **0** and **1**, the unary connective $\neg$, and the binary connectives $\wedge$ and $\vee$. We also make use of conditional expressions, for which we adopt a C-like syntax: $\varphi_1 \,?\, \varphi_2 : \varphi_3$.[1]

For brevity, we will sometimes use juxtaposition in place of $\wedge$, and use an overbar to denote negation; e.g., $(\neg x \wedge y \wedge z) \vee (x \wedge \neg y \wedge \neg z)$ may also be written as $\overline{x}yz \vee x\overline{y}\,\overline{z}$.

Propositional variables and negations of propositional variables will be referred to collectively as **literals**.

The (2-valued truth-functional) semantics for propositional logic is defined in the standard way:

Definition 7.1.1 An **assignment** a is a (finite) function in $\mathcal{V} \to \{0, 1\}$. Given a formula φ over the propositional variables $x_1, \ldots, x_n$ and an assignment a that is defined on (at least) $x_1, \ldots, x_n$, the **2-valued truth-functional meaning** of φ with respect to a, denoted by $[\![\varphi]\!](a)$, is the truth value in $\{0, 1\}$ defined inductively as follows:

$$
\begin{aligned}
[\![\mathbf{0}]\!](a) &= 0 & [\![x_i]\!](a) &= a(x_i) \\
[\![\mathbf{1}]\!](a) &= 1 & [\![\neg\varphi]\!](a) &= 1 - [\![\varphi]\!](a) \\
[\![\varphi_1 \wedge \varphi_2]\!](a) &= \min([\![\varphi_1]\!](a), [\![\varphi_2]\!](a)) \\
[\![\varphi_1 \vee \varphi_2]\!](a) &= \max([\![\varphi_1]\!](a), [\![\varphi_2]\!](a)) \\
[\![\varphi_1 \,?\, \varphi_2 : \varphi_3]\!](a) &= [\![(\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \varphi_3)]\!](a)
\end{aligned}
$$

We say that a **satisfies** φ, denoted by $a \models \varphi$, iff $[\![\varphi]\!](a) = 1$. □

Later on, it will be useful to be able to indicate that various semantically equivalent formulas are syntactically related in certain ways. We use $\equiv$ to denote syntactic equality between formulas, up to rearrangements of conjuncts and disjuncts; we use $\equiv_{\text{DM}}$ to denote $\equiv$, extended with applications

[1] For now, one can think of $\varphi_1 \,?\, \varphi_2 : \varphi_3$ as a shorthand for $(\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \varphi_3)$ (see Defn. 7.1.1). Later on, for technical reasons, we will consider it to be a shorthand for $(\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \varphi_3) \vee (\varphi_2 \wedge \varphi_3)$ (see Defn. 7.1.3 and Ex. 7.4.9).

of De Morgan's laws and introductions/cancellations of double negations. For instance,

$$\neg((x \wedge \neg y) \vee (\neg x \wedge z)) \equiv \neg((z \wedge \neg x) \vee (\neg y \wedge x))$$

$$\neg((x \wedge \neg y) \vee (\neg x \wedge z)) \not\equiv (\neg z \vee x) \wedge (y \wedge \neg x)$$

$$\neg((x \wedge \neg y) \vee (\neg x \wedge z)) \equiv_{\text{DM}} (\neg z \vee x) \wedge (y \wedge \neg x)$$

$$\neg((x \wedge \neg y) \vee (\neg x \wedge z)) \not\equiv_{\text{DM}} (\neg z \wedge y) \vee (\neg z \wedge \neg x) \vee (x \wedge y) \vee (x \wedge \neg x)$$

All four of the formulas used above have the same 2-valued truth-functional meaning. We reserve "=" for semantic equality (e.g., $[\![\neg((x \wedge \neg y) \vee (\neg x \wedge z))]\!] = [\![(\neg z \wedge y) \vee (\neg z \wedge \neg x) \vee (x \wedge y) \vee (x \wedge \neg x)]\!]$).

7.1.2 3-Valued Propositional Logic

Moving now to 3-valued logic, the language of formulas that we work with is identical to that defined in Sect. 7.1.1, except that there is one additional propositional constant, $\mathbf{1/2}$. At the semantic level, a third truth value—$1/2$—is introduced to denote uncertainty. We say that the values 0 and 1 are **definite values** and that $1/2$ is an **indefinite value**, and define a partial order $\sqsubseteq$ on truth values to reflect their degree of definiteness (or **information content**): $l_1 \sqsubseteq l_2$ denotes that l_1 is at least as definite as l_2:

Definition 7.1.2 [Information Order]. For $l_1, l_2 \in \{0, 1/2, 1\}$, we define the **information order** on truth values as follows: $l_1 \sqsubseteq l_2$ iff $l_1 = l_2$ or $l_2 = 1/2$. We use $l_1 \sqsubset l_2$ when $l_1 \sqsubseteq l_2$ and $l_1 \neq l_2$. The symbol $\sqcup$ denotes the least-upper-bound operation with respect to $\sqsubseteq$:

$\sqcup$	0	$1/2$	1
0	0	$1/2$	$1/2$
$1/2$	$1/2$	$1/2$	$1/2$
1	$1/2$	$1/2$	1

□

We now generalize Defn. 7.1.1 to define the meaning of a formula with respect to a 3-valued assignment A. (Our convention will be to use lower-case letters for 2-valued assignments, and upper-case letters for 3-valued assignments.)

Definition 7.1.3 A **3-valued assignment** A is a (finite) function in $\mathcal{V} \to \{0, 1, 1/2\}$. Given a formula φ over the propositional variables $x_1, \ldots, x_n$ and an assignment A that is defined on (at least) $x_1, \ldots, x_n$, the **3-valued truth-functional meaning** of φ with respect to A, denoted by $[\![\varphi]\!](A)$, yields a truth value in $\{0, 1, 1/2\}$. The meaning of φ is defined inductively as in Defn. 7.1.1, with the following changes:

$$[\![\mathbf{1/2}]\!](A) = 1/2$$
$$[\![\varphi_1 \,?\, \varphi_2 : \varphi_3]\!](A) = [\![(\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \varphi_3) \vee (\varphi_2 \wedge \varphi_3)]\!](A)$$

We say that A **potentially satisfies** φ, denoted by $A \models \varphi$, iff $[\![\varphi]\!](A) \sqsupseteq 1$ (i.e., $[\![\varphi]\!](A) = 1/2$ or $[\![\varphi]\!](A) = 1$). $\square$

The 3-valued truth tables for the propositional operators are shown in Fig. 7.1.

In 2-valued logic, $(\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \varphi_3)$ and $(\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \varphi_3) \vee (\varphi_2 \wedge \varphi_3)$ yield equivalent definitions of the 2-valued truth-functional meaning of $\varphi_1 \,?\, \varphi_2 : \varphi_3$. In 3-valued logic, however, $(\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \varphi_3) \vee (\varphi_2 \wedge \varphi_3)$ yields a more precise semantics. In the truth table for "$v_1 \,?\, v_2 : v_3$", the sub-table for "$v_1 = 1/2$" (which can be obtained by evaluating $(\mathbf{1/2} \wedge v_2) \vee (\neg\mathbf{1/2} \wedge v_3) \vee (v_2 \wedge v_3)$) is identical to the truth table for "$v_2 \sqcup v_3$" (cf. Defn. 7.1.2). Consequently, $\mathbf{1/2} \,?\, v_2 : v_3$ yields a definite value when v_2 and v_3 are either both 0 or both 1. In particular, for the assignment $[v_2 \mapsto 1, v_3 \mapsto 1]$, $(\mathbf{1/2} \wedge v_2) \vee (\neg\mathbf{1/2} \wedge v_3)$ yields the indefinite value $1/2$, whereas $(\mathbf{1/2} \wedge v_2) \vee (\neg\mathbf{1/2} \wedge v_3) \vee (v_2 \wedge v_3)$ yields 1. (We will look at this from another vantage point later, in Ex. 7.4.9.) We will use $\sqcup$ as a binary connective for constructing formulas: $\varphi_1 \sqcup \varphi_2$ is a shorthand for $\mathbf{1/2} \,?\, \varphi_1 : \varphi_2$.

It should be noted that throughout the remainder of the chapter, the symbol $\models$ means the potential-satisfaction relation of Defn. 7.1.3, even when we are talking about a 2-valued assignment. For instance, $[p \mapsto 1] \models p \wedge \mathbf{1/2}$ because $[\![p \wedge \mathbf{1/2}]\!]([p \mapsto 1]) = 1/2$.

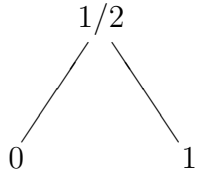
$\neg$	
0	1
1/2	1/2
1	0

$\wedge$	0	1/2	1
0	0	0	0
1/2	0	1/2	1/2
1	0	1/2	1

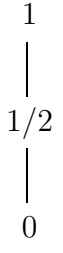
$\vee$	0	1/2	1
0	0	1/2	1
1/2	1/2	1/2	1
1	1	1	1

$v_1 \; ? \; v_2 : v_3$		v_3		
	v_2	0	1/2	1
$v_1 = 0$	0	0	1/2	1
	1/2	0	1/2	1
	1	0	1/2	1
$v_1 = 1/2$	0	0	1/2	1/2
	1/2	1/2	1/2	1/2
	1	1/2	1/2	1
$v_1 = 1$	0	0	0	0
	1/2	1/2	1/2	1/2
	1	1	1	1

Figure 7.1 The 3-valued truth tables for the propositional operators



(a) Information order



(b) Logical order

Figure 7.2 The semi-bilattice of 3-valued propositional logic

As shown in Fig. 7.2, the values 0, 1, and $1/2$ form a mathematical structure known as a semi-bilattice (see [30]). A semi-bilattice has two orderings: the **information order** and the **logical order**:

- The information order is the one defined in Defn. 7.1.2, which captures "(un)certainty".
- The logical order is the one used in Fig. 7.1: that is, $\wedge$ and $\vee$ are meet and join in the logical order (e.g., $1 \wedge 1/2 = 1/2$, $1 \vee 1/2 = 1$, $1/2 \wedge 0 = 0$, $1/2 \vee 0 = 1/2$, etc.).

A value that is "far enough up" in the logical order indicates "potential truth", and is called a **designated value**. We take $1/2$ and 1 as the designated values; thus, an assignment A potentially satisfies a formula when the formula's truth-functional meaning with respect to A is one of the designated values.

The information ordering on values is extended pointwise to an information ordering on assignments (also denoted by $\sqsubseteq$); e.g., $[p \mapsto 1, q \mapsto 0] \sqsubseteq [p \mapsto 1, q \mapsto 1/2]$, $[p \mapsto 1, q \mapsto 0] \sqsubseteq [p \mapsto 1/2, q \mapsto 0]$, $[p \mapsto 1, q \mapsto 0] \sqsubseteq [p \mapsto 1/2, q \mapsto 1/2]$, etc. When A does not contain any bindings of a propositional variable to the value 1/2, we say that A is **definite** (and usually write it with a lower-case a).

Kleene's 3-valued semantics is monotonic in the information order (cf. Fig. 7.1 and Defn. 7.1.3):

Lemma 7.1.4 Let φ be a formula, and let A and A' be two assignments such that $A \sqsubseteq A'$. Then $[\![\varphi]\!](A) \sqsubseteq [\![\varphi]\!](A')$. $\square$

Kleene's semantics retains a number of properties that are familiar from 2-valued logic, including De Morgan's laws and the ability to introduce/cancel double negations. For this reason, $\equiv$ and $\equiv_{\text{DM}}$ relate formulas that are semantically equivalent in 3-valued logic.

Lemma 7.1.4 provides a way to relate the 2-valued and 3-valued truth-functional meanings of formulas: the value obtained by evaluating any formula φ with respect to a 3-valued assignment A is always safe (i.e., greater than or equal to in the information order) compared to the value obtained by evaluating φ with respect to any 2-valued assignment $a \sqsubseteq A$. In particular,

- If $[\![\varphi]\!](A)$ yields a definite value, then $[\![\varphi]\!](a)$ must yield the same definite value.

- If $[\![\varphi]\!](A)$ yields $1/2$, then $[\![\varphi]\!](a)$ can be either 0 or 1.

We say that a 3-valued assignment A **represents** all 2-valued assignments $a \sqsubseteq A$. Another outlook on the way 2-valued and 3-valued assignments are related stems from the following corollary (which follows immediately from Lemma 7.1.4):

Corollary 7.1.5 Suppose that A represents a. If $a \models \varphi$, then $A \models \varphi$. $\square$

Thus, if we think of a propositional formula φ of 2-valued logic as a device for accepting a set S of 2-valued assignments, then when φ is considered as a formula of 3-valued logic, the *potential*-satisfaction relation corresponds to an implicit condition for accepting/rejecting an entire set of 2-valued assignments—those that are represented by a 3-valued assignment. Moreover, acceptance via the potential-satisfaction relation is *safe* with respect to the actual set of 2-valued assignments accepted by φ:

$$\{a \in \text{def. assignments} \mid a \models \varphi\} \subseteq \{a \text{ rep. by } A \mid A \models \varphi\}$$

This point of view is useful when 3-valued assignments are used as the space of abstract values in an abstract interpretation (e.g., see Chou's account of Symbolic Trajectory Evaluation in abstract-interpretation terms [14]). We will also adopt this viewpoint in Sect. 7.2.2 in order to justify our definition of the semantic-minimization problem.

7.2 The Semantic Minimization Problem

In Chapter 1, we observed that although the formula $\mathbf{1}$ is *equivalent* to $p \vee \neg p$ in 2-valued logic, in 3-valued logic, $\mathbf{1}$ is *better than* $p \vee \neg p$. This raises the question, "For any given φ, is there always a best formula?", which, in turn, raises the question, "What properties must a 'best' formula possess?"

7.2.1 Definition of the Minimization Problem

The concept of a "best formula" is formalized using the concept of a formula's "supervaluational meaning" [94]:

Definition 7.2.1 Given a formula φ and assignment A, the **3-valued supervaluational meaning** of φ with respect to A, denoted by $\langle\!\langle\varphi\rangle\!\rangle(A)$, is the truth value in $\{0, 1, 1/2\}$ defined by

$$\langle\!\langle\varphi\rangle\!\rangle(A) = \bigsqcup_{a \text{ rep. by } A} [\![\varphi]\!](a).$$

□

Definition 7.2.2 Given a propositional formula φ, we say that the formula ψ is a **semantically minimal variant** of φ iff, for all 3-valued assignments A, $[\![\psi]\!](A) = \langle\!\langle\varphi\rangle\!\rangle(A)$. The **semantic-minimization problem** for propositional logic is as follows:

> Given a propositional formula φ, find a formula ψ that is a semantically minimal variant of φ.

□

For instance, $\mathbf{1}$ is a semantically minimal variant of $p \vee \neg p$; in particular,

$$\begin{aligned}\langle\!\langle\varphi\rangle\!\rangle([p \mapsto 1/2]) &= \bigsqcup_{a \in \{[p \mapsto 0], [p \mapsto 1]\}} [\![\varphi]\!](a) \\ &= 1 \sqcup 1 \\ &= [\![\mathbf{1}]\!]([p \mapsto 1/2]).\end{aligned}$$

Similarly, $\mathbf{0}$ is a semantically minimal variant of $p \wedge \neg p$.

7.2.2 Justification of the Problem Definition

It is worthwhile to spend a few moments to consider why Defn. 7.2.2 is the appropriate definition of the semantic-minimization problem. Let us contrast Defn. 7.2.2 with a possible alternative definition:

Strawman Definition 7.2.3 Given a propositional formula φ, we say that ψ is a **semantically minimal variant** of φ iff for all 3-valued assignments A, $[\![\psi]\!](A) \sqsubseteq [\![\varphi]\!](A)$. □

The motivation behind this definition is that, by using ψ in place of φ, (i) we could sometimes obtain answers that are strictly more definite, and (ii) we could never obtain answers that are strictly less definite. The question we must ask, however, is whether we would obtain *acceptable* answers with Strawman Defn. 7.2.3.

Note that with Defn. 7.2.2, neither $\mathbf{0}$ nor $\mathbf{1}$ is a semantically minimal variant of $\mathbf{1/2}$. In contrast, with Strawman Defn. 7.2.3, the formulas $\mathbf{0}$ and $\mathbf{1}$ would both be semantically minimal variants of $\mathbf{1/2}$. The latter situation would get us into trouble because their meanings, $[\![\mathbf{0}]\!] = \lambda a.0$ and $[\![\mathbf{1}]\!] = \lambda a.1$, are in conflict; that is, with Strawman Defn. 7.2.3, the admissible ψ's are not all semantically equivalent in 2-valued logic. In contrast, with Defn. 7.2.2, the admissible ψ's are all semantically equivalent in 2-valued logic; by definition, they all have the meaning $\langle\!\langle\varphi\rangle\!\rangle$—the supervaluational meaning of φ.

An even better way to see that Strawman Defn. 7.2.3 is unsatisfactory is by considering how the two concepts of "semantically minimal variant" relate to the view of a formula as a device for accepting a set of assignments (cf. the discussion following Cor. 7.1.5).

Desideratum 7.2.4 [Better Acceptance Device I]. When we view a formula as a device for accepting a set of assignments, we would like for ψ to correspond to a **better** acceptance device than φ. That is, when applied to an assignment A, either 2-valued or 3-valued, ψ may yield a more precise acceptance condition than φ: in circumstances in which φ waffles (i.e., $[\![\varphi]\!](A) = 1/2$), ψ can either

- waffle itself (i.e., $[\![\psi]\!](A) = 1/2$)
- accept A (i.e., $[\![\psi]\!](A) = 1$)
- reject A (i.e., $[\![\psi]\!](A) = 0$)

However, ψ must always be safe with respect to the 2-valued assignments that φ accepts:

$$\{a \in \text{def. assignments} \mid a \models \varphi\} \subseteq \{a \text{ rep. by } A \mid A \models \psi\}. \tag{7.1}$$

Similarly, $\neg\psi$ must always be safe with respect to the 2-valued assignments that $\neg\varphi$ accepts:

$$\{a \in \text{def. assignments} \mid a \models \neg\varphi\} \subseteq \{a \text{ rep. by } A \mid A \models \neg\psi\}. \tag{7.2}$$

□

For instance, suppose that φ is the formula $\mathbf{1/2}$. Under Strawman Defn. 7.2.3, the formula $\mathbf{0}$ is an admissible ψ; however, Eqn. (7.1) does not hold:

$$\begin{aligned}\{a \in \text{def. assignments} \mid a \models \mathbf{1/2}\} &= \{a \in \text{def. assignments}\} \\ &\not\subseteq \emptyset \\ &= \{a \text{ rep. by } A \mid A \models \mathbf{0}\}.\end{aligned}$$

Similarly, under Strawman Defn. 7.2.3, the formula $\mathbf{1}$ is also an admissible ψ; however, Eqn. (7.2) does not hold:

$$\begin{aligned}\{a \in \text{def. assignments} \mid a \models \neg\mathbf{1/2}\} &= \{a \in \text{def. assignments}\} \\ &\not\subseteq \emptyset \\ &= \{a \text{ rep. by } A \mid A \models \neg\mathbf{1}\}.\end{aligned}$$

Under Defn. 7.2.2, $\mathbf{1/2}$ is an admissible ψ, but $\mathbf{0}$ and $\mathbf{1}$ are not; clearly, with $\varphi = \psi = \mathbf{1/2}$, Eqns. (7.1) and (7.2) both hold.

The notion of a "better acceptance device" can also be expressed in a different way, which nicely parallels the statement of Cor. 7.1.5, but with ψ in the consequent:[2]

Desideratum 7.2.5 [Better Acceptance Device II]. Let ψ be a semantically minimal variant of φ. Then, for every 3-valued assignment A and 2-valued assignment a such that A represents a, both of the following must hold:

1. If $a \models \varphi$, then $A \models \psi$.
2. If $a \models \neg\varphi$, then $A \models \neg\psi$.

□

In contrast to the unsatisfactory results obtained with Strawman Defn. 7.2.3, we have the following:

[2]The two properties in Desideratum 7.2.5 are parallel to (i) the property stated in Cor. 7.1.5, and (ii) the property stated in Cor. 7.1.5 with φ replaced by $\neg\varphi$.

Lemma 7.2.6 If "semantically minimal variant" means the concept defined in Defn. 7.2.2, then Desideratum 7.2.5 holds.

