Symbolic Abstraction: Algorithms and Applications

by

Aditya V. Thakur

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2014

Date of final oral examination: 08/08/2014

The dissertation is approved by the following members of the Final Oral Committee:

Thomas W. Reps, Professor, Computer Sciences
Somesh Jha, Professor, Computer Sciences
Kenneth L. McMillan, Senior Researcher, Microsoft Research
C. David Page Jr., Professor, Biostatistics & Medical Informatics, and Computer Sciences
Mooly Sagiv, Professor, School of Computer Science, Tel Aviv University
Xiaojin Zhu, Associate Professor, Computer Sciences

आदरणीय आई-बाबांना समर्पित

Acknowledgments

I would like to thank Prof. Thomas Reps. Without his guidance and encouragement this thesis would not have materialized. As an adviser, he taught me about computer science, about research, about teaching, about writing, about the 'which' that should have been a 'that', about the power of the haka dance, and about pronouncing names of French computer scientists. As a student, I hope I was not a complete disappointment.

Prof. Mooly Sagiv hosted me during two wonderful and useful visits to Tel Aviv. The first trip was a few months after I had embarked on generalizing Stålmarck's method; the discussions in Tel Aviv helped solidify a lot of ideas presented in this thesis. During the second trip, Mooly's three wonderful daughters taught me how to swing upside-down on a monkey bar (and I also worked on the research that led to Chapter 12).

Prof. Somesh Jha was always willing to impart useful advice, provided I reviewed a paper for him in return—I think I owe him at least four reviews.

The rest my thesis committee, Dr. Kenneth McMillan, Prof. David Page, and Prof. Xiaojin Zhu, asked a lot of interesting questions during my defense and gave many useful comments, which greatly improved the presentation of the dissertation.

I would have given up on pursuing a Ph.D. if it were not for the encouragement and confidence of Prof. R. Govindarajan, my adviser during my Master's at IISc, Bangalore.

Evan Driscoll was always a careful reader of my drafts and a constructive critic of my presentations. He was a sounding board for my ideas, and patient with my many questions on the vagaries of languages, compilers, and linkers. It was a wonderful experience writing the OpenNWA tool with Evan.

I has always been a learning experience working with Akash Lal during the time we overlapped at UW, and during later collaborations when he moved to Microsoft Research.

I did not get an opportunity to work with Nick Kidd while he was at UW, though I did interact with him during my work on the OpenNWA tool, which was built on top of the WAli tool written by Akash and Nick. I got an opportunity to work with Nick this last year, partly because he was assigned to be my Google Mentor. The distributed SAT solver (Chapter 10) would not have been built without his help. I learned a lot from him in the short time we collaborated.

It was a wonderful experience working with Jason Breck on SMASLTOV. I wish him the very best for his graduate career.

Junghee Lim developed the TSL infrastructure, which was used in much of the research on machine-code verification done in this dissertation.

The folks at GrammaTech, Inc., especially Brian Alliet, Denis Gopan, Alexey Loginov, and Suan Hsi Yong, provided tool support and help for the work I carried out on machine-code analysis.

Madison was a more fun place because of the dinners, dances, and drives with Gopika Nair.

The group that organized Diwali Night, especially Prachi, Muffi, Divya, and Rika, brought out talents in me that I never realized I had.

The Southwest chicken sandwich never tasted the same at The Library (Bar & Cafe) after Andres Jaan Tack left for Estonia.

Tristan Ravitch and I shared a passion for `pl-eating` and Mickies Dairy Bar. One day we shall write a SAT solver in Haskell.

Dean Sanders provided me with multiple drama-free years while sharing an apartment. We played the longest chess game of my life, and our discussions on astronomy and biology were always insightful.

Tycho Andersen kept me sane in Madison, and was also an awesome companion on many travels around the world. With him, I discovered *spewn*, ridin' dirty, and DUM DUM DUM.

Cindy Rubio González read many a drafts, commented on presentations, and was always there for me.

My parents have always provided me with unconditional love and support. As a son, I hope I am not a complete disappointment. To them, I dedicate this thesis. My parents would often remark how little they understand about my research. The dedication is written in Marathi so as to provide at least one page that is incomprehensible to my thesis committee and completely understandable to my parents.

My dissertation research was supported, in part, by AFOSR under grant FA9550-09-1-0279; NSF under grants CCF-0540955, CCF-0810053, and CCF-0904371; ONR under grants N00014-10-M-0251 and N00014-11-C-0447; ARL under grant W911NF-09-1-0413; DARPA under cooperative agreement HR0011-12-2-0012; and a Google Ph.D. Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this dissertation are those of the author, and do not necessarily reflect the views of the sponsoring agencies.

Contents

List of Tables

List of Figures

Abstract

This dissertation explores the use of *abstraction* in two areas of automated reasoning: verification of programs, and decision procedures for logics. Establishing that a program is correct is undecidable in general. Program-verification tools sidestep this tar-pit of undecidability by working on an abstraction of a program, which over-approximates the original program's behavior. The theory underlying this approach is called *abstract interpretation*. Developing a scalable and precise abstract interpreter is a challenging problem, especially when analyzing machine code. Abstraction provides a new language for the description of decision procedures, leading to new insights. I call such an abstraction-centric view of decision procedures *Satisfiability Modulo Abstraction (SMA)*.

The unifying theme behind the dissertation is the concept of *symbolic abstraction*:

> Given a formula φ in logic $\mathcal{L}$ and an abstract domain $\mathcal{A}$, the symbolic abstraction of φ is the strongest consequence of φ expressible in $\mathcal{A}$.

This dissertation advances the field of abstract interpretation by presenting two new algorithms for performing symbolic abstraction, which can be used to synthesize various operations required by an abstract interpreter. The dissertation presents two new algorithms for computing inductive invariants for programs. The dissertation shows how the use of symbolic abstraction enables the design of a new abstract domain capable of representing bit-vector inequality invariants.

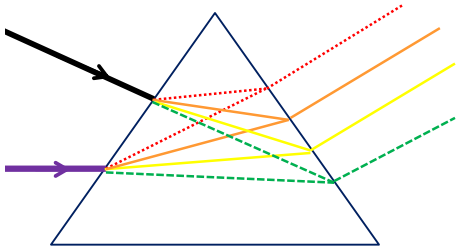
The dissertation advances the field of machine-code analysis by showing how symbolic abstraction can be used to implement machine-code analyses. Furthermore, the dissertation describes MCVETO, a new model-checking algorithm for machine code.

The dissertation advances the field of decision procedures by presenting a variety of SMA solvers. One is based on a generalization of Stålmarck's method, a decision procedure for propositional logic. When viewed through the lens of abstraction, Stålmarck's method can be lifted from propositional logic to richer logics, such as linear rational arithmetic. Furthermore, the SMA view shows that the generalized Stålmarck's method actually performs symbolic abstraction. Thus, the concept of symbolic abstraction brings forth the connection between abstract interpretation and decision procedures. The dissertation describes a new distributed decision procedure for propositional logic, called `DiSSolve`. Finally, the dissertation presents an SMA solver for a new fragment of separation logic.

Chapter 1

Through the Lens of Abstraction

> Cubism came about because, in the process of analyzing form, something that lay in the form, a plane, could be lifted out to float on its own.
>
> — Joseph Plaskett



This thesis explores the use of *abstraction* in two areas of automated reasoning: verification of programs, and decision procedures for logics.

Establishing that a program is correct is undecidable in general. Program-verification tools sidestep this tar-pit of undecidability by working on an abstraction of a program, which over-approximates the original program's behavior. The theory underlying this approach is called *abstract interpretation* (Cousot and Cousot, 1977), and is around forty years old. However, developing a scalable and precise abstract interpreter still remains a challenging problem.

This thesis also explores the use of abstraction in the design and implementation of decision procedures. Abstraction provides a new language for the description of decision procedures,

leading to new insights and new ways of thinking. I call such an abstraction-centric view of decision procedures *Satisfiability Modulo Abstraction*.

The common use of abstraction in this work also brings out a non-trivial and useful relationship between program verification and decision procedures.

The Need for Program Verification

Quis custodiet ipsos custodes?
Who will guard the guards themselves?
— Satires of Juvenal

Software is everywhere. There is an increasing use of computers in our daily lives. More importantly, safety-critical and security-critical computer systems are becoming ubiquitous. Computers are an integral part of communication systems, flight systems, financial systems, power systems, and medical systems. Furthermore, the functionality and complexity of these computer systems has also increased. With this pervasive use of computers it is becoming increasingly important to develop formal methods that certify that a given program is correct. There is an economic argument—faulty software costs the U.S. economy at least $5 billion per year (Charette, 2005)—as well as a moral argument—errors in safety-critical systems could lead to loss of life (Bowen, 2000; Abramson and Pike, 2011)—for formal verification of computer programs.