Proof: The desired properties can be restated as follows:

1. If $[\![\varphi]\!](a) \sqsupseteq 1$, then $[\![\psi]\!](A) \sqsupseteq 1$.
2. If $[\![\neg\varphi]\!](a) \sqsupseteq 1$, then $[\![\neg\psi]\!](A) \sqsupseteq 1$.

These are proved, respectively, as follows:

$$
\begin{aligned}
1. \quad [\![\psi]\!](A) &= \langle\!\langle\varphi\rangle\!\rangle(A) \\
&= \bigsqcup_{a \text{ rep. by } A} [\![\varphi]\!](a) \\
&\sqsupseteq 1 \\
2. \quad [\![\neg\psi]\!](A) &= 1 - [\![\psi]\!](A) \\
&= 1 - \langle\!\langle\varphi\rangle\!\rangle(A) \\
&= 1 - \bigsqcup_{a \text{ rep. by } A} [\![\varphi]\!](a) \\
&= \bigsqcup_{a \text{ rep. by } A} (1 - [\![\varphi]\!](a)) \\
&= \bigsqcup_{a \text{ rep. by } A} [\![\neg\varphi]\!](a) \\
&\sqsupseteq 1
\end{aligned}
$$

□

Henceforth, the term "semantically minimal variant" means the concept defined in Defn. 7.2.2.

Strawman Defn. 7.2.3 was motivated by the desire to obtain more precise answers by using ψ in place of φ. In fact, with Defn. 7.2.2, we do have such a property:

Lemma 7.2.7 If ψ is a semantically minimal variant of φ, then for all 3-valued assignments A, $[\![\psi]\!](A) \sqsubseteq [\![\varphi]\!](A)$.

Proof: For every semantically minimal variant ψ, we have, by the monotonicity of $[\![\varphi]\!]$ (i.e., Lemma 7.1.4),

$$\begin{aligned}[\![\psi]\!](A) &= \langle\!\langle\varphi\rangle\!\rangle(A) \\ &= \bigsqcup_{a \text{ rep. by } A} [\![\varphi]\!](a) \\ &\sqsubseteq [\![\varphi]\!](A).\end{aligned}$$

□

That the term "semantically minimal variant" is appropriate can be seen from:

Lemma 7.2.8 If ψ is a semantically minimal variant of φ, and φ' is any formula that agrees with φ on all 2-valued assignments, then for all 3-valued assignments A, $[\![\psi]\!](A) \sqsubseteq [\![\varphi']\!](A)$.

Proof: For every semantically minimal variant ψ, we have, by the monotonicity of $[\![\varphi']\!]$ (i.e., Lemma 7.1.4),

$$\begin{aligned}[\![\psi]\!](A) &= \langle\!\langle\varphi\rangle\!\rangle(A) \\ &= \bigsqcup_{a \text{ rep. by } A} [\![\varphi]\!](a) \\ &= \bigsqcup_{a \text{ rep. by } A} [\![\varphi']\!](a) \\ &\sqsubseteq [\![\varphi']\!](A).\end{aligned}$$

□

7.3 An Algorithm for Semantic Minimization

We now return to the question "For any given φ, is there always a best formula?", and answer it in the affirmative. (More precisely, for each formula φ, there is an equivalence class of best formulas, which may or may not contain φ itself.) Our solution relies on a result, due to Blamey [5–7], that relates Boolean functions and 3-valued propositional formulas. The result can be stated in a couple of different forms; these yield different methods for creating a best formula. (In Sect. 7.4,

we will focus on a special case of Blamey's result; the latter variant will allow us to define a more efficient minimization algorithm.)

7.3.1 Realization of Monotonic Boolean Functions Via Formulas

In this section, we review a theorem, due to Blamey, that relates monotonic Boolean functions and 3-valued propositional formulas. We say that a formula φ **realizes** a function f iff $[\![\varphi]\!] = f$. Blamey's theorem states that, for every 3-valued function $f : \{0, 1, 1/2\}^n \to \{0, 1, 1/2\}$ that is monotonic in the information order, there is a formula—built from $\mathbf{0}$, $\mathbf{1}$, $\neg$, $\wedge$, $\vee$, $\sqcup$, and the propositional variables x_1, ..., x_n—that realizes f.[3] Blamey's proof of the result provides an explicit method for constructing a formula that realizes a given f.

Definition 7.3.1 [5–7]. Let $f(x_1, \ldots, x_n)$ be any monotonic function in $\{0, 1, 1/2\}^n \to \{0, 1, 1/2\}$; let A be a 3-valued assignment that is defined on (at least) $x_1, \ldots, x_n$. We define One(f, A, i) and

[3] In a slight abuse of notation, we will refer to a Boolean function f as being a member of $\{0, 1, 1/2\}^n \to \{0, 1, 1/2\}$, but will make use of applications such as $f(A)$, where A is a 3-valued assignment in $\mathcal{V} \to \{0, 1, 1/2\}$. No confusion should result if one thinks of f as a function over the formal parameters $x_1, \ldots, x_n$, and assignment A as supplying values for $x_1, \ldots, x_n$. Under this convention, statements such as $[\![\varphi]\!] = f$ are sensible, even though $[\![\varphi]\!]$ is really of type $(\mathcal{V} \to \{0, 1, 1/2\}) \to \{0, 1, 1/2\}$.

$\text{Zero}(f, A, i)$, for $1 \leq i \leq n$, as well as $\text{One}[f]$, $\text{Zero}[f]$, and $\text{Formula}[f]$, as follows:

$$\text{One}(f, A, i) \stackrel{\text{def}}{=} \begin{cases} x_i & \text{if } f(A) = 1 \text{ and } A(x_i) = 1 \\ \neg x_i & \text{if } f(A) = 1 \text{ and } A(x_i) = 0 \\ \mathbf{1} & \text{if } f(A) = 1 \text{ and } A(x_i) = 1/2 \\ \mathbf{0} & \text{if } f(A) \sqsupseteq 0 \end{cases} \tag{7.3}$$

$$\text{Zero}(f, A, i) \stackrel{\text{def}}{=} \begin{cases} \neg x_i & \text{if } f(A) = 0 \text{ and } A(x_i) = 1 \\ x_i & \text{if } f(A) = 0 \text{ and } A(x_i) = 0 \\ \mathbf{0} & \text{if } f(A) = 0 \text{ and } A(x_i) = 1/2 \\ \mathbf{1} & \text{if } f(A) \sqsupseteq 1 \end{cases} \tag{7.4}$$

$$\text{One}[f] \stackrel{\text{def}}{=} \bigvee_{A \in \{0,1,1/2\}^n} \bigwedge_{1 \leq i \leq n} \text{One}(f, A, i) \tag{7.5}$$

$$\text{Zero}[f] \stackrel{\text{def}}{=} \bigwedge_{A \in \{0,1,1/2\}^n} \bigvee_{1 \leq i \leq n} \text{Zero}(f, A, i) \tag{7.6}$$

$$\text{Formula}[f] \stackrel{\text{def}}{=} \text{One}[f] \sqcup \text{Zero}[f] \tag{7.7}$$

□

Theorem 7.3.2 [Realization Theorem]. [5–7]. For any monotonic function $f(x_1, \ldots, x_n) : \{0, 1, 1/2\}^n \to \{0, 1, 1/2\}$, $\text{Formula}[f]$ realizes f, i.e., $[\![\text{Formula}[f]]\!] = f$.

Proof: See [5]. □

Example 7.3.3 Consider the formula $\varphi \stackrel{\text{def}}{=} x\overline{y} \vee \overline{x}\,\overline{z} \vee yz$. Its truth-functional semantics, $[\![\varphi]\!]$, is shown in the following truth table:

		z		
	y	0	$1/2$	1
$x = 0$	0	1	$1/2$	0
	$1/2$	1	$1/2$	$1/2$
	1	1	$1/2$	1
$x = 1/2$	0	$1/2$	$1/2$	$1/2$
	$1/2$	$1/2$	$1/2$	$1/2$
	1	$1/2$	$1/2$	1
$x = 1$	0	1	1	1
	$1/2$	$1/2$	$1/2$	$1/2$
	1	0	$1/2$	1

Of the twenty-seven disjuncts of One$[[\![\varphi]\!]]$, all but nine are $\mathbf{000}$: $\overline{x}\,\overline{y}\,\overline{z}$, $\overline{x}\mathbf{1}\overline{z}$, $\overline{x}y\overline{z}$, $\overline{x}yz$, $\mathbf{1}yz$, $x\overline{y}\,\overline{z}$, $x\overline{y}\mathbf{1}$, $x\overline{y}z$, xyz. Of the twenty-seven conjuncts of Zero$[[\![\varphi]\!]]$, all but two are $\mathbf{1} \vee \mathbf{1} \vee \mathbf{1}$: $x \vee y \vee \overline{z}$ and $\overline{x} \vee \overline{y} \vee z$. The formula that would be created by Eqn. (7.7) is

$$\text{Formula}[[\![\varphi]\!]] \equiv \left(\begin{array}{l} \quad \overline{x}\,\overline{y}\,\overline{z} \vee \overline{x}\mathbf{1}\overline{z} \vee \overline{x}y\overline{z} \vee \overline{x}yz \vee \mathbf{1}yz \\ \vee \; x\overline{y}\,\overline{z} \vee x\overline{y}\mathbf{1} \vee x\overline{y}z \vee xyz \end{array} \right) \sqcup (x \vee y \vee \overline{z})(\overline{x} \vee \overline{y} \vee z). \tag{7.8}$$

The reader can verify that Eqn. (7.8) realizes the truth table. □

Defn. 7.3.1 is somewhat subtle: for instance, an assignment A on which f evaluates to $1/2$ contributes the conjunction $\bigwedge_{1 \leq i \leq n} \text{One}(f, A, i) = \mathbf{00}\ldots\mathbf{0}$ to formula One$[f]$; however, even though $[[\![\mathbf{00}\ldots\mathbf{0}]\!]](A)$ will necessarily be 0, the overall value of $[[\![\text{One}[f]]\!]](A)$ is not necessarily 0—due to the contributions from other terms that capture how f behaves on other assignments, i.e., $[[\![\bigwedge_{1 \leq i \leq n} \text{One}(f, A', i)]\!]](A)$. Despite such effects, Blamey has shown that the Realization Theorem holds [5].

By applying De Morgan's laws, we can derive several variants of Eqn. (7.7). We have

$$\text{Zero}(f, A, i) \equiv_{\text{DM}} \neg\text{One}(\neg f, A, i)$$
$$\text{One}(f, A, i) \equiv_{\text{DM}} \neg\text{Zero}(\neg f, A, i),$$

which lead to the following variant forms of Eqn. (7.7):

$$\text{Formula}[f] \stackrel{\text{def}}{=} \text{One}[f] \sqcup \neg\text{One}[\neg f] \tag{7.9}$$
$$\text{Formula}[f] \stackrel{\text{def}}{=} \neg\text{Zero}[\neg f] \sqcup \text{Zero}[f] \tag{7.10}$$
$$\text{Formula}[f] \stackrel{\text{def}}{=} \neg\text{Zero}[\neg f] \sqcup \neg\text{One}[\neg f]. \tag{7.11}$$

With Eqn. (7.7), the formula constructed has the form "sum-of-products $\sqcup$ product-of-sums"; with Eqn. (7.9), it has the form "sum-of-products $\sqcup$ $\neg$sum-of-products"; etc. For instance, using Eqn. (7.9) in place of Eqn. (7.7), Eqn. (7.8) becomes

$$\text{Formula}[[\![\varphi]\!]] \equiv \left(\begin{array}{c} \overline{x}\,\overline{y}\,\overline{z} \vee \overline{x}\mathbf{1}\overline{z} \vee \overline{x}y\overline{z} \vee \overline{x}yz \vee \mathbf{1}yz \\ \vee\ x\overline{y}\,\overline{z} \vee x\overline{y}\mathbf{1} \vee x\overline{y}z \vee xyz \end{array}\right) \sqcup \neg(\overline{x}\,\overline{y}z \vee xy\overline{z}). \tag{7.12}$$

7.3.2 Creating a Semantically Minimal Variant

Defn. 7.2.1 and Eqns. (7.7), (7.9), (7.10), or (7.11) give us the tools needed to construct a semantically minimal variant of a formula φ:

Theorem 7.3.4 [Minimization Theorem]. Let ψ be Formula$[\langle\!\langle\varphi\rangle\!\rangle]$. Then ψ is a semantically minimal variant of φ.

Proof: It follows immediately from Defn. 7.2.1 that $\langle\!\langle\varphi\rangle\!\rangle$, the supervaluational semantics of φ, is a monotonic function in $\{0, 1, 1/2\}^n \rightarrow \{0, 1, 1/2\}$. Thus, $\langle\!\langle\varphi\rangle\!\rangle$ meets the conditions of Theorem 7.3.2:

$$\begin{aligned} [\![\psi]\!] &= [\![\text{Formula}[\langle\!\langle\varphi\rangle\!\rangle]]\!] \\ &= \langle\!\langle\varphi\rangle\!\rangle \qquad\qquad \text{(by Theorem 7.3.2),} \end{aligned}$$

and hence ψ is a semantically minimal variant of φ. $\square$

Example 7.3.5 Consider again the formula $\varphi \stackrel{\text{def}}{=} x\overline{y} \vee \overline{x}\,\overline{z} \vee yz$. The following table shows the three 3-valued assignments A for which $\langle\!\langle\varphi\rangle\!\rangle(A) \sqsubset [\![\varphi]\!](A)$:

A	$\langle\!\langle\varphi\rangle\!\rangle(A)$	$[\![\varphi]\!](A)$
$[\ x \mapsto 1/2,\ y \mapsto 0,\ z \mapsto 0\ ]$	1	1/2
$[\ x \mapsto 0,\ y \mapsto 1,\ z \mapsto 1/2\ ]$	1	1/2
$[\ x \mapsto 1,\ y \mapsto 1/2,\ z \mapsto 1\ ]$	1	1/2

Thus, the supervaluational semantics, $\langle\!\langle\varphi\rangle\!\rangle$, is as follows:

		z		
	y	0	1/2	1
$x = 0$	0	1	1/2	0
	1/2	1	1/2	1/2
	1	1	1	1
$x = 1/2$	0	1	1/2	1/2
	1/2	1/2	1/2	1/2
	1	1/2	1/2	1
$x = 1$	0	1	1	1
	1/2	1/2	1/2	1
	1	0	1/2	1

The formula that would be created by the semantic-minimization algorithm (using Eqn. (7.9)) is

$$\text{Formula}[\langle\!\langle\varphi\rangle\!\rangle] \equiv \left(\begin{array}{c} \overline{x}\,\overline{y}\,\overline{z} \vee \overline{x}\mathbf{1}\overline{z} \vee \overline{x}y\overline{z} \vee \overline{x}y\mathbf{1} \\ \vee\ \overline{x}yz \vee \mathbf{1}\overline{y}\,\overline{z} \vee \mathbf{1}yz \vee x\overline{y}\,\overline{z} \\ \vee\ x\overline{y}\mathbf{1} \vee x\overline{y}z \vee x\mathbf{1}z \vee xyz \end{array}\right) \sqcup \neg(\overline{x}\,\overline{y}z \vee xy\overline{z}). \tag{7.13}$$

Comparing with Eqn. (7.12), note the three additional disjuncts in the first part of Eqn. (7.13): $\mathbf{1}\overline{y}\,\overline{z}$, $\overline{x}y\mathbf{1}$, and $x\mathbf{1}z$. These correspond to the three entries that have value $1/2$ in the truth table for $[\![\varphi]\!]$, but have value 1 in the truth table for $\langle\!\langle\varphi\rangle\!\rangle$. □

7.3.3 An Improved Construction for Formula[f]

Blamey's thesis contains an improved construction for Formula$[f]$, which often results in a formula that has fewer, and less complicated, constituents.

Definition 7.3.6 [5]. Let $f(x_1, \ldots, x_n)$ be any monotonic function in $\{0, 1, 1/2\}^n \rightarrow \{0, 1, 1/2\}$; let A be a 3-valued assignment that is defined on (at least) $x_1, \ldots, x_n$. Formula$[f]$ is as defined in Eqns. (7.7), (7.9), (7.10), or (7.11), but with One$[f]$ and Zero$[f]$ redefined as follows:

$$\text{One}[f] \stackrel{\text{def}}{=} \bigvee_{\substack{A\,:\,f(A)=1 \text{ and} \\ \forall A' \sqsupset A.f(A')=1/2}} \bigwedge_{\substack{1 \le i \le n \text{ and} \\ A(x_i) \sqsubset 1/2}} \text{One}(f, A, i) \tag{7.14}$$

$$\text{Zero}[f] \stackrel{\text{def}}{=} \bigwedge_{\substack{A\,:\,f(A)=0 \text{ and} \\ \forall A' \sqsupset A.f(A')=1/2}} \bigvee_{\substack{1 \le i \le n \text{ and} \\ A(x_i) \sqsubset 1/2}} \text{Zero}(f, A, i) \tag{7.15}$$

An empty disjunction has the value 0; an empty conjunction has the value 1. $\square$

The differences between Eqns. (7.5) and (7.6) and Eqns. (7.14) and (7.15) are that, in the latter,

- The outer connectives are indexed by "$A : f(A) = 1$ and $\forall A' \sqsupset A.f(A') = 1/2$" and "$A : f(A) = 0$ and $\forall A' \sqsupset A.f(A') = 1/2$", respectively, which leads to fewer terms being generated.
- The indices of the inner connectives only range over values of i for which $A(x_i)$ is a definite value, and hence One(f, A, i) and Zero(f, A, i) generate only literals, leaving out unnecessary occurrences of $\mathbf{1}$ and $\mathbf{0}$.

Example 7.3.7 Consider again the formula $\varphi \stackrel{\text{def}}{=} x\overline{y} \vee \overline{x}\,\overline{z} \vee yz$ that was discussed in Exs. 7.3.3 and 7.3.5. Suppose that Formula$[f]$ is defined as in Eqn. (7.9), but that One$[f]$ and Zero$[f]$ are defined as in Defn. 7.3.6. The formula that would be created via Formula$[[\![\varphi]\!]]$ is

$$\text{Formula}[[\![\varphi]\!]] \equiv \quad x\overline{y} \vee \overline{x}\,\overline{z} \vee yz \sqcup \neg(\overline{x}\,\overline{y}z \vee xy\overline{z})$$

(cf. Eqn. (7.12)). The semantically minimal variant of φ that would be created via Formula$[\langle\langle\varphi\rangle\rangle]$ is

$$\text{Formula}[\langle\langle\varphi\rangle\rangle] \equiv \quad \overline{y}\,\overline{z} \vee yz \vee \overline{x}\,\overline{z} \vee \overline{x}y \vee xz \vee x\overline{y} \sqcup \neg(\overline{x}\,\overline{y}z \vee xy\overline{z}) \tag{7.16}$$

(cf. Eqn. (7.13)). □

Theorem 7.3.8 [5]. Let f be a monotonic function in $\{0, 1, 1/2\}^n \to \{0, 1, 1/2\}$. Let Formula$[f]$ be the formula One$[f] \sqcup$ Zero$[f]$, where One$[f]$ and Zero$[f]$ are defined as in Eqns. (7.14) and (7.15), respectively. Then Formula$[f]$ realizes f (i.e., $[\![\text{Formula}[f]]\!] = f$).
Proof: See [5]. □

Henceforth, One$[\cdot]$ and Zero$[\cdot]$ mean the operations defined in Defn. 7.3.6 (Eqns. (7.14) and (7.15), respectively).

7.4 A BDD-Based Minimization Algorithm

This section presents an improved algorithm for semantic minimization. The worst-case running time of the algorithm is exponential in the size of φ; however, all operations can be implemented using BDDs.

The issues that we face in using the material that has been presented in Sects. 7.3.2 and 7.3.3 are

1. How do we efficiently represent the function $\langle\langle\varphi\rangle\rangle : \{0, 1, 1/2\}^n \to \{0, 1, 1/2\}$?

2. To use Eqns. (7.14) and (7.15), how do we efficiently implement the indexing operations needed in the outermost connectives:

$$A : f(A) = 1 \text{ and } \forall A' \sqsupset A. f(A') = 1/2 \tag{7.17}$$

$$A : f(A) = 0 \text{ and } \forall A' \sqsupset A. f(A') = 1/2. \tag{7.18}$$

Our approach to issue 1 is to use BDDs [12]. Our approach to issue 2 is based on the following observation:

Observation 7.4.1 The Realization Theorem provides a way to construct a 3-valued propositional formula that realizes *any* given monotonic Boolean function. However, the realization problem that arises in the semantic-minimization problem does not require this general a method: In the semantic-minimization problem, the monotonic Boolean functions that arise are always ones that are in the range of $\langle\!\langle \cdot \rangle\!\rangle$; that is, we are only concerned with realization problems of the form Formula$[\langle\!\langle \varphi \rangle\!\rangle]$. □

Focusing on this special case of the realization problem allows us to sidestep issue 2 by implementing One$[\cdot]$ and Zero$[\cdot]$ differently from the way they are stated in Eqns. (7.14) and (7.15). The approach described takes advantage of the BDD-based representation used to address issue 1.

7.4.1 Representing the Supervaluational Semantics

Given φ, our goal is to find a semantically minimal variant ψ. The truth-valuational semantics of ψ must be equal to the supervaluational semantics of φ:

$$[\![\psi]\!](A) = \langle\!\langle \varphi \rangle\!\rangle(A) = \bigsqcup_{a \text{ rep. by } A} [\![\varphi]\!](a).$$

Thus, in order to capture $\langle\!\langle \varphi \rangle\!\rangle$, we need only concern ourselves with the truth-functional semantics of φ on definite assignments—i.e., just a portion of the truth-functional semantics of φ. In other words, rather than considering $[\![\varphi]\!]$ and $\langle\!\langle \varphi \rangle\!\rangle$ as functions in $\{0, 1, 1/2\}^n \to \{0, 1, 1/2\}$, we need only consider them as functions in $\{0, 1\}^n \to \{0, 1, 1/2\}$. (Functions of the latter type are called **Boolean functions with don't cares** or **incompletely specified Boolean functions**.)

An incompletely specified Boolean function f can be represented via a pair of **total Boolean functions** (i.e., functions in $\{0, 1\}^n \to \{0, 1\}$), denoted by $(\lfloor f \rfloor, \lceil f \rceil)$, where $\lfloor f \rfloor$ conflates 0 and

$1/2$, and $\lceil f \rceil$ conflates 1 and $1/2$:[4]

$$\lfloor f \rfloor(a) = \begin{cases} 1 & \text{if } f(a) = 1 \\ 0 & \text{if } f(a) \sqsupseteq 0 \end{cases} \qquad \lceil f \rceil(a) = \begin{cases} 1 & \text{if } f(a) \sqsupseteq 1 \\ 0 & \text{if } f(a) = 0 \end{cases}$$

$\lfloor f \rfloor$ and $\lceil f \rceil$ can each be represented using ordinary BDDs [12], as proposed by Minato et al. [64].

To capture $\langle\!\langle \varphi \rangle\!\rangle$ (as a function from $\{0,1\}^n \to \{0,1,1/2\}$), it will be represented as the pair $(\lfloor \langle\!\langle \varphi \rangle\!\rangle \rfloor, \lceil \langle\!\langle \varphi \rangle\!\rangle \rceil)$, where $\lfloor \langle\!\langle \varphi \rangle\!\rangle \rfloor$ and $\lceil \langle\!\langle \varphi \rangle\!\rangle \rceil$ are both functions in $\{0,1\}^n \to \{0,1\}$. Given the formula φ (over the propositional variables $x_1, \ldots, x_n$), this can be accomplished by traversing φ, applying the following translation rules bottom-up:

$$\begin{aligned}
\mathbf{0} &\longrightarrow (\lambda a.0, \lambda a.0) \\
\mathbf{1} &\longrightarrow (\lambda a.1, \lambda a.1) \\
\mathbf{1/2} &\longrightarrow (\lambda a.0, \lambda a.1) \\
x_i &\longrightarrow (\lambda a.a(x_i), \lambda a.a(x_i)) \\
\neg(\lfloor f \rfloor, \lceil f \rceil) &\longrightarrow (\neg\lceil f \rceil, \neg\lfloor f \rfloor) \\
(\lfloor f_1 \rfloor, \lceil f_1 \rceil) \wedge (\lfloor f_2 \rfloor, \lceil f_2 \rceil) &\longrightarrow (\lfloor f_1 \rfloor \wedge \lfloor f_2 \rfloor, \lceil f_1 \rceil \wedge \lceil f_2 \rceil) \\
(\lfloor f_1 \rfloor, \lceil f_1 \rceil) \vee (\lfloor f_2 \rfloor, \lceil f_2 \rceil) &\longrightarrow (\lfloor f_1 \rfloor \vee \lfloor f_2 \rfloor, \lceil f_1 \rceil \vee \lceil f_2 \rceil) \\
(\lfloor f_1 \rfloor, \lceil f_1 \rceil) \sqcup (\lfloor f_2 \rfloor, \lceil f_2 \rceil) &\longrightarrow (\lfloor f_1 \rfloor \wedge \lfloor f_2 \rfloor, \lceil f_1 \rceil \vee \lceil f_2 \rceil) \\
(\lfloor f_1 \rfloor, \lceil f_1 \rceil) \,?\, (\lfloor f_2 \rfloor, \lceil f_2 \rceil) : (\lfloor f_3 \rfloor, \lceil f_3 \rceil) &\longrightarrow \\
&\left(\begin{array}{l} (\lfloor f_1 \rfloor \,?\, \lfloor f_2 \rfloor : \lfloor f_3 \rfloor) \wedge (\lceil f_1 \rceil \,?\, \lfloor f_2 \rfloor : \lfloor f_3 \rfloor), \\ (\lceil f_1 \rceil \,?\, \lceil f_2 \rceil : \lceil f_3 \rceil) \wedge (\lfloor f_1 \rfloor \,?\, \lceil f_2 \rceil : \lceil f_3 \rceil) \end{array} \right)
\end{aligned} \tag{7.19}$$

All of the operations on total Boolean functions that are required on the right-hand sides of the above rules are ones from the standard repertoire of BDD operations: the creation of BDDs for $\lambda a.0$, $\lambda a.1$, and $\lambda a.a(x_i)$ (for $1 \le i \le n$), and the application of the following Boolean operations to existing BDDs: $\neg$, $\wedge$, $\vee$, and $(\cdot\,?\,\cdot:\cdot)$ (also known as ITE [11]).

[4] Thus, there are three types of Boolean functions that play a role in this chapter:

(3-valued) Boolean functions:	$\{0,1,1/2\}^n \to \{0,1,1/2\}$
incompletely specified Boolean functions:	$\{0,1\}^n \to \{0,1,1/2\}$
total Boolean functions:	$\{0,1\}^n \to \{0,1\}$

We do not introduce any special notation to distinguish among functions of the three types (although terms of the form $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ always denote total Boolean functions). In particular, occurrences of $[\![\varphi]\!]$ and $\langle\!\langle\varphi\rangle\!\rangle$ sometimes denote incompletely specified Boolean functions (which may be total Boolean functions), but sometimes denote 3-valued Boolean functions. However, it should always be clear from context which use is intended.

7.4.2 Realization for Semantic Minimization

Definition 7.4.2 (Cf. [18,74].) A superscripted propositional variable x^b, where $b \in \{0, 1\}$, stands for a literal:

$$x^b \stackrel{\text{def}}{=} \begin{cases} x & \text{if } b = 1 \\ \neg x & \text{if } b = 0 \end{cases}$$

Let f be a total Boolean function over the propositional variables $x_1, \ldots, x_n$. A conjunction of literals in which each propositional variable appears at most once—either negated or unnegated—i.e.,

$$\bigwedge_{i \in S \subseteq \{1,\ldots,n\}} x_i^{b_i} \tag{7.20}$$

is an **implicant** of f if, for every 2-valued assignment a such that $a(x_i) = b_i$ for all $i \in S$, we have $f(a) = 1$.

Each conjunction of literals of the form shown in (7.20) can be thought of as the set of literals $\{x_i^{b_i} \mid i \in S\}$. An implicant is a **prime implicant** if none of its proper subsets is an implicant (i.e., corresponds to a conjunction of literals that is an implicant). □

Example 7.4.3 xy, xz, and xyz are all implicants of $xz \vee y\overline{z}$. x, y, and z are not implicants of $xz \vee y\overline{z}$; hence, xy and xz are prime implicants. □

Our concern is with realization problems of the form Formula$[f]$, where $f = \langle\langle \varphi \rangle\rangle$. We now show that for this case, realization can be can be implemented as follows:

$$\text{Formula}[f] \equiv \text{Primes}[\lfloor f \rfloor] \sqcup \neg\text{Primes}[\neg\lceil f \rceil],$$

where Primes$[g]$ is the operation that, given a total Boolean function g, creates the disjunction of g's prime implicants:

$$\text{Primes}[g] \stackrel{\text{def}}{=} \bigvee_{\pi \text{ a prime implicant of } g} \pi.$$

Lemma 7.4.4 Let A be a 3-valued assignment such that (i) $\langle\!\langle\varphi\rangle\!\rangle(A) = 1$, and (ii) for all $A' \sqsupset A$, $\langle\!\langle\varphi\rangle\!\rangle(A') = 1/2$. Then the formula

$$\pi \stackrel{\text{def}}{=} \bigwedge_{\substack{1 \leq i \leq n \text{ and} \\ A(x_i) \sqsubset 1/2}} \text{One}(\langle\!\langle\varphi\rangle\!\rangle, A, i)$$

is a prime implicant of $\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor$.

Proof: See App. B. $\square$

Lemma 7.4.5 Let π by a prime implicant of $\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor$. Then there is a 3-valued assignment A such that

(i) $\langle\!\langle\varphi\rangle\!\rangle(A) = 1$

(ii) For all $A' \sqsupset A$, $\langle\!\langle\varphi\rangle\!\rangle(A') = 1/2$

(iii) $\pi \equiv \bigwedge_{\substack{1 \leq i \leq n \text{ and} \\ A(x_i) \sqsubset 1/2}} \text{One}(\langle\!\langle\varphi\rangle\!\rangle, A, i)$

Proof: See App. B. $\square$

These results yield the following procedure for semantic minimization:

Theorem 7.4.6 Let ψ be $\text{Primes}[\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor] \sqcup \neg\text{Primes}[\neg\lceil\langle\!\langle\varphi\rangle\!\rangle\rceil]$. Then ψ is a semantically minimal variant of φ.

Proof: From Defn. 7.3.6 (i.e., Eqn. (7.14)), and Lemmas 7.4.4 and 7.4.5, it follows that $\text{One}[\langle\!\langle\varphi\rangle\!\rangle] \equiv \text{Primes}[\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor]$. $\neg\langle\!\langle\varphi\rangle\!\rangle$ is represented by both $(\lfloor\neg\langle\!\langle\varphi\rangle\!\rangle\rfloor, \lceil\neg\langle\!\langle\varphi\rangle\!\rangle\rceil)$ and $\neg(\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor, \lceil\langle\!\langle\varphi\rangle\!\rangle\rceil)$; by translation method (7.19), the latter equals $(\neg\lceil\langle\!\langle\varphi\rangle\!\rangle\rceil, \neg\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor)$. This implies that $\lfloor\neg\langle\!\langle\varphi\rangle\!\rangle\rfloor = \neg\lceil\langle\!\langle\varphi\rangle\!\rangle\rceil$, and hence

$$\begin{aligned} \text{Zero}[\langle\!\langle\varphi\rangle\!\rangle] &\equiv_{\text{DM}} \neg\text{One}[\neg\langle\!\langle\varphi\rangle\!\rangle] \\ &\equiv \neg\text{Primes}[\lfloor\neg\langle\!\langle\varphi\rangle\!\rangle\rfloor] \\ &\equiv \neg\text{Primes}[\neg\lceil\langle\!\langle\varphi\rangle\!\rangle\rceil]. \end{aligned}$$

The claim now follows from Theorems 7.3.8 and 7.3.4. $\square$

```
[1] formula MinimizeFormula(formula φ) {
[2]   Transform φ to (⌊⟪φ⟫⌋, ⌈⟪φ⟫⌉) using
                    translation method (7.19)
[3]   return Primes[⌊⟪φ⟫⌋] ⊔ ¬Primes[¬⌈⟪φ⟫⌉]
[4] }
```

Figure 7.3 A minimization algorithm

Theorem 7.4.6 provides the justification for the function MinimizeFormula, shown in Fig. 7.3; given a propositional formula φ as input, MinimizeFormula creates and returns a semantically minimal variant of φ. MinimizeFormula uses the auxiliary procedure Primes$[f]$, which creates a sum-of-prime-implicants formula for a given formula f. Any of several known methods for efficiently generating prime implicants can be used for this step [18, 19, 85]. (These methods all start from the BDD representation of f; thus, when line [2] is implemented with BDDs, exactly the right kind of input structure is at hand.)

Let us return again to the formula $\varphi \stackrel{\text{def}}{=} x\overline{y} \vee \overline{x}\,\overline{z} \vee yz$ (cf. Exs. 7.3.3, 7.3.5, and 7.3.7). MinimizeFormula would create the formula that we saw in Eqn. (7.16) of Ex. 7.3.7—although MinimizeFormula would arrive at the answer by a different, and more efficient, method:

$$\text{Formula}[\langle\!\langle\varphi\rangle\!\rangle] \equiv \quad \overline{y}\,\overline{z} \vee yz \vee \overline{x}\,\overline{z} \vee \overline{x}y \vee xz \vee x\overline{y} \sqcup \neg(\overline{x}\,\overline{y}z \vee xy\overline{z}). \tag{7.16}$$

A drawback of MinimizeFormula is the need to generate all prime implicants. One might try to substitute other sum-of-products expressions that can be used to represent a given function (in 2-valued logic), such as an irredundant prime cover [20, 63]. This approach is not tenable, however; for instance, for the formula $\varphi \stackrel{\text{def}}{=} x\overline{y} \vee \overline{x}\,\overline{z} \vee yz$, if we substitute the irredundant-prime-cover algorithm from [20] for the two calls on Primes$[\cdot]$ in line [3] of MinimizeFormula, we would get the following formula:

$$\overline{y}\,\overline{z} \vee yz \vee \overline{x}\,\overline{z} \vee xz \sqcup \neg(\overline{x}\,\overline{y}z \vee xy\overline{z}). \tag{7.21}$$

However, formula (7.21) is not a semantically minimal variant of φ: for the assignment $[x \mapsto 0, y \mapsto 1, z \mapsto 1/2]$, Eqn. (7.16) evaluates to 1, whereas formula (7.21) evaluates to $1/2$. Moreover, formula (7.21) is actually *worse* than φ itself (and Eqn. (7.16)) for the assignment $[x \mapsto 1, y \mapsto 0, z \mapsto 1/2]$: φ and Eqn. (7.16) evaluate to 1, whereas formula (7.21) evaluates to $1/2$.

7.4.3 Other Semantically Minimal Formulas

In this section, we derive some other forms, different from the one that was the subject of Theorem 7.4.6, in which one can express a semantically minimal formula. The proof of the following theorem can be found in App. B:

Theorem 7.4.7 If f is a total Boolean function, then $[\![\text{Primes}[f]]\!] = [\![\neg\text{Primes}[\neg f]]\!]$. □

In the proof of Theorem 7.4.6, we showed

$$\begin{aligned} \text{One}[\langle\!\langle\varphi\rangle\!\rangle] &\equiv \quad \text{Primes}[\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor] \\ \text{Zero}[\langle\!\langle\varphi\rangle\!\rangle] &\equiv_{\text{DM}} \neg\text{Primes}[\neg\lceil\langle\!\langle\varphi\rangle\!\rangle\rceil]. \end{aligned}$$

These imply

$$[\![\text{One}[\langle\!\langle\varphi\rangle\!\rangle]]\!] = [\![\text{Primes}[\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor]]\!] \tag{7.22}$$

$$[\![\text{Zero}[\langle\!\langle\varphi\rangle\!\rangle]]\!] = [\![\neg\text{Primes}[\neg\lceil\langle\!\langle\varphi\rangle\!\rangle\rceil]]\!]. \tag{7.23}$$

Applying Theorem 7.4.7 results in two new relationships:

$$[\![\text{One}[\langle\!\langle\varphi\rangle\!\rangle]]\!] = [\![\neg\text{Primes}[\neg\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor]]\!] \tag{7.24}$$

$$[\![\text{Zero}[\langle\!\langle\varphi\rangle\!\rangle]]\!] = [\![\text{Primes}[\lceil\langle\!\langle\varphi\rangle\!\rangle\rceil]]\!]. \tag{7.25}$$

Consequently, we can "mix and match" Eqns. (7.22), (7.23), (7.24), and (7.25) to create expressions that yield semantically minimal formulas different from the one given in Theorem 7.4.6. That is, the function MinimizeFormula of Fig. 7.3 creates a semantically minimal variant of φ with any of the following four expressions used in line [3]:

		Zero	
		sum-of-products	¬sum-of-products
One	sum-of-products	Primes[$\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor$] ⊔ Primes[$\lceil\langle\!\langle\varphi\rangle\!\rangle\rceil$]	Primes[$\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor$] ⊔ ¬Primes[$\neg\lceil\langle\!\langle\varphi\rangle\!\rangle\rceil$]
	¬sum-of-products	¬Primes[$\neg\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor$] ⊔ Primes[$\lceil\langle\!\langle\varphi\rangle\!\rangle\rceil$]	¬Primes[$\neg\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor$] ⊔ ¬Primes[$\neg\lceil\langle\!\langle\varphi\rangle\!\rangle\rceil$]

A term in ¬sum-of-products form can be put in product-of-sums form (with no blow-up in size) by applying De Morgan's laws. Thus, we may create a semantically minimal formula with any combination of sum-of-products and product-of-sums terms that we desire.

Our final result provides a condition under which we may generate a semantically minimal variant that does not contain an occurrence of ⊔:

Corollary 7.4.8 Suppose that φ is a formula such that $\langle\!\langle\varphi\rangle\!\rangle$ is a total Boolean function. Let $\psi_1 \stackrel{\text{def}}{=} \text{Primes}[\langle\!\langle\varphi\rangle\!\rangle]$ and $\psi_2 \stackrel{\text{def}}{=} \neg\text{Primes}[\neg\langle\!\langle\varphi\rangle\!\rangle]$. Then ψ_1 and ψ_2 are both semantically minimal variants of φ.

Proof: When $\langle\!\langle\varphi\rangle\!\rangle$ is a total Boolean function, $\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor = \lceil\langle\!\langle\varphi\rangle\!\rangle\rceil = \langle\!\langle\varphi\rangle\!\rangle$. The result follows from Eqns. (7.22), (7.23), (7.24), and (7.25)—and the elimination of duplicate terms in the diagonal entries of the table given above. □

In particular, any formula that does not contain an explicit occurrence of $\mathbf{1/2}$ or ⊔ has a semantically minimal variant that does not contain an occurrence of ⊔.

Example 7.4.9 Consider the formula $\varphi \stackrel{\text{def}}{=} xy \vee \overline{x}z$. In 2-valued logic, φ can be treated as a syntactic shorthand for $x\ ?\ y\ :\ z$. In Sect. 7.1.2, we discussed why φ is not a suitable syntactic shorthand for (the extension of) $x\ ?\ y\ :\ z$ to 3-valued logic, and why $xy \vee \overline{x}z \vee yz$ is a suitable shorthand.

We can now derive this by means of our results on semantic minimization: $xy \vee \overline{x}z$ does not contain an explicit occurrence of $\mathbf{1/2}$ or ⊔, and thus $\langle\!\langle xy \vee \overline{x}z\rangle\!\rangle$ is a total Boolean function. Because $\text{Primes}[\langle\!\langle xy \vee \overline{x}z\rangle\!\rangle] = xy \vee \overline{x}z \vee yz$, $xy \vee \overline{x}z \vee yz$ is a semantically minimal variant of $xy \vee \overline{x}z$. □

7.5 Related Work

There is a substantial body of work that addresses methods for syntactic minimization of propositional formula. Previous work has addressed finding minimal-size sum-of-products formulas [74], as well as minimal-size formulas for other forms [86]. In contrast, this chapter concerns semantic minimization (in 3-valued propositional logic). Because the minimization criterion is a semantic one, rather than a syntactic one, the formula ψ that results is not necessarily smaller than φ.

The realization problem, and the two versions of the Realization Theorem that we have used, are due to Blamey [5–7]. However, Blamey's work did not address the semantic-minimization problem that we defined in Sect. 7.2. In Sect. 7.4, we focused on a special case of the realization problem, which allowed us to define a semantic-minimization algorithm (MinimizeFormula) that is more efficient than what one would have using the general realization constructions given by Blamey.