Ideally, one would like to establish that a computer program P strictly adheres to its functional, safety, and security specification—that is, P satisfies a given property $\mathcal{I}$ for all possible inputs. In theory, determining whether P satisfies $\mathcal{I}$ is undecidable. In practice, the program-verification problem is often addressed by focusing on a restricted set of properties; however, the complexity and size of the software makes program verification difficult even when the class of properties to be verified is restricted. It is the goal of formal methods to make the practice of program verification feasible for industrial-scale systems. Instead of verifying full functional correctness, most program-verification tools verify comparatively "shallow", yet critical, safety properties of

programs, such as absence of null-pointer dereferences, absence of out-of-bounds array accesses, absence of division by zero, and verification of data-structure invariants.

In the last two decades, there has been tremendous innovation in formal verification, and a greater adoption of formal methods, resulting in various industrial-scale tools being developed (Holzmann, 2004; Ball et al., 2004; Delmas and Souyris, 2007; Jetley et al., 2008; Bessey et al., 2010; Chapman and Schanda, 2014). However, implementing a correct, precise, and scalable program-verification tool is an onerous and formidable task. Furthermore, as the analyses and techniques become more complex, the need for automating the construction of such tools increases (Cuoq et al., 2012).

The Emergence of a Disruptive Technology

> (It is shown that) any recognition problem solved by a polynomial time-bounded nondeterministic Turing machine can be "reduced" to the problem of determining whether a given propositional formula is a tautology.
>
> — Stephen A. Cook (1971)

A decision procedure for a logic $\mathcal{L}$ is an algorithm that given a formula $\varphi \in \mathcal{L}$ determines whether (i) φ is *satisfiable* if there exists an assignment to the variables of φ that satisfies φ, or (ii) φ is *unsatisfiable* if no such satisfying assignment exists. A decision procedure for propositional logic is called a *SAT solver*. For example, given the propositional-logic formula $\psi_1 := p_1 \wedge p_2 \wedge (p_3 \vee p_4)$, a SAT solver would determine that ψ_1 is satisfiable with the following (Boolean) assignment to the variables: $[p_1 \mapsto \textbf{true}, p_2 \mapsto \textbf{true}, p_3 \mapsto \textbf{false}, p_4 \mapsto \textbf{true}]$. Although propositional satisfiability is an NP-complete problem, the last fifteen years have born witness to a wide range of SAT solvers that are efficient for deciding satisfiability of formulas with millions of variables arising in practical applications. Furthermore, decision procedures for richer logics, called *Satisfiability Modulo Theories (SMT) solvers*, have been implemented. This tremendous progress in SAT and SMT solvers has resulted in new, practical solutions to a wide variety of problems, including planning problems (Kautz and Selman, 1992; Rintanen, 2009), cryptanalysis (Massacci and Marraro, 2000), bounded model checking (Biere, 2009), program synthesis (Alur et al., 2013),

and automated test-input generation (Godefroid et al., 2005). The application of SMT solvers to automated test-input generation leads us to the topic of the next section.

From Paths in Program to Formulas in Logic

Testing shows the presence, not the absence of bugs.

— Edsger W. Dijkstra (1969)

Suppose that we are asked to determine whether the ERROR statement is reachable in the following C-code snippet:

```
if (x < y) {
  if (2*x+3*z < w)  {

  }
  else {
      ERROR:
  }
}
```

It should be clear that the ERROR statement is reachable if and only if the following formula in quantifier-free bit-vector logic (QFBV) is satisfiable: $(x < y) \wedge \neg(2x + 3z < w)$. Thus, we already see a (rather weak) connection between program verification and decision procedures—namely, the conversion of a particular path in a program to a formula in a logic.

Unfortunately, most programs have loops, procedure calls, and recursion. Consequently, the number of paths that could reach a particular statement is large (or even infinite). Hence, it is not possible to check feasibility of each such path using a decision procedure. This simple observation motivates the use of abstraction in program verification.

1.1 Abstraction in Program Verification

Abstract interpretation (Cousot and Cousot, 1977) provides a way to obtain information about the possible states that a program reaches during execution, but without actually running the program on specific inputs. Instead, it explores the program's behavior for *all* possible inputs, thereby accounting for all possible states that the program can reach.