Our motivation for investigating the semantic-minimization problem for propositional logic was as a heuristic for creating "better" formulas in 3-valued first-order logic (with a transitive-closure operator), when applying the finite-differencing transformations to defining formulas of instrumentation relations. By replacing a formula φ with a formula ψ, we may improve the precision of the answers that the system obtains. We have implemented MinimizeFormula, and have used it as a subroutine in a heuristic method for minimizing first-order formulas; the method works on a formula bottom-up, applying MinimizeFormula to the body of each non-propositional operator (i.e., each quantifier or transitive-closure operator).

Chapter 8

Conclusions and Future Work

This thesis addressed the following fundamental challenges in applying abstract interpretation, namely, given a program, the concrete semantics for a language, and a query of interest,

1. How does one create an abstraction that is sufficiently precise to verify that the program satisfies the query?

and

2. How does one create the associated abstract transformers?

Challenge 2 arises in program-analysis problems in which the semantics of statements is expressed using logical formulas that describe changes to core-relation values. When instrumentation relations have been introduced to refine an abstraction, the challenge is to reflect the changes in core-relation values in the values of the instrumentation relations. The algorithm presented in Chapter 3 provides a way to create formulas that maintain correct values for the instrumentation relations, and thereby provides a way to generate—*completely automatically*—the part of the transformers of an abstract semantics that deals with instrumentation relations. This research was motivated by work on static analysis based on 3-valued logic; however, any analysis method that relies on logic—2-valued or 3-valued—to express a program's semantics may be able to benefit from these techniques.

Chapter 5 addressed Challenge 1 by presenting an approach to creating abstractions automatically for use in program analysis. As in some previous work, the approach involves the successive refinement of the abstraction in use. Unlike previous work, the work presented in that

chapter is aimed at programs that manipulate pointers and heap-allocated data structures. However, while we demonstrated our approach on shape-analysis problems, the approach is applicable in any program-analysis setting that uses first-order logic. Refinement is performed by introducing new instrumentation relations (defined via logical formulas over core relations). Our abstraction-refinement method uses two refinement strategies. The first strategy, subformula-based refinement, analyzes the sources of imprecision in the evaluation of the query, and chooses how to define new instrumentation relations using subformulas of the query. The second strategy, ILP-based refinement, employs inductive logic programming to learn new instrumentation relations that can stave off imprecision due to abstraction. The steps of ILP go beyond merely forming Boolean combinations of existing relations (as in many refinement techniques based on predicate abstraction); ILP can create new relations by introducing quantifiers during the learning process.

Chapter 6 discussed the automated verification of the *total correctness* (partial correctness and termination) of the Deutsch-Schorr-Waite tree-traversal algorithm. Past approaches have involved hand-written proofs of complicated invariants to verify the partial correctness of the algorithm. Even with some automation, these efforts were usually laborious: a proof performed in 2002 with the help of the Jape proof editor took 152 pages [8]. The key advantage of our abstract-interpretation approach over proof-theoretic approaches is that a relatively small number of concepts are involved in defining an abstraction of the structures that can arise on any execution, and verification is then carried out automatically by symbolic exploration of all memory configurations that can arise.

While studying the question of precision in analyses based on 3-valued logic, we encountered a non-standard logic-minimization problem that arises in 3-valued propositional logic. Chapter 7 presented a formalization of the "semantic-minimization" problem, showed that a semantically-minimal formula always exists, and gave several methods for creating semantically-minimal formulas.

Discussion

The methodology followed in this work makes use of a powerful logic for expressing the semantics of programs and the query of interest. This logic (first-order logic extended with a transitive-closure operator) allows the user to express properties and transformations of heap-allocated data structures with ease. However, this convenience comes at a cost. The logic, as well as most fragments that are expressive enough for stating interesting properties of heap-allocated data structures, is undecidable. While semi-decision procedures for the logic exist, the performance of the theorem provers that implement them makes their use prohibitively expensive for many verification tasks, at least at the present time.

In our work, we chose not to rely on theorem provers. The key operation in our analysis is formula evaluation, rather than a deductive step. This presented the following non-standard issue: a pair of formulas that are really equivalent are not seen as equivalent by the analysis. Below we give some instances where this issue arises and state what novel techniques we were led to develop while addressing that instance of the issue.

The formula $p \vee \neg p$ is equivalent to the formula $\mathbf{1}$, yet the latter evaluates to a more precise value under the assignment $[p \mapsto 1/2]$. This observation led us to formalize the problem of semantic minimization and to define algorithms for computing semantically-minimal formulas.

The formula $p(\cdot)$ is "equivalent" to the formula ψ_p, i.e., the defining formula of p, yet $p(\cdot)$ will frequently evaluate to a more precise value than ψ_p. This observation led us to work on finite differencing, which provides a mechanism to update the values of instrumentation relations while maximizing the reuse of stored values of instrumentation relations. Additionally, it led us to work on an abstraction-refinement method that introduces new instrumentation relations to serve as new sources of stored values.

Our abstraction-refinement method is intended to remove the obligation of identifying loop invariants and translating them into appropriate abstraction definitions. In the absence of a decision procedure that can help in identifying loop invariants, we had to design techniques to "extract" information from the logical structures that arise during abstract interpretation. We decided to

forego deductive techniques in favor of viewing logical structures as unstructured collections of values and learning relationships that are present in those collections.

Future Work

The use of heap-allocated data structures offers a programmer great flexibility, and, as a result, such data structures are ubiquitous in today's software. However, code that manipulates heap-allocated storage can harbor errors, as well as unnecessary slowdowns and potential resource bottlenecks. We plan to continue to do research and develop better tools for program understanding, debugging, and reverse engineering that apply to the analysis of programs that manipulate heap-allocated data structures.

The expressive power of our research system makes it suitable in settings besides shape analysis. In future work, we would like to address other important issues in the quality of software, such as concurrency and information flow. TVLA's power of automatic exploration may also be suitable for solving challenging problems that arise in Computer Architecture, e.g., the verification of cache-coherence protocols.

In an effort to design a more scalable analysis, we are intrigued by the possibility of defining *parsimonious abstractions* [36]. Introduced in the software-model-checking community, parsimonious abstractions allow the precision of the abstraction to vary from one program point to another. The application of inductive learning at different points in the program offers interesting possibilities in this regard.

Inductive learning may have interesting applications in software model checking, where the standard abstraction mechanism is predicate abstraction, which groups states based on their values for a set of nullary logical formulas. Our extension of the FOIL algorithm to permit learning nullary formulas that differentiate one set of structures from another makes it possible to refine an abstraction in the predicate-abstraction framework by inventing relations that capture properties of heap-allocated storage.

In a broader context, we plan to continue to mine connections between program analysis and machine learning exposed in this thesis. We believe that program code itself, and not comments,

annotations, or specifications, is the best descriptor of the behavior of the program. We will further explore how machine-learning techniques can be employed to extract interesting information in static, as well as dynamic, analyses.

In the domain of program understanding, an interesting application of our techniques is to apply inductive learning for inferring loop invariants automatically. This application also suggests an obvious way to integrate our approach with proof-based approaches to verification. The need for manually defined loop invariants prevents widespread use of proof-based systems. The use of inductive learning for inferring loop invariants automatically can address this limitation.

Another possible application of inductive learning in the domain of program understanding is to summarize the effect of several statements (or even entire procedures) via inductive learning. (This application can also benefit program analysis by creating summary transformers.) In this application, the system can invoke a learning algorithm to learn a formula (during static analysis or from data gathered during dynamic analysis), and then use static analysis to verify that the formula is indeed an invariant of the program. We saw instances of this in the experiments presented in Chapter 5, where learning resulted in relations r_2, r_{21}, and r_{23} defined by Formulas (5.13), (5.16), and (5.18), respectively, which capture interesting properties of sorting procedures and procedure `Reverse`.

An interesting challenge is to describe the essence of a bug fix by using the states of the fixed and the buggy version of a program as sources of positive and negative examples, respectively. This application can help in understanding and documenting the change, as well as in regression-test generation.

Suppose the analyzed software system consists of a client and a server, and the client does not respect the server's API. If a client that respects the API is available but the specification for the API is not available, learning (or specification mining) can be used to infer the API based on the states of the two clients. Additionally, whether or not the API is available, the answers produced by the learning algorithm can be used to suggest possible fixes to bring the incorrect client in line with the correct one.

If the API is available but a correct client is not, it may be possible to search for a fix to the buggy client by performing small perturbations to the program (e.g., interchange certain system calls). When a collection of perturbations results in learning a specification that agrees with the server's API, a fix is obtained. The success of Delta Debugging gives hope that such an approach is feasible [101, 102].

Learning can serve to "bridge the gap" between testing and verification. The application of learning to the results of testing (or data collected during a dynamic analysis) can inform verification by helping construct an abstraction based on the states observed in testing (or during dynamic analysis). Many algorithms that manipulate linked data structures are "storeless": the invariants of such an algorithm (and thus an abstraction that is sufficient to verify the algorithm's correctness) can be stated via properties of nodes in different regions of the data structure that the algorithm manipulates.[1] (`InsertSort` and Deutsch-Schorr-Waite are examples of storeless algorithms.) Learning can be applied to learn properties that hold for the nodes of a given region.

On the other hand, the search performed by a learning algorithm on abstract states computed during verification may be able to find the key differentiators between good and bad states, thus helping to construct test cases that can drive the code to the good and the bad states.

We anticipate that learning will occupy a prominent place in our future research. In large software systems of today, it is important to automate many tasks currently left to the user. For instance, while most research today focuses on verifying that a software system conforms to an existing formal specification, it is often unrealistic to expect that formal specifications of software systems (or even specifications of APIs) are available. Learning may hold the key to many future endeavors in program understanding, verification, testing, and reverse engineering by providing techniques to infer specifications, verification conditions, abstractions, and even test cases.

[1] The regions can be defined in terms of reachability relations.

LIST OF REFERENCES

[1] S. Akers. On a theory of Boolean functions. *J. Society for Industrial and Applied Math.*, 7(4):487–498, December 1959.

[2] G. Ammons, R. Bodik, and J. Larus. Mining specifications. In *Symp. on Principles of Programming Languages*, pages 4–16, January 2002.

[3] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Conf. on Programming Language Design and Impl.*, June 2001.

[4] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN workshop on Model checking of software*, pages 103–122, May 2001.

[5] S. Blamey. *Partial-Valued Logic*. PhD thesis, Univ. of Oxford, Oxford, Eng., 1980.

[6] S. Blamey. Partial logic. In D.M. Gabbay and F. Guenthner, editors, *Handbook of Phil. Logic, Vol. 3: Alternatives To Classical Logic*, pages 1–70. D. Reidel Publishing, 1986.

[7] S. Blamey. Partial logic. In D.M. Gabbay and F. Guenthner, editors, *Handbook of Phil. Logic, 2nd. Ed., Vol. 5*, pages 261–353. Kluwer Acad., 2002.

[8] R. Bornat. Proofs of pointer programs in Jape. "Available at http://www.dcs.qmul.ac.uk/ ~richard/pointers/".

[9] R. Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, July 2000.

[10] R. Bornat and B. Sufrin. Animating formal proofs at the surface: The Jape proof calculator. *The Computer Journal*, 43:177–192, 1999.

[11] K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a BDD package. In *Design Automation Conf.*, pages 40–45, June 1990.

[12] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, C-35(6):677–691, August 1986.

[13] D. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *Conf. on Programming Language Design and Impl.*, pages 296–310, June 1990.

[14] C.-T. Chou. The mathematical foundation of symbolic trajectory evaluation. In *Computer-Aided Verification*. Springer-Verlag, July 1999.

[15] P. Clark and T. Niblett. The CN2 induction algorithm. *Machine Learning*, 3:261–283, 1989.

[16] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer-Aided Verification*, pages 154–169, July 2000.

[17] C. Colby, P. Godefroid, and L. Jagadeesan. Automatically closing open reactive programs. In *Conf. on Programming Language Design and Impl.*, pages 345–357, June 1998.

[18] O. Coudert. Two-level logic minimization: An overview. *Integration, The VLSI Journal*, 17(2):97–140, October 1994.

[19] O. Coudert and J.-C. Madre. A new graph based prime computation technique. In T. Sasao, editor, *Logic Synthesis and Optimization*, pages 33–57. Kluwer Acad., January 1993.

[20] O. Coudert, J.-C. Madre, H. Fraisse, and H. Touati. Implicit prime cover computation: An overview. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI)*, October 1993.

[21] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Principles of Programming Languages*, pages 269–282, January 1979.

[22] S. Das and D. Dill. Counter-example based predicate discovery in predicate abstraction. In *Formal Methods in Computer-Aided Design*, pages 19–32, November 2002.

[23] S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *Computer-Aided Verification*, pages 160–171, July 1999.

[24] G. Dong and J. Su. Incremental and decremental evaluation of transitive closure by first-order queries. *Information and Computation*, 120:101–106, 1995.

[25] G. Dong and J. Su. Incremental maintenance of recursive views using relational calculus/SQL. *SIGMOD Record*, 29(1):44–51, 2000.

[26] N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *Static Analysis Symp.*, June 2000.

[27] P. Flach. A framework for inductive logic programming. In S. Muggleton, editor, *Inductive Logic Programming*, volume 38 of *the APIC Series*, pages 193–211. Academic Press, 1992.

[28] C. Flanagan. Software model checking via iterative abstraction refinement of constraint logic queries. In *Workshop on Constraint Programming and Constraints for Verification*, March 2004.

[29] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361–416, 2000.

[30] M. Ginsberg. Multivalued logics: A uniform approach to inference in artificial intelligence. *Computational Intelligence*, 4:265–316, 1988.

[31] D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *Tools and Algs. for the Construction and Analysis of Systems*, pages 512–529, March 2004.

[32] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification*, pages 72–83, June 1997.

[33] A. Gupta and I.S. Mumick, editors. *Materialized Views: Techniques, Implementations, and Applications*. The M.I.T. Press, Cambridge, MA, 1999.

[34] N. Helft. Induction as nonmonotonic inference. In *Principles of Knowledge Representation and Reasoning*, pages 149–156, May 1989.

[35] L. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Dept. of Computer Science, Cornell University, January 1990.

[36] T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *Symp. on Principles of Programming Languages*, pages 232–244, January 2004.

[37] W. Hesse. *Dynamic Computational Complexity*. PhD thesis, Dept. of Computer Science, University of Massachusetts, June 2003.

[38] N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive closure logics. In *Workshop on Computer Science Logic*, pages 160–174, September 2004.

[39] N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. Verification via structure simulation. In *Computer-Aided Verification*, pages 281–294, July 2004.

[40] S. Ishtiaq and P. O'Hearn. Bi as an assertion language for mutable data structures. In *Symp. on Principles of Programming Languages*, pages 14–26, January 2001.

[41] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In *Static Analysis Symp.*, pages 39–50, September 1999.

[42] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *Static Analysis Symp.*, pages 246–264, August 2004.

[43] N. Jones and S. Muchnick. Flow analysis and optimization of Lisp-like structures. In *Program Flow Analysis: Theory and Applications*, pages 102–131. Prentice-Hall, 1981.

[44] N. Klarlund and M. Schwartzbach. Graph types. In *Symp. on Principles of Programming Languages*, January 1993.

[45] S. Kleene. *Introduction to Metamathematics*. North-Holland, 2nd edition, 1987.

[46] D. Knuth. *The Art of Computer Programming – Vol. 1, Fundamental Algorithms*. Addison-Wesley, 1973.

[47] R. Kurshan. *Computer-aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[48] S. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *Symp. on Principles of Programming Languages*, pages 115–126, January 2006.

[49] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *Tools and Algs. for the Construction and Analysis of Systems*, pages 98–112, April 2001.

[50] N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, 1994.

[51] O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *European Symp. On Programming*, pages 124–140, April 2005.

[52] T. Lev-Ami, N. Immerman, and M. Sagiv. Fast and precise abstraction for shape analysis. In *Computer-Aided Verification*, pages 533–546, August 2006.

[53] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *Int. Symp. on Software Testing and Analysis*, pages 26–38, August 2000.

[54] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symp.*, pages 280–301, June 2000.

[55] B. Liblit, A. Aiken, A. Zheng, and M. Jordan. Bug isolation via remote program sampling. In *Conf. on Programming Language Design and Impl.*, pages 141–154, June 2003.

[56] G. Lindstrom. Scanning list structures without stacks or tag bits. *Information Processing Letters*, 2(2):47–51, June 1973.

[57] Y. Liu, S. Stoller, and T. Teitelbaum. Discovering auxiliary information for incremental computation. In *Symp. on Principles of Programming Languages*, pages 157–170, January 1996.

[58] Y. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Science of Computer Programming*, 24:1–39, February 1995.

[59] R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *Static Analysis Symp.*, pages 265–279, August 2004.

[60] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *Verification, Model Checking, and Abstract Interpretation*, pages 181–198, January 2005.

[61] K. McMillan. Verification of infinite state systems by compositional model checking. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 219–234, September 1999.

[62] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In *Int. Conf. on Automated Deduction (CADE)*, pages 121–135, July 2003.

[63] S. Minato. Fast generation of irredundant sum-of-products forms from binary decision diagrams. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI)*, pages 64–73, April 1992.

[64] S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *Design Automation Conf.*, pages 52–57, June 1990.

[65] T. Mitchell. *Machine Learning*. McGraw-Hill, New York, June 1997.

[66] A. Møller and M. Schwartzbach. The pointer assertion logic engine. In *Conf. on Programming Language Design and Impl.*, pages 221–231, June 2001.

[67] J. Morris. Verification-oriented language design. Tech. Report TR-7, Computer Science Div., University of California–Berkeley, December 1972.

[68] S. Muggleton. Inductive logic programming. *New Generation Comp.*, 8(4):295–317, 1991.

[69] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *J. Logic Programming*, 19/20:629–679, 1994.

[70] G. Nelson. Verifying reachability invariants of linked structures. In *Symp. on Principles of Programming Languages*, pages 38–47, January 1983.

[71] R. Paige and S. Koenig. Finite differencing of computable expressions. *Trans. on Programming Languages and Systems (TOPLAS)*, 4(3):402–454, July 1982.

[72] C. Pasareanu, M. Dwyer, and W. Visser. Finding feasible counter-examples when model checking Java programs. In *Tools and Algs. for the Construction and Analysis of Systems*, pages 284–298, April 2001.

[73] S. Patnaik and N. Immerman. Dyn-FO: A parallel, dynamic complexity class. *J. Computational Systems Science*, 55(2):199–209, October 1997.

[74] W. Quine. The problem of simplifying truth functions. *American Mathematical Monthly*, 59(8):521–531, October 1952.

[75] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[76] J.R. Quinlan and R.M. Cameron-Jones. FOIL: A midterm report. In *European Conference on Machine Learning*, pages 3–20, April 1993.

[77] J.R. Quinlan and R.M. Cameron-Jones. Induction of logic programs: FOIL and related systems. *New Generation Comp., Special issue on ILP*, 13(3-4):287–312, 1995.

[78] G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *Conf. on Programming Language Design and Impl.*, pages 83–94, June 2002.

[79] T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *European Symp. On Programming*, pages 380–398, April 2003.

[80] T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *Verification, Model Checking, and Abstract Interpretation*, pages 252–266, January 2004.

[81] J. Reynolds. Separation Logic: A logic for shared mutable data structures. In *Symp. on Logic in Computer Science*, pages 55–74, July 2002.

[82] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *Symp. on Principles of Programming Languages*, pages 296–309, January 2005.

[83] N. Rinetzky and M. Sagiv. Interprocedural shape analysis for recursive programs. In *Int. Conf. on Compiler Construction*, pages 133–149, April 2001.

[84] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Trans. on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.

[85] T. Sasao. Ternary decision diagrams and their applications. In Sasao and Fujita [86], pages 269–292.

[86] T. Sasao and M. Fujita, editors. *Representations of Discrete Functions*. Kluwer Acad., May 1996.

[87] H. Schorr and W. Waite. An efficient machine independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.

[88] R. Shaham, E. Yahav, E. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. In *Static Analysis Symp.*, pages 483–503, June 2003.

[89] M. Sharir. Some observations concerning formal differentiation of set theoretic expressions. *Trans. on Programming Languages and Systems (TOPLAS)*, 4(2):196–225, April 1982.

[90] N. Suzuki. *Automatic Verification of Programs with Complex Data Structures*. PhD thesis, Dept. of Computer Science, Stanford University, February 1976.

[91] O. Tkachuk, M. Dwyer, and C. Pasareanu. Automated environment generation for software model checking. In *Automated Software Engineering*, pages 116–129, October 2003.

[92] R. Topor. The correctness of the Schorr-Waite list marking algorithm. Tech. Report MIP-R-104, School of Artificial Intelligence, University of Edinburgh, July 1974.

[93] TVLA system. Available at http://www.cs.tau.ac.il/ ~tvla/.

[94] B. van Fraassen. Singular terms, truth-value gaps, and free logic. *J. Philosophy*, 63(17):481–495, September 1966.

[95] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Symp. on Principles of Programming Languages*, pages 27–40, January 2001.

[96] E. Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In *Conf. on Programming Language Design and Impl.*, pages 25–34, June 2004.

[97] E. Yahav and M. Sagiv. Automatically verifying concurrent queue algorithms. *Electronic Notes in Theoretical Computer Science*, 89(3), 2003.

[98] H. Yang. *Local Reasoning for Stateful Programs*. PhD thesis, Dept. of Computer Science, University of Illinois, Urbana-Champaign, June 2001.

[99] L. Yelowitz and A. Duncan. Abstractions, instantiations, and proofs of marking algorithms. In *Symp. on Artificial Intelligence and Programming Languages*, pages 13–21, August 1977.

[100] G. Yorsh, T. Reps, M. Sagiv, and R. Wilhelm. Logical characterizations of heap abstractions. To appear in *ACM Transactions on Computational Logic (TOCL)*.

[101] A. Zeller. Yesterday, my program worked. today, it does not. why? In *European Software Engineering Conf. (ESEC/FSE)*, pages 253–267, September 1999.

[102] A. Zeller. Automated debugging: Are we close? *IEEE Computer*, pages 26–31, November 2001.

Appendix A: Correctness of the Finite-Differencing Scheme of Sect. 3.2

The proofs in this section are by induction, using a size measure for formulas based on the process of putting φ in core normal form. Because of the assumption of no circular dependences among the definitions of instrumentation relations, φ can always be put in core normal form by repeated substitution until only core relations remain. The size measure is basically the size of φ when put in core normal form, except that each occurrence of an instrumentation relation $p(w_1, \ldots, w_k)$, $p \in \mathcal{I}$, encountered during the process is counted as being 1 larger than the size measure of $\psi_p\{w_1, \ldots, w_k\}$, the defining formula for relation p with $w_1, \ldots, w_k$ substituted for ψ_p's formal parameters. The proofs, therefore, look like standard structural-induction proofs, except that in the case for $p(w_1, \ldots, w_k)$, $p \in \mathcal{I}$, we are permitted to assume that the induction hypothesis holds for $\psi_p\{w_1, \ldots, w_k\}$.

Recall from Sect. 3.2 that our results are couched in terms of 2-valued logic, but by the Embedding Theorem (Theorem 2.6, [84, Theorem 4.9]), the relation-maintenance formulas that we define provide sound results when interpreted in 3-valued logic.

We only consider first-order formulas because the correctness of the extension of the Finite-Differencing Scheme for Reachability and Transitive Closure of has been argued in Sect. 3.3.

Lemma A.1 (Lemma 3.4) For every formula φ, φ_1, φ_2 and statement st, the following properties hold:[1]

(i) $\Delta^{+}_{st}[\varphi] \overset{\text{meta}}{\Longleftrightarrow} \mathbf{F}_{st}[\varphi] \wedge \neg\varphi$

(ii) $\Delta^{-}_{st}[\varphi] \overset{\text{meta}}{\Longleftrightarrow} \varphi \wedge \neg\mathbf{F}_{st}[\varphi]$

(iii) (a) $\mathbf{F}_{st}[\neg\varphi_1] \overset{\text{meta}}{\Longleftrightarrow} \neg\mathbf{F}_{st}[\varphi_1]$

(b) $\mathbf{F}_{st}[\varphi_1 \vee \varphi_2] \overset{\text{meta}}{\Longleftrightarrow} \mathbf{F}_{st}[\varphi_1] \vee \mathbf{F}_{st}[\varphi_2]$

(c) $\mathbf{F}_{st}[\varphi_1 \wedge \varphi_2] \overset{\text{meta}}{\Longleftrightarrow} \mathbf{F}_{st}[\varphi_1] \wedge \mathbf{F}_{st}[\varphi_2]$

[1] To simplify the presentation, we use $lhs \overset{\text{meta}}{\Longleftrightarrow} rhs$ and $lhs \overset{\text{meta}}{\Longrightarrow} rhs$ as shorthands for $[\![lhs]\!]^S_2(Z) = [\![rhs]\!]^S_2(Z)$ and $[\![lhs]\!]^S_2(Z) \leq [\![rhs]\!]^S_2(Z)$, respectively, for any $S \in$ 2-STRUCT and assignment Z that is complete for lhs and rhs.

(d) $\mathbf{F}_{st}[\exists\, v\colon \varphi_1] \overset{\text{meta}}{\Longleftrightarrow} \exists\, v\colon \mathbf{F}_{st}[\varphi_1]$

(e) $\mathbf{F}_{st}[\forall\, v\colon \varphi_1] \overset{\text{meta}}{\Longleftrightarrow} \forall\, v\colon \mathbf{F}_{st}[\varphi_1]$

Proof *Atomic* For the cases $\varphi \equiv \boldsymbol{l}$, where $\boldsymbol{l} \in \{\mathbf{0}, \mathbf{1}\}$, and $\varphi \equiv (v_1 = v_2)$, $\Delta^{+}_{st}[\varphi] = \Delta^{-}_{st}[\varphi] = 0$, and (i) and (ii) follow immediately.

For $\varphi \equiv p(w_1, \ldots, w_k)$, $p \in \mathcal{C}$, and $\tau_{p,st}$ is of the form $p\,?\,\neg\delta^{-}_{p,st} : \delta^{+}_{p,st}$

(i) $\Delta^{+}_{st}[p(w_1, \ldots, w_k)] \overset{\text{meta}}{\Longleftrightarrow} (\delta^{+}_{p,st} \wedge \neg p)\{w_1, \ldots, w_k\}$

$$\overset{\text{meta}}{\Longleftrightarrow} \left(\begin{array}{l} p(w_1, \ldots, w_k) \\ \quad ?\ \neg(\delta^{-}_{p,st} \wedge p)\{w_1, \ldots, w_k\} \\ \quad :\ (\delta^{+}_{p,st} \wedge \neg p)\{w_1, \ldots, w_k\} \end{array}\right) \wedge \neg p(w_1, \ldots, w_k)$$

$$\overset{\text{meta}}{\Longleftrightarrow} (\mathbf{F}_{st}[p] \wedge \neg p)\{w_1, \ldots, w_k\}$$

(by the definitions of $\mathbf{F}_{st}[\cdot]$, $\Delta^{+}_{st}[\cdot]$, and $\Delta^{-}_{st}[\cdot]$)

(ii) $\Delta^{-}_{st}[p(w_1, \ldots, w_k)] \overset{\text{meta}}{\Longleftrightarrow} (\delta^{-}_{p,st} \wedge p)\{w_1, \ldots, w_k\}$

$$\overset{\text{meta}}{\Longleftrightarrow} p\{w_1, \ldots, w_k\} \wedge \left(\begin{array}{l} p(w_1, \ldots, w_k) \\ \quad ?\ (\delta^{-}_{p,st} \wedge p)\{w_1, \ldots, w_k\} \\ \quad :\ \neg(\delta^{+}_{p,st} \wedge \neg p)\{w_1, \ldots, w_k\} \end{array}\right)$$

$$\overset{\text{meta}}{\Longleftrightarrow} p\{w_1, \ldots, w_k\} \wedge \neg \left(\begin{array}{l} p(w_1, \ldots, w_k) \\ \quad ?\ \neg(\delta^{-}_{p,st} \wedge p)\{w_1, \ldots, w_k\} \\ \quad :\ (\delta^{+}_{p,st} \wedge \neg p)\{w_1, \ldots, w_k\} \end{array}\right)$$

$$\overset{\text{meta}}{\Longleftrightarrow} (p \wedge \neg\mathbf{F}_{st}[p])\{w_1, \ldots, w_k\}$$

(by the definitions of $\mathbf{F}_{st}[\cdot]$, $\Delta^{+}_{st}[\cdot]$, and $\Delta^{-}_{st}[\cdot]$)

For $\varphi \equiv p(w_1, \ldots, w_k)$, $p \in \mathcal{C}$, and $\tau_{p,st}$ is of the form $p \vee \delta_{p,st}$ or $\delta_{p,st} \vee p$

(i) $\Delta^{+}_{st}[p(w_1, \ldots, w_k)] \overset{\text{meta}}{\Longleftrightarrow} (\delta_{p,st} \wedge \neg p)\{w_1, \ldots, w_k\}$

$$\overset{\text{meta}}{\Longleftrightarrow} \begin{array}{ll} & (p(w_1, \ldots, w_k)\ ?\ \neg 0 : (\delta_{p,st} \wedge \neg p)\{w_1, \ldots, w_k\}) \\ \wedge & \neg p(w_1, \ldots, w_k) \end{array}$$

$$\overset{\text{meta}}{\Longleftrightarrow} (\mathbf{F}_{st}[p] \wedge \neg p)\{w_1, \ldots, w_k\}$$

(by the definitions of $\mathbf{F}_{st}[\cdot]$, $\Delta^{+}_{st}[\cdot]$, and $\Delta^{-}_{st}[\cdot]$)

(ii) $\Delta^{-}_{st}[p(w_1, \ldots, w_k)] \overset{\text{meta}}{\Longleftrightarrow} 0$

$$\overset{\text{meta}}{\Longleftrightarrow} p\{w_1, \ldots, w_k\} \wedge \neg p\{w_1, \ldots, w_k\} \wedge (\neg \delta_{p,st} \vee p)\{w_1, \ldots, w_k\}$$

$$\overset{\text{meta}}{\Longleftrightarrow} \begin{array}{ll} & p\{w_1, \ldots, w_k\} \\ \wedge & \neg(p\{w_1, \ldots, w_k\} \vee (\delta_{p,st} \wedge \neg p)\{w_1, \ldots, w_k\}) \end{array}$$

$$\overset{\text{meta}}{\Longleftrightarrow} \begin{array}{ll} & p\{w_1, \ldots, w_k\} \\ \wedge & \neg(p(w_1, \ldots, w_k)\ ?\ \neg 0 : (\delta_{p,st} \wedge \neg p)\{w_1, \ldots, w_k\}) \end{array}$$

$$\overset{\text{meta}}{\Longleftrightarrow} (p \wedge \neg \mathbf{F}_{st}[p])\{w_1, \ldots, w_k\}$$

(by the definitions of $\mathbf{F}_{st}[\cdot]$, $\Delta^{+}_{st}[\cdot]$, and $\Delta^{-}_{st}[\cdot]$)

For $\varphi \equiv p(w_1, \ldots, w_k)$, $p \in \mathcal{C}$, and $\tau_{p,st}$ is of the form $p \wedge \delta_{p,st}$ or $\delta_{p,st} \wedge p$

(i) $\Delta^{+}_{st}[p(w_1, \ldots, w_k)] \overset{\text{meta}}{\Longleftrightarrow} 0$

$$\overset{\text{meta}}{\Longleftrightarrow} p\{w_1, \ldots, w_k\} \wedge (\delta_{p,st} \vee \neg p)\{w_1, \ldots, w_k\} \wedge \neg p\{w_1, \ldots, w_k\}$$

$$\overset{\text{meta}}{\Longleftrightarrow} \begin{array}{ll} & (p(w_1, \ldots, w_k)\ ?\ (\delta_{p,st} \vee \neg p)\{w_1, \ldots, w_k\} : 0) \\ \wedge & \neg p\{w_1, \ldots, w_k\} \end{array}$$

$$\overset{\text{meta}}{\Longleftrightarrow} \begin{array}{ll} & (p(w_1, \ldots, w_k)\ ?\ \neg(\neg \delta_{p,st} \wedge p)\{w_1, \ldots, w_k\} : 0) \\ \wedge & \neg p\{w_1, \ldots, w_k\} \end{array}$$

$$\overset{\text{meta}}{\Longleftrightarrow} (\mathbf{F}_{st}[p] \wedge \neg p)\{w_1, \ldots, w_k\}$$

(by the definitions of $\mathbf{F}_{st}[\cdot]$, $\Delta^{+}_{st}[\cdot]$, and $\Delta^{-}_{st}[\cdot]$)

(ii) $\Delta^{-}_{st}[p(w_1,\ldots,w_k)] \overset{\text{meta}}{\Longleftrightarrow} (\neg\delta_{p,st} \wedge p)\{w_1,\ldots,w_k\}$

$$\overset{\text{meta}}{\Longleftrightarrow} \begin{array}{ll} & p\{w_1,\ldots,w_k\} \\ \wedge & (p(w_1,\ldots,w_k)\ ?\ (\neg\delta_{p,st} \wedge p)\{w_1,\ldots,w_k\} : 1) \end{array}$$

$$\overset{\text{meta}}{\Longleftrightarrow} \begin{array}{ll} & p\{w_1,\ldots,w_k\} \\ \wedge & \neg(p(w_1,\ldots,w_k)\ ?\ \neg(\neg\delta_{p,st} \wedge p)\{w_1,\ldots,w_k\} : 0) \end{array}$$

$$\overset{\text{meta}}{\Longleftrightarrow} (p \wedge \neg\mathbf{F}_{st}[p])\{w_1,\ldots,w_k\}$$

(by the definitions of $\mathbf{F}_{st}[\cdot]$, $\Delta^{+}_{st}[\cdot]$, and $\Delta^{-}_{st}[\cdot]$)

For $\varphi \equiv p(w_1,\ldots,w_k)$, $p \in \mathcal{C}$, but $\tau_{p,st}$ is not of the above forms

(i) $\Delta^{+}_{st}[p(w_1,\ldots,w_k)] \overset{\text{meta}}{\Longleftrightarrow} (\tau_{p,st} \wedge \neg p)\{w_1,\ldots,w_k\}$

$$\overset{\text{meta}}{\Longleftrightarrow} \begin{pmatrix} p(w_1,\ldots,w_k) \\ \quad ?\ \neg(p \wedge \tau_{p,st})\{w_1,\ldots,w_k\} \\ \quad :\ (\tau_{p,st} \wedge \neg p)\{w_1,\ldots,w_k\} \end{pmatrix} \wedge \neg p(w_1,\ldots,w_k)$$

$$\overset{\text{meta}}{\Longleftrightarrow} (\mathbf{F}_{st}[p] \wedge \neg p)\{w_1,\ldots,w_k\}$$

(by the definitions of $\mathbf{F}_{st}[\cdot]$, $\Delta^{+}_{st}[\cdot]$, and $\Delta^{-}_{st}[\cdot]$)

(ii) $\Delta^{-}_{st}[p(w_1,\ldots,w_k)] \overset{\text{meta}}{\Longleftrightarrow} (p \wedge \neg\tau_{p,st})\{w_1,\ldots,w_k\}$

$$\overset{\text{meta}}{\Longleftrightarrow} p\{w_1,\ldots,w_k\} \wedge \begin{pmatrix} p(w_1,\ldots,w_k) \\ \quad ?\ (p \wedge \neg\tau_{p,st})\{w_1,\ldots,w_k\} \\ \quad :\ \neg(\tau_{p,st} \wedge \neg p)\{w_1,\ldots,w_k\} \end{pmatrix}$$

$$\overset{\text{meta}}{\Longleftrightarrow} p\{w_1,\ldots,w_k\} \wedge \neg \begin{pmatrix} p(w_1,\ldots,w_k) \\ \quad ?\ \neg(p \wedge \neg\tau_{p,st})\{w_1,\ldots,w_k\} \\ \quad :\ (\tau_{p,st} \wedge \neg p)\{w_1,\ldots,w_k\} \end{pmatrix}$$

$$\overset{\text{meta}}{\Longleftrightarrow} (p \wedge \neg\mathbf{F}_{st}[p])\{w_1,\ldots,w_k\}$$

(by the definitions of $\mathbf{F}_{st}[\cdot]$, $\Delta^{+}_{st}[\cdot]$, and $\Delta^{-}_{st}[\cdot]$)

For $p(w_1, \ldots, w_k)$, $p \in \mathcal{I}$,

$$\begin{aligned}
\text{(i)}\ \ \Delta^{+}_{st}[p(w_1, \ldots, w_k)] &\overset{\text{meta}}{\iff} \Delta^{+}_{st}[\psi_p\{w_1, \ldots, w_k\}] \\
&\overset{\text{meta}}{\iff} \mathbf{F}_{st}[\psi_p\{w_1, \ldots, w_k\}] \wedge \neg\psi_p\{w_1, \ldots, w_k\} \\
&\qquad \text{(by inductive hypothesis (i) for } \psi_p\text{)} \\
&\overset{\text{meta}}{\iff} (\mathbf{F}_{st}[p] \wedge \neg p)\{w_1, \ldots, w_k\} \\
&\qquad (\psi_p \text{ is the defining formula for } p)
\end{aligned}$$

$$\begin{aligned}
\text{(ii)}\ \ \Delta^{-}_{st}[p(w_1, \ldots, w_k)] &\overset{\text{meta}}{\iff} \Delta^{-}_{st}[\psi_p\{w_1, \ldots, w_k\}] \\
&\overset{\text{meta}}{\iff} \psi_p\{w_1, \ldots, w_k\} \wedge \neg\mathbf{F}_{st}[\psi_p\{w_1, \ldots, w_k\}] \\
&\qquad \text{(by inductive hypothesis (ii) for } \psi_p\text{)} \\
&\overset{\text{meta}}{\iff} (p \wedge \neg\mathbf{F}_{st}[p])\{w_1, \ldots, w_k\} \\
&\qquad (\psi_p \text{ is the defining formula for } p)
\end{aligned}$$

Not $\varphi \equiv \neg\varphi_1$.

$$\begin{aligned}
\text{(i)}\ \ \Delta^{+}_{st}[\neg\varphi_1] &\overset{\text{meta}}{\iff} \Delta^{-}_{st}[\varphi_1] \\
&\overset{\text{meta}}{\iff} \varphi_1 \wedge \neg\mathbf{F}_{st}[\varphi_1] && \text{(by inductive hypothesis (ii) for } \varphi_1\text{)} \\
&\overset{\text{meta}}{\iff} \mathbf{F}_{st}[\neg\varphi_1] \wedge \neg(\neg\varphi_1) && \text{(by inductive hypothesis (iii) for } \varphi_1\text{)}
\end{aligned}$$

$$\begin{aligned}
\text{(ii)}\ \ \Delta^{-}_{st}[\neg\varphi_1] &\overset{\text{meta}}{\iff} \Delta^{+}_{st}[\varphi_1] \\
&\overset{\text{meta}}{\iff} \mathbf{F}_{st}[\varphi_1] \wedge \neg\varphi_1 && \text{(by inductive hypothesis (i) for } \varphi_1\text{)} \\
&\overset{\text{meta}}{\iff} (\neg\varphi_1) \wedge \neg\neg\mathbf{F}_{st}[\varphi_1] \\
&\overset{\text{meta}}{\iff} (\neg\varphi_1) \wedge \neg\mathbf{F}_{st}[\neg\varphi_1] && \text{(by inductive hypothesis (iii) for } \varphi_1\text{)}
\end{aligned}$$

(iii) $\mathbf{F}_{st}[\neg\varphi_1] \overset{\text{meta}}{\iff} (\neg\varphi_1)\ ?\ \neg\Delta^{-}_{st}[\neg\varphi_1] : \Delta^{+}_{st}[\neg\varphi_1]$

$\overset{\text{meta}}{\iff} \varphi_1\ ?\ \Delta^{+}_{st}[\neg\varphi_1] : \neg\Delta^{-}_{st}[\neg\varphi_1]$

$\overset{\text{meta}}{\iff} \varphi_1\ ?\ \Delta^{-}_{st}[\varphi_1] : \neg\Delta^{+}_{st}[\varphi_1]$ (by the definitions of $\Delta^{+}_{st}[\cdot]$ and $\Delta^{-}_{st}[\cdot]$)

$\overset{\text{meta}}{\iff} \neg(\varphi_1\ ?\ \neg\Delta^{-}_{st}[\varphi_1] : \Delta^{+}_{st}[\varphi_1])$

$\overset{\text{meta}}{\iff} \neg\mathbf{F}_{st}[\varphi_1]$

Or $\varphi \equiv \varphi_1 \vee \varphi_2$.