Before describing the basic concepts of abstract interpretation, I first describe *concrete interpretation* of programs. A *concrete state* of a program characterizes the state of the program at a particular point during program execution, and typically consists of the valuations of the various program variables. For instance, if x and y are integer program variables, a concrete state σ would be described as $[x \mapsto 3, y \mapsto 42]$. (The similarity between the notation used for describing concrete states of programs and models of formulas is not coincidental.) The concrete interpretation of a program gives a *concrete transformer* that describes, for each program statement, how a single (input) concrete state is transformed into a single (output) concrete state according to the semantics of the programming language[1]. For instance, given the statement $S \equiv$ x := x + 5, which increments the value of variable x by 5, the concrete state σ would be transformed into the output concrete state $\sigma' = [x' \mapsto 8, y' \mapsto 42]$; the primes on the variables are merely used to denote that these variables are in the output state.

In contrast to the situation in concrete interpretation, the *abstract states* during abstract interpretation are finite-sized descriptors that represent a *collection of concrete states*. For instance, an abstract state $a = [x \mapsto [2, 10], y \mapsto [20, 200]]$ denotes the set of concrete states in which variable x takes values from the interval from 2 to 10 (inclusive), and variable y takes values from the interval from 20 to 200 (inclusive). The collection of abstract states or *abstract values* form a lattice called an *abstract domain*. The collection of concrete states described by an abstract value a is denoted by $\gamma(a)$. For example, the abstract value a is a value in the abstract domain of intervals *Interval*. Various such abstract domains can be defined, each differing in what aspects of the collection of concrete states they capture. For example, one can use abstract states that represent only the *sign* of a variable's value: `neg`, `zero`, `pos`, or `unknown`.

Paralleling the scenario in the concrete world, in abstract interpretation an *abstract transformer* describes how, for each program statement, an input abstract state gets transformed into an output abstract state. The abstract transformer should be *sound* with respect to the corresponding concrete transformer; that is, the abstract transformer should be an over-approximation of the concrete transformer. For example, given the input abstract state a and the statement S, the

[1]For simplicity of exposition, the discussion here is restricted to deterministic semantics.

output abstract state is $a' = [x' \mapsto [7, 15], y' \mapsto [20, 200]]$. Because each abstract state represents a collection of concrete states, operationally, one can think of abstract interpretation as running the program "in the aggregate." This seemingly simple concept of abstract interpretation has led to much beautiful theory that helps reason about the soundness of sophisticated static analyses of programs. In particular, as long as the abstract transformer is an over-approximation of the concrete semantics of the program, the program properties inferred by the abstract interpreter describe a superset of the states that can actually occur, and can be used as invariants.

Abstract Domains as Logic Fragments

An abstract domain $\mathcal{A}$ can be seen as a logic fragment, and each abstract value can be expressed as a formula in this logic fragment. Let $\mathcal{L}_{\mathcal{A}}$ be some fragment of a general-purpose logic $\mathcal{L}$. We say that $\widehat{\gamma}$ is a *symbolic-concretization operation* if it maps each $A \in \mathcal{A}$ to $\varphi_A \in \mathcal{L}_{\mathcal{A}}$ such that the meaning of φ_A equals the collection of concrete states described by A; that is, $[\![\varphi_A]\!] = \gamma(A)$. $\mathcal{L}_{\mathcal{A}}$ is often defined by a syntactic restriction on the formulas of $\mathcal{L}$.

Example 1.1. For instance, for abstract states over intervals, $\mathcal{L}_{\mathit{Interval}}$ is the set of conjunctions of one-variable inequalities over the program variables. Our experience is that it is generally easy to implement the $\widehat{\gamma}$ operation for an abstract domain. For example, given $A \in$ Interval, it is straightforward to read off the appropriate $\varphi_A \in \mathcal{L}_{\mathit{Interval}}$: each entry $x \mapsto [c_{\mathit{low}}, c_{\mathit{high}}]$ contributes the conjuncts "$c_{\mathit{low}} \leq x$" and "$x \leq c_{\mathit{high}}$." ■

Symbolic Abstraction

Unfortunately, the theory of abstract interpretation does not provide algorithms for computing the abstract transformers from the concrete transformers. Consequently, in practice, an analysis writer needs to manually write the abstract transformers for each concrete operation. This task can be tedious and error-prone, especially for machine code where most instructions involve bit-wise operations. Thus, abstract interpretation has a well-deserved reputation of being a kind of "black art", and consequently difficult to work with.