(i) $\Delta^{+}_{st}[\varphi_1 \vee \varphi_2] \overset{\text{meta}}{\iff} (\Delta^{+}_{st}[\varphi_1] \wedge \neg\varphi_2) \vee (\neg\varphi_1 \wedge \Delta^{+}_{st}[\varphi_2])$

$\overset{\text{meta}}{\iff} (\mathbf{F}_{st}[\varphi_1] \wedge \neg\varphi_1 \wedge \neg\varphi_2) \vee (\neg\varphi_1 \wedge \mathbf{F}_{st}[\varphi_2] \wedge \neg\varphi_2)$

(by inductive hypothesis (i) for φ_1 and φ_2)

$\overset{\text{meta}}{\iff} (\mathbf{F}_{st}[\varphi_1] \vee \mathbf{F}_{st}[\varphi_2]) \wedge (\neg\varphi_1 \wedge \neg\varphi_2)$

$\overset{\text{meta}}{\iff} (\mathbf{F}_{st}[\varphi_1 \vee \varphi_2]) \wedge \neg(\varphi_1 \vee \varphi_2)$

(by part (iii) for $\varphi_1 \vee \varphi_2$, proved independently below)

(ii) $\Delta^{-}_{st}[\varphi_1 \vee \varphi_2] \overset{\text{meta}}{\iff} (\Delta^{-}_{st}[\varphi_1] \wedge \neg\mathbf{F}_{st}[\varphi_2]) \vee (\neg\mathbf{F}_{st}[\varphi_1] \wedge \Delta^{-}_{st}[\varphi_2])$

$\overset{\text{meta}}{\iff} (\varphi_1 \wedge \neg\mathbf{F}_{st}[\varphi_1] \wedge \neg\mathbf{F}_{st}[\varphi_2]) \vee (\neg\mathbf{F}_{st}[\varphi_1] \wedge \neg\mathbf{F}_{st}[\varphi_2] \wedge \varphi_2)$

(by inductive hypothesis (ii) for φ_1 and φ_2)

$\overset{\text{meta}}{\iff} (\varphi_1 \vee \varphi_2) \wedge (\neg\mathbf{F}_{st}[\varphi_1] \wedge \neg\mathbf{F}_{st}[\varphi_2])$

$\overset{\text{meta}}{\iff} (\varphi_1 \vee \varphi_2) \wedge \neg(\mathbf{F}_{st}[\varphi_1] \vee \mathbf{F}_{st}[\varphi_2])$

$\overset{\text{meta}}{\iff} (\varphi_1 \vee \varphi_2) \wedge \neg\mathbf{F}_{st}[\varphi_1 \vee \varphi_2]$

(by part (iii) for $\varphi_1 \vee \varphi_2$, proved independently below)

$$
\begin{aligned}
\text{(iii)}\ \mathbf{F}_{st}[\varphi_1 \vee \varphi_2] &\overset{\text{meta}}{\Longleftrightarrow} (\varphi_1 \vee \varphi_2)\ ?\ \neg\Delta^{-}_{st}[\varphi_1 \vee \varphi_2] : \Delta^{+}_{st}[\varphi_1 \vee \varphi_2] \\
&\overset{\text{meta}}{\Longleftrightarrow} (\varphi_1 \vee \varphi_2) \\
&\qquad ?\ \neg\left[(\Delta^{-}_{st}[\varphi_1] \wedge \neg\mathbf{F}_{st}[\varphi_2]) \vee (\neg\mathbf{F}_{st}[\varphi_1] \wedge \Delta^{-}_{st}[\varphi_2])\right] \\
&\qquad :\ (\Delta^{+}_{st}[\varphi_1] \wedge \neg\varphi_2) \vee (\neg\varphi_1 \wedge \Delta^{+}_{st}[\varphi_2]) \\
&\qquad \text{(by the definitions of } \Delta^{+}_{st}[\cdot] \text{ and } \Delta^{-}_{st}[\cdot]) \\
&\overset{\text{meta}}{\Longleftrightarrow} \left(\begin{array}{ll} & (\varphi_1 \vee \varphi_2) \\ \wedge & \neg\left[(\Delta^{-}_{st}[\varphi_1] \wedge \neg\mathbf{F}_{st}[\varphi_2]) \vee (\neg\mathbf{F}_{st}[\varphi_1] \wedge \Delta^{-}_{st}[\varphi_2])\right] \end{array}\right) \\
&\qquad \vee \left(\begin{array}{ll} & \neg(\varphi_1 \vee \varphi_2) \\ \wedge & \left[(\Delta^{+}_{st}[\varphi_1] \wedge \neg\varphi_2) \vee (\neg\varphi_1 \wedge \Delta^{+}_{st}[\varphi_2])\right] \end{array}\right) \\
&\overset{\text{meta}}{\Longleftrightarrow} \left(\begin{array}{ll} & (\varphi_1 \vee \varphi_2) \\ \wedge & \neg\left[(\Delta^{-}_{st}[\varphi_1] \wedge \neg\mathbf{F}_{st}[\varphi_2]) \vee (\neg\mathbf{F}_{st}[\varphi_1] \wedge \Delta^{-}_{st}[\varphi_2])\right] \end{array}\right) \\
&\qquad \vee \quad (\neg\varphi_1 \wedge \Delta^{+}_{st}[\varphi_1] \wedge \neg\varphi_2) \vee (\neg\varphi_1 \wedge \Delta^{+}_{st}[\varphi_2] \wedge \neg\varphi_2) \\
&\overset{\text{meta}}{\Longleftrightarrow} \left\{\begin{array}{ll} & \left(\begin{array}{ll} & (\varphi_1 \vee \varphi_2) \\ \wedge & (\neg\Delta^{-}_{st}[\varphi_1] \vee \mathbf{F}_{st}[\varphi_2]) \\ \wedge & (\mathbf{F}_{st}[\varphi_1] \vee \neg\Delta^{-}_{st}[\varphi_2]) \end{array}\right) \\ \vee & (\neg\varphi_1 \wedge \Delta^{+}_{st}[\varphi_1] \wedge \neg\varphi_2) \vee (\neg\varphi_1 \wedge \Delta^{+}_{st}[\varphi_2] \wedge \neg\varphi_2) \end{array}\right. \\
&\overset{\text{meta}}{\Longleftrightarrow} \left\{\begin{array}{ll} & \varphi_1 \wedge \neg\Delta^{-}_{st}[\varphi_1] \wedge \mathbf{F}_{st}[\varphi_1] \\ \vee & \varphi_1 \wedge \neg\Delta^{-}_{st}[\varphi_1] \wedge \neg\Delta^{-}_{st}[\varphi_2] \\ \vee & \varphi_1 \wedge \mathbf{F}_{st}[\varphi_2] \wedge \mathbf{F}_{st}[\varphi_1] \\ \vee & \varphi_1 \wedge \mathbf{F}_{st}[\varphi_2] \wedge \neg\Delta^{-}_{st}[\varphi_2] \\ \vee & \varphi_2 \wedge \neg\Delta^{-}_{st}[\varphi_1] \wedge \mathbf{F}_{st}[\varphi_1] \\ \vee & \varphi_2 \wedge \neg\Delta^{-}_{st}[\varphi_1] \wedge \neg\Delta^{-}_{st}[\varphi_2] \\ \vee & \varphi_2 \wedge \mathbf{F}_{st}[\varphi_2] \wedge \mathbf{F}_{st}[\varphi_1] \\ \vee & \varphi_2 \wedge \mathbf{F}_{st}[\varphi_2] \wedge \neg\Delta^{-}_{st}[\varphi_2] \\ \vee & \neg\varphi_1 \wedge \Delta^{+}_{st}[\varphi_1] \wedge \neg\varphi_2 \\ \vee & \neg\varphi_1 \wedge \Delta^{+}_{st}[\varphi_2] \wedge \neg\varphi_2 \end{array}\right.
\end{aligned}
\tag{A.1}
$$

We consider the direction $\mathbf{F}_{st}[\varphi_1 \vee \varphi_2] \overset{\text{meta}}{\Longrightarrow} \mathbf{F}_{st}[\varphi_1] \vee \mathbf{F}_{st}[\varphi_2]$ first. We consider the ten cases that correspond to the cases that (at least) one of the ten disjuncts of Formula (A.1) holds. Each case that concerns a disjunct that contains $\mathbf{F}_{st}[\varphi_1]$ or $\mathbf{F}_{st}[\varphi_2]$ as a conjunct trivially implies that $\mathbf{F}_{st}[\varphi_1] \vee \mathbf{F}_{st}[\varphi_2]$ holds. We consider the remaining four cases.

$$\begin{aligned}
\varphi_1 \wedge \neg\Delta^{-}_{st}[\varphi_1] \wedge \neg\Delta^{-}_{st}[\varphi_2] &\overset{\text{meta}}{\Longrightarrow} \varphi_1 \wedge \neg\Delta^{-}_{st}[\varphi_1] \\
&\overset{\text{meta}}{\Longrightarrow} \varphi_1 \wedge \neg\Delta^{-}_{st}[\varphi_1] \vee \neg\varphi_1 \wedge \Delta^{+}_{st}[\varphi_1] \\
&\overset{\text{meta}}{\Longleftrightarrow} \mathbf{F}_{st}[\varphi_1] \qquad \text{(by the definition of } \mathbf{F}_{st}[\cdot]\text{)}
\end{aligned}$$

$$\begin{aligned}
\varphi_2 \wedge \neg\Delta^{-}_{st}[\varphi_1] \wedge \neg\Delta^{-}_{st}[\varphi_2] &\overset{\text{meta}}{\Longrightarrow} \varphi_2 \wedge \neg\Delta^{-}_{st}[\varphi_2] \\
&\overset{\text{meta}}{\Longrightarrow} \varphi_2 \wedge \neg\Delta^{-}_{st}[\varphi_2] \vee \neg\varphi_2 \wedge \Delta^{+}_{st}[\varphi_2] \\
&\overset{\text{meta}}{\Longleftrightarrow} \mathbf{F}_{st}[\varphi_2] \qquad \text{(by the definition of } \mathbf{F}_{st}[\cdot]\text{)}
\end{aligned}$$

$$\begin{aligned}
\neg\varphi_1 \wedge \Delta^{+}_{st}[\varphi_1] \wedge \neg\varphi_2 &\overset{\text{meta}}{\Longrightarrow} \neg\varphi_1 \wedge \Delta^{+}_{st}[\varphi_1] \\
&\overset{\text{meta}}{\Longrightarrow} \varphi_1 \wedge \neg\Delta^{-}_{st}[\varphi_1] \vee \neg\varphi_1 \wedge \Delta^{+}_{st}[\varphi_1] \\
&\overset{\text{meta}}{\Longleftrightarrow} \mathbf{F}_{st}[\varphi_1] \qquad \text{(by the definition of } \mathbf{F}_{st}[\cdot]\text{)}
\end{aligned}$$

$$\begin{aligned}
\neg\varphi_1 \wedge \Delta^{+}_{st}[\varphi_2] \wedge \neg\varphi_2 &\overset{\text{meta}}{\Longrightarrow} \neg\varphi_2 \wedge \Delta^{+}_{st}[\varphi_2] \\
&\overset{\text{meta}}{\Longrightarrow} \varphi_2 \wedge \neg\Delta^{-}_{st}[\varphi_2] \vee \neg\varphi_2 \wedge \Delta^{+}_{st}[\varphi_2] \\
&\overset{\text{meta}}{\Longleftrightarrow} \mathbf{F}_{st}[\varphi_2] \qquad \text{(by the definition of } \mathbf{F}_{st}[\cdot]\text{)}
\end{aligned}$$

We consider the direction $\mathbf{F}_{st}[\varphi_1] \vee \mathbf{F}_{st}[\varphi_2] \overset{\text{meta}}{\Longrightarrow} \mathbf{F}_{st}[\varphi_1 \vee \varphi_2]$ next. Without loss of generality, assume that $\mathbf{F}_{st}[\varphi_1]$ holds. We consider two cases: $\varphi_1 \wedge \neg\Delta^{-}_{st}[\varphi_1]$ holds; $\neg\varphi_1 \wedge \Delta^{+}_{st}[\varphi_1]$ holds. We show that both cases imply that a disjunct of Formula (A.1) holds. If a disjunct of Formula (A.1) holds, then $\mathbf{F}_{st}[\varphi_1 \vee \varphi_2]$ must hold because the latter holds if and only if Formula (A.1) holds. First, assume that $\varphi_1 \wedge \neg\Delta^{-}_{st}[\varphi_1]$ holds.

$$\begin{aligned}
\varphi_1 \wedge \neg\Delta^{-}_{st}[\varphi_1] &\overset{\text{meta}}{\Longleftrightarrow} \varphi_1 \wedge \neg\Delta^{-}_{st}[\varphi_1] \wedge \mathbf{F}_{st}[\varphi_1] \qquad \text{(by inductive hypothesis (ii) for } \varphi_1\text{)} \\
&\overset{\text{meta}}{\Longrightarrow} \mathbf{F}_{st}[\varphi_1 \vee \varphi_2] \qquad \text{(the RHS above is a disjunct of Formula (A.1))}
\end{aligned}$$

Now, assume that $\neg\varphi_1 \wedge \Delta^+_{st}[\varphi_1]$ holds. We consider two subcases: φ_2 holds; $\neg\varphi_2$ holds. Assume that φ_2 holds.

$$\begin{aligned}
\varphi_2 \wedge \neg\varphi_1 \wedge \Delta^+_{st}[\varphi_1] &\overset{\text{meta}}{\Longleftrightarrow} \varphi_2 \wedge \neg\Delta^-_{st}[\varphi_1] \wedge \neg\varphi_1 \wedge \Delta^+_{st}[\varphi_1] \\
&\quad (\neg\varphi_1 \overset{\text{meta}}{\Longrightarrow} \neg\Delta^-_{st}[\varphi_1] \text{ by inductive hypothesis (ii))} \\
&\overset{\text{meta}}{\Longrightarrow} \varphi_2 \wedge \neg\Delta^-_{st}[\varphi_1] \wedge \mathbf{F}_{st}[\varphi_1] \qquad \text{(by the definition of } \mathbf{F}_{st}[\cdot]) \\
&\overset{\text{meta}}{\Longrightarrow} \mathbf{F}_{st}[\varphi_1 \vee \varphi_2] \quad \text{(the RHS above is a disjunct of Formula (A.1))}
\end{aligned}$$

If $\neg\varphi_2$ holds (the second subcase), the result is immediate; it implies that the following disjunct of Formula (A.1) holds: $\neg\varphi_1 \wedge \Delta^+_{st}[\varphi_1] \wedge \neg\varphi_2$.

And $\varphi \equiv \varphi_1 \wedge \varphi_2$. The entries for $\Delta^+_{st}[\varphi_1 \wedge \varphi_2]$ and $\Delta^-_{st}[\varphi_1 \wedge \varphi_2]$ can be derived from those for $\Delta^+_{st}[\varphi_1 \vee \varphi_2]$, $\Delta^-_{st}[\varphi_1 \vee \varphi_2]$, $\Delta^+_{st}[\neg\varphi_1]$, and $\Delta^-_{st}[\neg\varphi_1]$.

$$\begin{aligned}
\Delta^+_{st}[\varphi_1 \wedge \varphi_2] &\overset{\text{meta}}{\Longleftrightarrow} \Delta^+_{st}[\neg(\neg\varphi_1 \vee \neg\varphi_2)] \\
&\overset{\text{meta}}{\Longleftrightarrow} \Delta^-_{st}[\neg\varphi_1 \vee \neg\varphi_2] \qquad \text{(by the definition of } \Delta^+_{st}[\cdot]) \\
&\overset{\text{meta}}{\Longleftrightarrow} (\Delta^-_{st}[\neg\varphi_1] \wedge \neg\mathbf{F}_{st}[\neg\varphi_2]) \vee (\neg\mathbf{F}_{st}[\neg\varphi_1] \wedge \Delta^-_{st}[\neg\varphi_2]) \\
&\quad \text{(by the definition of } \Delta^-_{st}[\cdot]) \\
&\overset{\text{meta}}{\Longleftrightarrow} (\Delta^+_{st}[\varphi_1] \wedge \mathbf{F}_{st}[\varphi_2]) \vee (\mathbf{F}_{st}[\varphi_1] \wedge \Delta^+_{st}[\varphi_2]) \\
&\quad \text{(by the definition of } \Delta^-_{st}[\cdot] \text{ and inductive hypothesis (iii))}
\end{aligned}$$

$$\begin{aligned}
\Delta^-_{st}[\varphi_1 \wedge \varphi_2] &\overset{\text{meta}}{\Longleftrightarrow} \Delta^-_{st}[\neg(\neg\varphi_1 \vee \neg\varphi_2)] \\
&\overset{\text{meta}}{\Longleftrightarrow} \Delta^+_{st}[\neg\varphi_1 \vee \neg\varphi_2] \qquad \text{(by the definition of } \Delta^-_{st}[\cdot]) \\
&\overset{\text{meta}}{\Longleftrightarrow} (\Delta^+_{st}[\neg\varphi_1] \wedge \neg(\neg\varphi_2)) \vee (\neg(\neg\varphi_1) \wedge \Delta^+_{st}[\neg\varphi_2]) \\
&\quad \text{(by the definition of } \Delta^+_{st}[\cdot]) \\
&\overset{\text{meta}}{\Longleftrightarrow} (\Delta^-_{st}[\varphi_1] \wedge \varphi_2) \vee (\varphi_1 \wedge \Delta^-_{st}[\varphi_2]) \qquad \text{(by the definition of } \Delta^+_{st}[\cdot])
\end{aligned}$$

Exists $\varphi \equiv \exists\, v_1 \colon \varphi_1$.

$$
\begin{aligned}
\text{(i)}\ \ \Delta^{+}_{st}[\exists\, v_1 \colon \varphi_1] &\overset{\text{meta}}{\iff} (\exists\, v_1 \colon \Delta^{+}_{st}[\varphi_1]) \wedge \neg(\exists\, v_1 \colon \varphi_1) \qquad \text{(by the definition of } \Delta^{+}_{st}[\cdot]) \\
&\overset{\text{meta}}{\iff} (\exists\, v_1 \colon \Delta^{+}_{st}[\varphi_1]) \wedge \neg(\exists\, v_1 \colon \varphi_1) \wedge \neg(\exists\, v_1 \colon \varphi_1) \\
&\overset{\text{meta}}{\iff} \left(\begin{array}{l} (\exists\, v_1 \colon \varphi_1) \\ \quad ?\ \neg\left[(\exists\, v_1 \colon \Delta^{-}_{st}[\varphi_1]) \wedge \neg(\exists\, v_1 \colon \mathbf{F}_{st}[\varphi_1])\right] \\ \quad :\ (\exists\, v_1 \colon \Delta^{+}_{st}[\varphi_1]) \wedge \neg(\exists\, v_1 \colon \varphi_1) \end{array} \right) \wedge \neg(\exists\, v_1 \colon \varphi_1) \\
&\overset{\text{meta}}{\iff} \left[(\exists\, v_1 \colon \varphi_1)\ ?\ \neg\Delta^{-}_{st}[(\exists\, v_1 \colon \varphi_1)] : \Delta^{+}_{st}[(\exists\, v_1 \colon \varphi_1)]\right] \wedge \neg(\exists\, v_1 \colon \varphi_1) \\
&\qquad \text{(by the definitions of } \Delta^{-}_{st}[\cdot] \text{ and } \Delta^{+}_{st}[\cdot]) \\
&\overset{\text{meta}}{\iff} \mathbf{F}_{st}[\exists\, v_1 \colon \varphi_1] \wedge \neg(\exists\, v_1 \colon \varphi_1) \qquad \text{(by the definition of } \mathbf{F}_{st}[\cdot])
\end{aligned}
$$

$$
\begin{aligned}
\text{(ii)}\ \ \Delta^{-}_{st}[\exists\, v_1 \colon \varphi_1] &\overset{\text{meta}}{\iff} (\exists\, v_1 \colon \Delta^{-}_{st}[\varphi_1]) \wedge \neg(\exists\, v_1 \colon \mathbf{F}_{st}[\varphi_1]) \qquad \text{(by the definition of } \Delta^{-}_{st}[\cdot]) \\
&\overset{\text{meta}}{\iff} (\exists\, v_1 \colon \varphi_1) \wedge (\exists\, v_1 \colon \Delta^{-}_{st}[\varphi_1]) \wedge \neg(\exists\, v_1 \colon \mathbf{F}_{st}[\varphi_1]) \\
&\qquad ((\exists\, v_1 \colon \Delta^{-}_{st}[\varphi_1]) \overset{\text{meta}}{\implies} (\exists\, v_1 \colon \varphi_1) \ \text{ by inductive hypothesis (ii))} \\
&\overset{\text{meta}}{\iff} (\exists\, v_1 \colon \varphi_1) \wedge \left(\begin{array}{l} (\exists\, v_1 \colon \varphi_1) \\ \quad ?\ (\exists\, v_1 \colon \Delta^{-}_{st}[\varphi_1]) \wedge \neg(\exists\, v_1 \colon \mathbf{F}_{st}[\varphi_1]) \\ \quad :\ \neg\left[(\exists\, v_1 \colon \Delta^{+}_{st}[\varphi_1]) \wedge \neg(\exists\, v_1 \colon \varphi_1)\right] \end{array} \right) \\
&\overset{\text{meta}}{\iff} (\exists\, v_1 \colon \varphi_1) \wedge \neg \left(\begin{array}{l} (\exists\, v_1 \colon \varphi_1) \\ \quad ?\ \neg\left[(\exists\, v_1 \colon \Delta^{-}_{st}[\varphi_1]) \wedge \neg(\exists\, v_1 \colon \mathbf{F}_{st}[\varphi_1])\right] \\ \quad :\ (\exists\, v_1 \colon \Delta^{+}_{st}[\varphi_1]) \wedge \neg(\exists\, v_1 \colon \varphi_1) \end{array} \right) \\
&\overset{\text{meta}}{\iff} (\exists\, v_1 \colon \varphi_1) \wedge \neg\left[(\exists\, v_1 \colon \varphi_1)\ ?\ \neg\Delta^{-}_{st}[(\exists\, v_1 \colon \varphi_1)] : \Delta^{+}_{st}[(\exists\, v_1 \colon \varphi_1)]\right] \\
&\qquad \text{(by the definitions of } \Delta^{-}_{st}[\cdot] \text{ and } \Delta^{+}_{st}[\cdot]) \\
&\overset{\text{meta}}{\iff} (\exists\, v_1 \colon \varphi_1) \wedge \neg\mathbf{F}_{st}[\exists\, v_1 \colon \varphi_1] \qquad \text{(by the definition of } \mathbf{F}_{st}[\cdot])
\end{aligned}
$$

(iii) We consider the direction $\mathbf{F}_{st}[\exists\, v_1 \colon \varphi_1] \overset{\text{meta}}{\Longrightarrow} \exists\, v_1 \colon \mathbf{F}_{st}[\varphi_1]$ first.

$$\begin{aligned}
\mathbf{F}_{st}[\exists\, v_1 \colon \varphi_1] &\overset{\text{meta}}{\Longleftrightarrow} (\exists\, v_1 \colon \varphi_1)\ ?\ \neg\Delta^{-}_{st}[(\exists\, v_1 \colon \varphi_1)] : \Delta^{+}_{st}[(\exists\, v_1 \colon \varphi_1)] \\
&\overset{\text{meta}}{\Longleftrightarrow} \begin{cases} (\exists\, v_1 \colon \varphi_1) \\ \quad ?\ \neg\left[(\exists\, v_1 \colon \Delta^{-}_{st}[\varphi_1]) \wedge \neg(\exists\, v_1 \colon \mathbf{F}_{st}[\varphi_1])\right] \\ \quad :\ (\exists\, v_1 \colon \Delta^{+}_{st}[\varphi_1]) \wedge \neg(\exists\, v_1 \colon \varphi_1) \end{cases} \\
&\qquad \text{(by the definitions of } \Delta^{-}_{st}[\cdot] \text{ and } \Delta^{+}_{st}[\cdot]) \\
&\overset{\text{meta}}{\Longleftrightarrow} \begin{cases} \quad (\exists\, v_1 \colon \varphi_1) \wedge \neg(\exists\, v_1 \colon \Delta^{-}_{st}[\varphi_1]) \\ \vee \quad (\exists\, v_1 \colon \varphi_1) \wedge (\exists\, v_1 \colon \mathbf{F}_{st}[\varphi_1]) \\ \vee \quad \neg(\exists\, v_1 \colon \varphi_1) \wedge (\exists\, v_1 \colon \Delta^{+}_{st}[\varphi_1]) \end{cases} \qquad \text{(A.2)}
\end{aligned}$$

We consider the three cases that correspond to the cases that (at least) one of the three disjuncts of Formula (A.2) holds. The case that concerns the middle disjunct, which contains $(\exists\, v_1 \colon \mathbf{F}_{st}[\varphi_1])$ as a conjunct, is immediate. We consider the remaining two cases. First, assume that $(\exists\, v_1 \colon \varphi_1) \wedge \neg(\exists\, v_1 \colon \Delta^{-}_{st}[\varphi_1])$ holds.

$$\begin{aligned}
(\exists\, v_1 \colon \varphi_1) \wedge \neg(\exists\, v_1 \colon \Delta^{-}_{st}[\varphi_1]) &\overset{\text{meta}}{\Longrightarrow} \exists\, v_1 \colon (\varphi_1 \wedge \neg\Delta^{-}_{st}[\varphi_1]) \\
&\overset{\text{meta}}{\Longrightarrow} \exists\, v_1 \colon (\varphi_1\ ?\ \neg\Delta^{-}_{st}[\varphi_1] : \Delta^{+}_{st}[\varphi_1]) \\
&\overset{\text{meta}}{\Longleftrightarrow} \exists\, v_1 \colon \mathbf{F}_{st}[\varphi_1] \qquad \text{(by the definition of } \mathbf{F}_{st}[\cdot])
\end{aligned}$$

Now, assume that $\neg(\exists\, v_1 \colon \varphi_1) \wedge (\exists\, v_1 \colon \Delta^{+}_{st}[\varphi_1])$ holds.

$$\begin{aligned}
\neg(\exists\, v_1 \colon \varphi_1) \wedge (\exists\, v_1 \colon \Delta^{+}_{st}[\varphi_1]) &\overset{\text{meta}}{\Longrightarrow} \exists\, v_1 \colon (\neg\varphi_1 \wedge \Delta^{+}_{st}[\varphi_1]) \\
&\overset{\text{meta}}{\Longrightarrow} \exists\, v_1 \colon (\varphi_1\ ?\ \neg\Delta^{-}_{st}[\varphi_1] : \Delta^{+}_{st}[\varphi_1]) \\
&\overset{\text{meta}}{\Longleftrightarrow} \exists\, v_1 \colon \mathbf{F}_{st}[\varphi_1] \qquad \text{(by the definition of } \mathbf{F}_{st}[\cdot])
\end{aligned}$$

We consider the direction $\exists\, v_1 \colon \mathbf{F}_{st}[\varphi_1] \overset{\text{meta}}{\Longrightarrow} \mathbf{F}_{st}[\exists\, v_1 \colon \varphi_1]$ next.

$$\begin{aligned}
\exists\, v_1 \colon \mathbf{F}_{st}[\varphi_1] &\overset{\text{meta}}{\Longleftrightarrow} \exists\, v_1 \colon (\varphi_1\ ?\ \neg\Delta^{-}_{st}[\varphi_1] : \Delta^{+}_{st}[\varphi_1]) \qquad \text{(by the definition of } \mathbf{F}_{st}[\cdot]) \\
&\overset{\text{meta}}{\Longleftrightarrow} \exists\, v_1 \colon (\varphi_1 \wedge \neg\Delta^{-}_{st}[\varphi_1] \vee \neg\varphi_1 \wedge \Delta^{+}_{st}[\varphi_1]) \\
&\overset{\text{meta}}{\Longleftrightarrow} (\exists\, v_1 \colon \varphi_1 \wedge \neg\Delta^{-}_{st}[\varphi_1]) \vee (\exists\, v_1 \colon \neg\varphi_1 \wedge \Delta^{+}_{st}[\varphi_1]) \qquad \text{(A.3)}
\end{aligned}$$

We consider the two cases that correspond to the cases that (at least) one of the two disjuncts of Formula (A.3) holds. First, assume that $(\exists\, v_1 \colon \varphi_1 \wedge \neg\Delta^-_{st}[\varphi_1])$ holds.

$$
\begin{aligned}
\exists\, v_1 \colon \varphi_1 \wedge \neg\Delta^-_{st}[\varphi_1] &\overset{\text{meta}}{\Longleftrightarrow} (\exists\, v_1 \colon \varphi_1) \wedge (\exists\, v_1 \colon \varphi_1 \wedge \neg\Delta^-_{st}[\varphi_1]) \\
&\overset{\text{meta}}{\Longrightarrow} (\exists\, v_1 \colon \varphi_1) \wedge (\exists\, v_1 \colon \varphi_1 \;?\; \neg\Delta^-_{st}[\varphi_1] : \Delta^+_{st}[\varphi_1]) \\
&\overset{\text{meta}}{\Longleftrightarrow} (\exists\, v_1 \colon \varphi_1) \wedge (\exists\, v_1 \colon \mathbf{F}_{st}[\varphi_1]) \qquad \text{(by the definition of } \mathbf{F}_{st}[\cdot]) \\
&\overset{\text{meta}}{\Longrightarrow} (\exists\, v_1 \colon \varphi_1) \wedge \left[\neg(\exists\, v_1 \colon \Delta^-_{st}[\varphi_1]) \vee (\exists\, v_1 \colon \mathbf{F}_{st}[\varphi_1])\right] \\
&\overset{\text{meta}}{\Longrightarrow} (\exists\, v_1 \colon \varphi_1) \\
&\qquad ?\ \neg\left[(\exists\, v_1 \colon \Delta^-_{st}[\varphi_1]) \wedge \neg(\exists\, v_1 \colon \mathbf{F}_{st}[\varphi_1])\right] \\
&\qquad :\ (\exists\, v_1 \colon \Delta^+_{st}[\varphi_1]) \wedge \neg(\exists\, v_1 \colon \varphi_1) \\
&\overset{\text{meta}}{\Longleftrightarrow} (\exists\, v_1 \colon \varphi_1) \;?\; \neg\Delta^-_{st}[(\exists\, v_1 \colon \varphi_1)] : \Delta^+_{st}[(\exists\, v_1 \colon \varphi_1)] \\
&\qquad \text{(by the definitions of } \Delta^-_{st}[\cdot] \text{ and } \Delta^+_{st}[\cdot]) \\
&\overset{\text{meta}}{\Longleftrightarrow} \mathbf{F}_{st}[\exists\, v_1 \colon \varphi_1] \qquad \text{(by the definition of } \mathbf{F}_{st}[\cdot])
\end{aligned}
$$

Now, assume that $(\exists\, v_1 \colon \neg\varphi_1 \wedge \Delta^+_{st}[\varphi_1])$ holds. We consider two subcases: $(\exists\, v_1 \colon \varphi_1)$ holds; $\neg(\exists\, v_1 \colon \varphi_1)$ holds. Assume that $(\exists\, v_1 \colon \varphi_1)$ holds.

$$
\begin{aligned}
(\exists\, v_1 \colon \varphi_1) \wedge (\exists\, v_1 \colon \neg\varphi_1 \wedge \Delta^+_{st}[\varphi_1]) &\overset{\text{meta}}{\Longrightarrow} (\exists\, v_1 \colon \varphi_1) \wedge (\exists\, v_1 \colon \varphi_1 \;?\; \neg\Delta^-_{st}[\varphi_1] : \Delta^+_{st}[\varphi_1]) \\
&\overset{\text{meta}}{\Longleftrightarrow} (\exists\, v_1 \colon \varphi_1) \wedge (\exists\, v_1 \colon \mathbf{F}_{st}[\varphi_1]) \\
&\qquad \text{(by the definition of } \mathbf{F}_{st}[\cdot]) \\
&\overset{\text{meta}}{\Longrightarrow} (\exists\, v_1 \colon \varphi_1) \wedge \left[\neg(\exists\, v_1 \colon \Delta^-_{st}[\varphi_1]) \vee (\exists\, v_1 \colon \mathbf{F}_{st}[\varphi_1])\right] \\
&\overset{\text{meta}}{\Longrightarrow} (\exists\, v_1 \colon \varphi_1) \\
&\qquad ?\ \neg\left[(\exists\, v_1 \colon \Delta^-_{st}[\varphi_1]) \wedge \neg(\exists\, v_1 \colon \mathbf{F}_{st}[\varphi_1])\right] \\
&\qquad :\ (\exists\, v_1 \colon \Delta^+_{st}[\varphi_1]) \wedge \neg(\exists\, v_1 \colon \varphi_1) \\
&\overset{\text{meta}}{\Longleftrightarrow} (\exists\, v_1 \colon \varphi_1) \;?\; \neg\Delta^-_{st}[(\exists\, v_1 \colon \varphi_1)] : \Delta^+_{st}[(\exists\, v_1 \colon \varphi_1)] \\
&\qquad \text{(by the definitions of } \Delta^-_{st}[\cdot] \text{ and } \Delta^+_{st}[\cdot]) \\
&\overset{\text{meta}}{\Longleftrightarrow} \mathbf{F}_{st}[\exists\, v_1 \colon \varphi_1] \qquad \text{(by the definition of } \mathbf{F}_{st}[\cdot])
\end{aligned}
$$

Assume that $\neg(\exists\, v_1 \colon \varphi_1)$ holds (the second subcase).

$$\begin{aligned}
\neg(\exists\, v_1 \colon \varphi_1) \wedge (\exists\, v_1 \colon \neg\varphi_1 \wedge \Delta^{+}_{st}[\varphi_1]) &\overset{\text{meta}}{\Longrightarrow} \neg(\exists\, v_1 \colon \varphi_1) \wedge (\exists\, v_1 \colon \Delta^{+}_{st}[\varphi_1]) \\
&\overset{\text{meta}}{\Longleftrightarrow} \neg(\exists\, v_1 \colon \varphi_1) \wedge (\exists\, v_1 \colon \Delta^{+}_{st}[\varphi_1]) \wedge \neg(\exists\, v_1 \colon \varphi_1) \\
&\overset{\text{meta}}{\Longrightarrow} (\exists\, v_1 \colon \varphi_1) \\
&\qquad ? \neg \left[(\exists\, v_1 \colon \Delta^{-}_{st}[\varphi_1]) \wedge \neg(\exists\, v_1 \colon \mathbf{F}_{st}[\varphi_1])\right] \\
&\qquad : (\exists\, v_1 \colon \Delta^{+}_{st}[\varphi_1]) \wedge \neg(\exists\, v_1 \colon \varphi_1) \\
&\overset{\text{meta}}{\Longleftrightarrow} (\exists\, v_1 \colon \varphi_1)\, ?\, \neg\Delta^{-}_{st}[(\exists\, v_1 \colon \varphi_1)] : \Delta^{+}_{st}[(\exists\, v_1 \colon \varphi_1)] \\
&\qquad \text{(by the definitions of } \Delta^{-}_{st}[\cdot] \text{ and } \Delta^{+}_{st}[\cdot]) \\
&\overset{\text{meta}}{\Longleftrightarrow} \mathbf{F}_{st}[\exists\, v_1 \colon \varphi_1] \qquad \text{(by the definition of } \mathbf{F}_{st}[\cdot])
\end{aligned}$$

Forall $\varphi \equiv \forall\, v_1 \colon \varphi_1$. The entries for $\Delta^{+}_{st}[\forall\, v_1 \colon \varphi_1]$ and $\Delta^{-}_{st}[\forall\, v_1 \colon \varphi_1]$ can be derived from those for $\Delta^{+}_{st}[\exists\, v_1 \colon \varphi_1]$, $\Delta^{-}_{st}[\exists\, v_1 \colon \varphi_1]$, $\Delta^{+}_{st}[\neg\varphi_1]$, and $\Delta^{-}_{st}[\neg\varphi_1]$.

$$\begin{aligned}
\Delta^{+}_{st}[\forall\, v_1 \colon \varphi_1] &\overset{\text{meta}}{\Longleftrightarrow} \Delta^{+}_{st}[\neg(\exists\, v_1 \colon \neg\varphi_1)] \\
&\overset{\text{meta}}{\Longleftrightarrow} \Delta^{-}_{st}[\exists\, v_1 \colon \neg\varphi_1] \qquad \text{(by the definition of } \Delta^{+}_{st}[\cdot]) \\
&\overset{\text{meta}}{\Longleftrightarrow} (\exists\, v_1 \colon \Delta^{-}_{st}[\neg\varphi_1]) \wedge \neg(\exists\, v_1 \colon \mathbf{F}_{st}[\neg\varphi_1]) \qquad \text{(by the definition of } \Delta^{-}_{st}[\cdot]) \\
&\overset{\text{meta}}{\Longleftrightarrow} (\exists\, v_1 \colon \Delta^{+}_{st}[\varphi_1]) \wedge \neg(\exists\, v_1 \colon \neg\mathbf{F}_{st}[\varphi_1]) \\
&\qquad \text{(by the definition of } \Delta^{-}_{st}[\cdot] \text{ and inductive hypothesis (iii))} \\
&\overset{\text{meta}}{\Longleftrightarrow} (\exists\, v_1 \colon \Delta^{+}_{st}[\varphi_1]) \wedge (\forall\, v_1 \colon \mathbf{F}_{st}[\varphi_1])
\end{aligned}$$

$$\begin{aligned}
\Delta^{-}_{st}[\forall\, v_1 \colon \varphi_1] &\overset{\text{meta}}{\Longleftrightarrow} \Delta^{-}_{st}[\neg(\exists\, v_1 \colon \neg\varphi_1)] \\
&\overset{\text{meta}}{\Longleftrightarrow} \Delta^{+}_{st}[\exists\, v_1 \colon \neg\varphi_1] \qquad \text{(by the definition of } \Delta^{-}_{st}[\cdot]) \\
&\overset{\text{meta}}{\Longleftrightarrow} (\exists\, v_1 \colon \Delta^{+}_{st}[\neg\varphi_1]) \wedge \neg(\exists\, v_1 \colon \neg\varphi_1) \qquad \text{(by the definition of } \Delta^{+}_{st}[\cdot]) \\
&\overset{\text{meta}}{\Longleftrightarrow} (\exists\, v_1 \colon \Delta^{-}_{st}[\varphi_1]) \wedge (\forall\, v_1 \colon \varphi_1) \qquad \text{(by the definition of } \Delta^{+}_{st}[\cdot])
\end{aligned}$$

□

Theorem A.2 (Theorem 3.5) Let S be a structure in 2-STRUCT, and let S'_{proto} be the proto-structure for statement st obtained from S. Let S' be the structure obtained by using S'_{proto} as the first approximation to S' and then filling in instrumentation relations in a topological ordering of the dependences among them: for each arity-k relation $p \in \mathcal{I}$, $\iota^{S'}(p)$ is obtained by evaluating $[\![\psi_p(v_1, \ldots, v_k)]\!]_2^{S'}([v_1 \mapsto u'_1, \ldots, v_k \mapsto u'_k])$ for all tuples $(u'_1, \ldots, u'_k) \in (U^{S'})^k$. Then for every formula $\varphi(v_1, \ldots, v_k)$ and complete assignment Z for $\varphi(v_1, \ldots, v_k)$,

$$[\![\mathbf{F}_{st}[\varphi(v_1, \ldots, v_k)]]\!]_2^S(Z) = [\![\varphi(v_1, \ldots, v_k)]\!]_2^{S'}(Z)$$

Proof The proof is by induction on the size of φ. Let Z be $[v_1 \mapsto u_1, \ldots, v_k \mapsto u_k]$. By Lemma 3.4(iii) and the induction hypothesis, we need only consider the cases for atomic formulas.

1. For $\varphi \equiv \boldsymbol{l}$, where $\boldsymbol{l} \in \{\mathbf{0}, \mathbf{1}\}$,

$$\begin{aligned}
[\![\mathbf{F}_{st}[\boldsymbol{l}]]\!]_2^S(Z) &= [\![\boldsymbol{l} \ ? \ \neg\Delta_{st}^-[\boldsymbol{l}] : \Delta_{st}^+[\boldsymbol{l}]]\!]_2^S(Z) \\
&= [\![\boldsymbol{l} \ ? \ \neg\mathbf{0} : \mathbf{0}]\!]_2^S(Z) \\
&= [\![\boldsymbol{l}]\!]_2^S(Z) \\
&= l \\
&= [\![\boldsymbol{l}]\!]_2^{S'}(Z)
\end{aligned}$$

2. For $\varphi \equiv (v_{i_1} = v_{i_2})$,

$$\begin{aligned}
[\![\mathbf{F}_{st}[v_{i_1} = v_{i_2}]]\!]_2^S(Z) &= [\![v_{i_1} = v_{i_2} \ ? \ \neg\Delta_{st}^-[v_{i_1} = v_{i_2}] : \Delta_{st}^+[v_{i_1} = v_{i_2}]]\!]_2^S(Z) \\
&= [\![v_{i_1} = v_{i_2} \ ? \ \neg\mathbf{0} : \mathbf{0}]\!]_2^S(Z) \\
&= [\![v_{i_1} = v_{i_2}]\!]_2^S(Z) \\
&= Z(v_{i_1}) = Z(v_{i_2}) \\
&= [\![v_{i_1} = v_{i_2}]\!]_2^{S'}(Z)
\end{aligned}$$

3. For $\varphi \equiv p(v_{i_1}, \ldots, v_{i_k})$, $p \in \mathcal{C}$, and $\tau_{p,st}$ is of the form $p \,?\, \neg\delta^-_{p,st} : \delta^+_{p,st}$

$$
\begin{aligned}
[\![\mathbf{F}_{st}[p(v_{i_1}, \ldots, v_{i_k})]]\!]^S_2(Z) &= [\![p(v_{i_1}, \ldots, v_{i_k}) \,?\, \neg\Delta^-_{st}[p(v_{i_1}, \ldots, v_{i_k})] : \Delta^+_{st}[p(v_{i_1}, \ldots, v_{i_k})]]\!]^S_2(Z) \\
&= \left[\!\!\left[\begin{array}{l} p(v_{i_1}, \ldots, v_{i_k}) \\ \quad ?\ \neg(\delta^-_{p,st} \wedge p)\{v_{i_1}, \ldots, v_{i_k}\} \\ \quad :\ (\delta^+_{p,st} \wedge \neg p)\{v_{i_1}, \ldots, v_{i_k}\} \end{array} \right]\!\!\right]^S_2 (Z) \\
&= [\![(p \,?\, \neg\delta^-_{p,st} : \delta^+_{p,st})\{v_{i_1}, \ldots, v_{i_k}\}]\!]^S_2(Z) \\
&= [\![\tau_{p,st}(v_{i_1}, \ldots, v_{i_k})]\!]^S_2(Z) \\
&= [\![p(v_{i_1}, \ldots, v_{i_k})]\!]^{S'}_2(Z)
\end{aligned}
$$

4. For $\varphi \equiv p(w_1, \ldots, w_k)$, $p \in \mathcal{C}$, and $\tau_{p,st}$ is of the form $p \vee \delta_{p,st}$ or $\delta_{p,st} \vee p$

$$
\begin{aligned}
[\![\mathbf{F}_{st}[p(v_{i_1}, \ldots, v_{i_k})]]\!]^S_2(Z) &= [\![p(v_{i_1}, \ldots, v_{i_k}) \,?\, \neg\Delta^-_{st}[p(v_{i_1}, \ldots, v_{i_k})] : \Delta^+_{st}[p(v_{i_1}, \ldots, v_{i_k})]]\!]^S_2(Z) \\
&= [\![p(v_{i_1}, \ldots, v_{i_k}) \,?\, \neg 0 : (\delta_{p,st} \wedge \neg p)\{v_{i_1}, \ldots, v_{i_k}\}]\!]^S_2(Z) \\
&= [\![(p \vee \neg p \wedge \delta_{p,st})\{v_{i_1}, \ldots, v_{i_k}\}]\!]^S_2(Z) \\
&= [\![(p \vee \delta_{p,st})\{v_{i_1}, \ldots, v_{i_k}\}]\!]^S_2(Z) \\
&= [\![\tau_{p,st}(v_{i_1}, \ldots, v_{i_k})]\!]^S_2(Z) \\
&= [\![p(v_{i_1}, \ldots, v_{i_k})]\!]^{S'}_2(Z)
\end{aligned}
$$

5. For $\varphi \equiv p(w_1, \ldots, w_k)$, $p \in \mathcal{C}$, and $\tau_{p,st}$ is of the form $p \wedge \delta_{p,st}$ or $\delta_{p,st} \wedge p$

$$
\begin{aligned}
[\![\mathbf{F}_{st}[p(v_{i_1}, \ldots, v_{i_k})]]\!]^S_2(Z) &= [\![p(v_{i_1}, \ldots, v_{i_k}) \,?\, \neg\Delta^-_{st}[p(v_{i_1}, \ldots, v_{i_k})] : \Delta^+_{st}[p(v_{i_1}, \ldots, v_{i_k})]]\!]^S_2(Z) \\
&= [\![p(v_{i_1}, \ldots, v_{i_k}) \,?\, \neg(\neg\delta_{p,st} \wedge p)\{v_{i_1}, \ldots, v_{i_k}\} : 0]\!]^S_2(Z) \\
&= [\![(p \wedge \neg(\neg\delta_{p,st} \wedge p))\{v_{i_1}, \ldots, v_{i_k}\}]\!]^S_2(Z) \\
&= [\![(p \wedge (\delta_{p,st} \vee \neg p))\{v_{i_1}, \ldots, v_{i_k}\}]\!]^S_2(Z) \\
&= [\![(p \wedge \delta_{p,st})\{v_{i_1}, \ldots, v_{i_k}\}]\!]^S_2(Z) \\
&= [\![\tau_{p,st}(v_{i_1}, \ldots, v_{i_k})]\!]^S_2(Z) \\
&= [\![p(v_{i_1}, \ldots, v_{i_k})]\!]^{S'}_2(Z)
\end{aligned}
$$

6. For $\varphi \equiv p(v_{i_1}, \ldots, v_{i_k})$, $p \in \mathcal{C}$, but $\tau_{p,st}$ is not of the above forms

$$
\begin{aligned}
[\![\mathbf{F}_{st}[p(v_{i_1}, \ldots, v_{i_k})]]\!]_2^S(Z) &= [\![p(v_{i_1}, \ldots, v_{i_k}) \,?\, \neg\Delta_{st}^-[p(v_{i_1}, \ldots, v_{i_k})] : \Delta_{st}^+[p(v_{i_1}, \ldots, v_{i_k})]]\!]_2^S(Z) \\
&= \left[\!\!\left[\begin{array}{l} p(v_{i_1}, \ldots, v_{i_k}) \\ \quad ?\ \neg(p \wedge \neg\tau_{p,st})\{v_{i_1}, \ldots, v_{i_k}\} \\ \quad :\ (\tau_{p,st} \wedge \neg p)\{v_{i_1}, \ldots, v_{i_k}\} \end{array} \right]\!\!\right]_2^S (Z) \\
&= [\![((p \wedge \neg p) \vee (p \wedge \tau_{p,st}) \vee (\tau_{p,st} \wedge \neg p))\{v_{i_1}, \ldots, v_{i_k}\}]\!]_2^S(Z) \\
&= [\![\tau_{p,st}(v_{i_1}, \ldots, v_{i_k})]\!]_2^S(Z) \\
&= [\![p(v_{i_1}, \ldots, v_{i_k})]\!]_2^{S'}(Z)
\end{aligned}
$$

7. For $\varphi \equiv p(v_{i_1}, \ldots, v_{i_k})$, $p \in \mathcal{I}$,

$$
\begin{aligned}
[\![\mathbf{F}_{st}[p(v_{i_1}, \ldots, v_{i_k})]]\!]_2^S(Z) &= [\![p(v_{i_1}, \ldots, v_{i_k}) \,?\, \neg\Delta_{st}^-[p(v_{i_1}, \ldots, v_{i_k})] : \Delta_{st}^+[p(v_{i_1}, \ldots, v_{i_k})]]\!]_2^S(Z) \\
&= \left[\!\!\left[\begin{array}{l} p(v_{i_1}, \ldots, v_{i_k}) \\ \quad ?\neg\Delta_{st}^-[\psi_p]\{v_{i_1}, \ldots, v_{i_k}\} : \Delta_{st}^+[\psi_p]\{v_{i_1}, \ldots, v_{i_k}\} \end{array} \right]\!\!\right]_2^S (Z) \\
&= \left[\!\!\left[\begin{array}{l} \psi_p\{v_{i_1}, \ldots, v_{i_k}\} \\ \quad ?\neg\Delta_{st}^-[\psi_p]\{v_{i_1}, \ldots, v_{i_k}\} : \Delta_{st}^+[\psi_p]\{v_{i_1}, \ldots, v_{i_k}\} \end{array} \right]\!\!\right]_2^S (Z) \\
&= [\![\mathbf{F}_{st}[\psi_p]\{v_{i_1}, \ldots, v_{i_k}\}]\!]_2^S(Z) \\
&= [\![\psi_p\{v_{i_1}, \ldots, v_{i_k}\}]\!]_2^{S'}(Z) \\
&= [\![p(v_{i_1}, \ldots, v_{i_k})]\!]_2^{S'}(Z)
\end{aligned}
$$

□

Appendix B: Proofs of Propositions from Chapter 7

Lemma 7.4.4 *Let A be a 3-valued assignment such that (i) $\langle\!\langle\varphi\rangle\!\rangle(A) = 1$, and (ii) for all $A' \sqsupset A$, $\langle\!\langle\varphi\rangle\!\rangle(A') = 1/2$. Then the formula*

$$\pi \stackrel{\text{def}}{=} \bigwedge_{\substack{1 \leq i \leq n \text{ and} \\ A(x_i) \sqsubset 1/2}} \text{One}(\langle\!\langle\varphi\rangle\!\rangle, A, i)$$

is a prime implicant of $\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor$.

Proof: By Defn. 7.3.1 (Eqn. (7.3)), and the presence of the guard "$A(x_i) \sqsubset 1/2$" in the index of the conjunction, π is a conjunction of literals of the form shown in (7.20), namely the conjunction

$$\pi \equiv \bigwedge_{A(x_i) \sqsubset 1/2} x_i^{A(x_i)}. \tag{B.1}$$

By assumption (i), $\langle\!\langle\varphi\rangle\!\rangle(A) = 1$. Defn. 7.2.1 implies that $\langle\!\langle\varphi\rangle\!\rangle$ is a monotonic function in $\{0, 1, 1/2\}^n \rightarrow \{0, 1, 1/2\}$. Thus, for all a rep. by A, $\langle\!\langle\varphi\rangle\!\rangle(a) \sqsubseteq \langle\!\langle\varphi\rangle\!\rangle(A) = 1$, which, by the definition of $\lfloor\cdot\rfloor$, implies that $\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor(a) = 1$. Because $a(x_i) = A(x_i)$ for all x_i for which $A(x_i) \sqsubset 1/2$, π is an implicant of $\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor$.

To see that π is a prime implicant of $\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor$, consider the right-hand side of Eqn. (B.1) to be the set of literals

$$S_\pi = \{x_i^{A(x_i)} \mid A(x_i) \sqsubset 1/2\}.$$

For any S that is a strict subset of S_π, we would have the set of literals

$$S = \{x_i^{A'(x_i)} \mid A'(x_i) \sqsubset 1/2\}$$

corresponding to an assignment A', where $A' \sqsupset A$. However, by assumption (ii), $\langle\!\langle\varphi\rangle\!\rangle(A') = 1/2$, which means, by Defn. 7.2.1, that

$$\{\langle\!\langle\varphi\rangle\!\rangle(a) \mid a \text{ rep. by } A'\} = \{0, 1\}.$$

Therefore, there is a 2-valued assignment a_0 such that $a_0(x_i) = A'(x_i)$ for all x_i for which $A'(x_i) \sqsubset 1/2$, and $\langle\!\langle\varphi\rangle\!\rangle(a_0) = 0$, and hence $\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor(a_0) = 0$. This means that S, or more precisely,

$$\bigwedge_{A'(x_i) \sqsubset 1/2} x_i^{A'(x_i)}$$

is not an implicant of $\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor$.

Consequently, π is a prime implicant of $\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor$. $\square$

Lemma 7.4.5 *Let π by a prime implicant of $\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor$. Then there is a 3-valued assignment A such that*

(i) $\langle\!\langle\varphi\rangle\!\rangle(A) = 1$

(ii) For all $A' \sqsupset A$, $\langle\!\langle\varphi\rangle\!\rangle(A') = 1/2$

(iii) $\pi \equiv \bigwedge_{\substack{1 \leq i \leq n \text{ and} \\ A(x_i) \sqsubset 1/2}} \text{One}(\langle\!\langle\varphi\rangle\!\rangle, A, i)$

Proof: Let π be the product

$$\pi \stackrel{\text{def}}{=} \bigwedge_{i \in S \subseteq \{1,\ldots,n\}} x_i^{b_i},$$

and let A_S be the assignment

$$A_S = \left[x_i \mapsto \begin{cases} b_i & \text{if } i \in S \\ 1/2 & \text{otherwise} \end{cases} \right]$$

(i) Because π is a prime implicant of $\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor$, for all a represented by A_S, $\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor(a) = 1$. Therefore,

$$\begin{aligned} \langle\!\langle\varphi\rangle\!\rangle(a) &= 1 \qquad \text{(by the definition of } \lfloor\cdot\rfloor) \\ [\![\varphi]\!](a) &= 1 \qquad \text{(by Defn. 7.2.1)} \\ \langle\!\langle\varphi\rangle\!\rangle(A_S) &= 1 \qquad \text{(by Defn. 7.2.1)} \end{aligned}$$

(ii) If $A_S = [x_i \mapsto 1/2 \mid 1 \leq i \leq n]$ (and hence π is the formula $\mathbf{1}$), then property (ii) holds vacuously. Thus, we may assume that A_S binds at least one x_i to a definite value.

Let A' be an assignment such that $A' \sqsupset A_S$. A' and A_S agree on some (possibly empty) set $\{x_i \mid i \in S'\}$, for some $S' \subset S$. Let S'_π be $\{x_i^{b_i} \mid A'(x_i) = b_i, i \in S'\}$. By our assumptions, S'_π is not an implicant of $\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor$. Thus, there is a 2-valued assignment a_0 such

that $a_0(x_i) = A'(x_i)$ for all $i \in S'$, and $\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor(a_0) = 0$; consequently, by the definition of $\lfloor\cdot\rfloor$, $\langle\!\langle\varphi\rangle\!\rangle(a_0) \sqsupseteq 0$.

Pick any a_1 represented by A_S. Because π is a prime implicant of $\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor$, we have

$$\begin{aligned}
\lfloor\langle\!\langle\varphi\rangle\!\rangle\rfloor(a_1) &= 1 \qquad \text{(by Defn. 7.4.2)} \\
\langle\!\langle\varphi\rangle\!\rangle(a_1) &= 1 \qquad \text{(by the definition of } \lfloor\cdot\rfloor) \\
[\![\varphi]\!](a_1) &= 1 \qquad \text{(by Defn. 7.2.1)}
\end{aligned}$$

Because $A' \sqsupset A_S$, a_1 is also represented by A'. Thus,

$$\begin{aligned}
\langle\!\langle\varphi\rangle\!\rangle(A') &= \bigsqcup_{a \text{ rep. by } A'} [\![\varphi]\!](a) \\
&\sqsupseteq [\![\varphi]\!](a_0) \sqcup [\![\varphi]\!](a_1) \\
&\sqsupseteq 0 \sqcup 1 \\
&= 1/2
\end{aligned}$$

(iii) π can be rewritten as

$$\bigwedge_{\substack{1 \le i \le n \text{ and} \\ A_S(x_i) \sqsubset 1/2}} x_i^{A_S(x_i)}. \tag{B.2}$$

Because we showed in (i) that $\langle\!\langle\varphi\rangle\!\rangle(A_S) = 1$, formula (B.2) can, in turn, be expressed as

$$\bigwedge_{\substack{1 \le i \le n \text{ and} \\ A_S(x_i) \sqsubset 1/2}} \text{One}(\langle\!\langle\varphi\rangle\!\rangle, A_S, i).$$

□

Lemma B.0.1 If f is a total Boolean function, then $[\![\text{Primes}[f]]\!] = \langle\!\langle\text{Primes}[f]\rangle\!\rangle$.

Proof: Let A be a 3-valued assignment. By Defn. 7.2.1 and the monotonicity of $[\![\cdot]\!]$ (Lemma 7.1.4), if $[\![\text{Primes}[f]]\!](A)$ yields a definite value d, then $\langle\!\langle\text{Primes}[f]\rangle\!\rangle(A)$ must also yield d.

Thus, we must only consider the case in which

$$[\![\text{Primes}[f]]\!](A) = 1/2. \tag{B.3}$$

To show that $\langle\!\langle \text{Primes}[f] \rangle\!\rangle(A) = 1/2$, we need to show that there exist definite assignments (i) a_1 rep. by A, such that $[\![\text{Primes}[f]]\!](a_1) = 1$, and (ii) a_0 rep. by A, such that $[\![\text{Primes}[f]]\!](a_0) = 0$.

(i) Pick any disjunct of Primes$[f]$ that evaluates to $1/2$ under assignment A, say $\pi_S \stackrel{\text{def}}{=} \bigwedge_{i \in S \subseteq \{1,\ldots,n\}} x_i^{b_i}$. Consider the variables x_j such that $j \in S$ and $A(x_j) = 1/2$. Create a_1 from A by replacing each binding $x_j \mapsto 1/2$ with $x_j \mapsto b_j$. Because $[\![\pi_S]\!](a_1) = 1$, we have $[\![\text{Primes}[f]]\!](a_1) = 1$.

(ii) Suppose for the sake of argument that,

$$\text{for all } a \text{ rep. by } A, [\![\text{Primes}[f]]\!](a) = 1. \tag{B.4}$$

By Defn. 7.4.2, this implies that for all a represented by A, $f(a) = 1$, which means that the formula

$$\pi_A \stackrel{\text{def}}{=} \bigwedge_{A(x_i) \sqsubset 1/2} x_i^{A(x_i)}$$

is an implicant of f. Consequently, Primes$[f]$ contains a disjunct π such that π (considered as a set of literals) is a subset of π_A (considered as a set of literals). Therefore, $[\![\pi]\!](A) = 1$, which means that $[\![\text{Primes}[f]]\!](A) = 1$. However, this contradicts assumption (B.3), and hence our subsequent assumption, assumption (B.4), must be incorrect.

Because f is a total Boolean function, there cannot be any definite assignment a such that $[\![\text{Primes}[f]]\!](a) = 1/2$. Thus, the fact that assumption (B.4) is incorrect implies that there must exist an a_0 rep. by A such that $[\![\text{Primes}[f]]\!](a_0) = 0$.

□

Lemma B.0.2 If f is a total Boolean function, then for all definite assignments a, $[\![\text{Primes}[f]]\!](a) = [\![\neg\text{Primes}[\neg f]]\!](a)$.

Proof:

$$\begin{aligned}
[\![\text{Primes}[f]]\!](a) &= f(a) && \text{(follows from Defn. 7.4.2)}\\
&= 1 - (1 - f(a))\\
&= 1 - \overline{f(a)}\\
&= 1 - \overline{f}(a)\\
&= 1 - [\![\text{Primes}[\neg f]]\!](a) && \text{(follows from Defn. 7.4.2)}\\
&= [\![\neg\text{Primes}[\neg f]]\!](a)
\end{aligned}$$

□

Theorem 7.4.7 *If f is a total Boolean function, then* $[\![\text{Primes}[f]]\!] = [\![\neg\text{Primes}[\neg f]]\!]$.

Proof: Let A be a 3-valued assignment.

$$\begin{aligned}
[\![\text{Primes}[f]]\!](A) &= \langle\!\langle \text{Primes}[f] \rangle\!\rangle(A) && \text{(by Lemma B.0.1)}\\
&= \bigsqcup_{a \text{ rep. by } A} [\![\text{Primes}[f]]\!](a) && \text{(by Defn. 7.2.1)}\\
&= \bigsqcup_{a \text{ rep. by } A} [\![\neg\text{Primes}[\neg f]]\!](a) && \text{(by Lemma B.0.2)}\\
&= \bigsqcup_{a \text{ rep. by } A} (1 - [\![\text{Primes}[\neg f]]\!](a))\\
&= 1 - \bigsqcup_{a \text{ rep. by } A} [\![\text{Primes}[\neg f]]\!](a)\\
&= 1 - \langle\!\langle \text{Primes}[\neg f] \rangle\!\rangle(A) && \text{(by Defn. 7.2.1)}\\
&= 1 - [\![\text{Primes}[\neg f]]\!](A) && \text{(by Lemma B.0.1)}\\
&= [\![\neg\text{Primes}[\neg f]]\!](A)
\end{aligned}$$

□