Many of the key operations needed by an abstract interpreter, including computing abstract transformers, can be reduced to the problem of *symbolic abstraction* (Reps et al., 2004), which connects abstract interpretation and logic.

> Given a formula φ in logic $\mathcal{L}$ and an abstract domain $\mathcal{A}$, find the strongest consequence of φ expressible in $\mathcal{A}$.

We use $\widehat{\alpha}_{\mathcal{A}}(\varphi)$ to denote the symbolic abstraction of $\varphi \in \mathcal{L}$ with respect to an abstract domain $\mathcal{A}$; we drop the subscript $\mathcal{A}$ from $\widehat{\alpha}_{\mathcal{A}}(\cdot)$ when the abstract domain is clear from the context.

Let us see how an algorithm for symbolic abstraction can be used to compute abstract transformers.

Example 1.2. Consider again the statement $S \equiv \mathtt{x := x + 5}$ and the input abstract value $a = [x \mapsto [2, 10], y \mapsto [20, 200]]$ in the abstract domain *Interval*. The semantics of a concrete operation can be stated as a formula in a logic $\mathcal{L}$ that specifies the relation between input and output states. In this example, the semantics of the statement S can be expressed in quantifier-free bit-vector logic (QFBV) as the formula $\varphi_S \equiv x' = x + 5$. The abstract value $a \in$ *Interval* can be expressed as a formula $\widehat{\gamma}(a) \equiv (2 \leq x \leq 10 \wedge 20 \leq y \leq 200)$.

The output abstract state a' can be computed as finding the strongest consequence of the formula $\psi \equiv (\varphi_s \wedge \widehat{\gamma}(a)) = (2 \leq x \leq 10 \wedge 20 \leq y \leq 200 \wedge x' = x + 5)$ that can be expressed as an interval over the variables x' and y'; that is, $\widehat{\alpha}(\psi) = [x' \mapsto [7, 15], y' \mapsto [20, 200]]$. ■

Example 1.3 illustrates that applying the abstract transformer can be complex even for a single machine-code instruction.

Example 1.3. Consider the Intel x86 instruction $\tau \equiv$ `add bh,al`, which adds `al`, the low-order byte of 32-bit register `eax`, to `bh`, the second-to-lowest byte of 32-bit register `ebx`. No other register apart from `ebx` is modified. For simplicity, we only consider the registers `eax`, `ebx`, and `ecx`. The semantics of τ can be expressed in QFBV logic as the formula φ_τ:

$$\varphi_\tau \stackrel{\text{def}}{=} \mathtt{ebx}' = \begin{pmatrix} (\mathtt{ebx} \,\&\, \mathrm{0xFFFF00FF}) \\ |\, ((\mathtt{ebx} + 256 * (\mathtt{eax} \,\&\, \mathrm{0xFF})) \,\&\, \mathrm{0xFF00}) \end{pmatrix} \wedge \mathtt{eax}' = \mathtt{eax} \\ \wedge\, \mathtt{ecx}' = \mathtt{ecx}, \tag{1.1}$$

where "&" and "|" denote bitwise-and and bitwise-or, respectively, and a symbol with a prime denotes the value of the symbol in the post-state. Equation (1.1) shows that the semantics of a seemingly simple `add` instruction involves non-linear bit-masking operations.

Now suppose that the abstract domain $\mathcal{E}_{2^{32}}$ is the domain of affine equalities over the 32-bit registers `eax`, `ebx`, and `ecx`, and that we would like to compute the abstract transformer for τ when the input abstract value $a \in \mathcal{E}_{2^{32}}$ is $\texttt{ebx} = \texttt{ecx}$. This task corresponds to finding the strongest consequence of the formula $\psi \equiv (\texttt{ebx} = \texttt{ecx} \wedge \varphi_\tau)$ that can be expressed as affine relation among $\texttt{eax}'$, $\texttt{ebx}'$, and $\texttt{ecx}'$, which turns out to be $\widehat{\alpha}(\psi) \equiv (2^{16}\texttt{ebx}' = 2^{16}\texttt{ecx}' + 2^{24}\texttt{eax}') \wedge (2^{24}\texttt{ebx}' = 2^{24}\texttt{ecx}')$. Multiplying by a power of 2 serves to shift bits to the left; because it is performed in arithmetic mod 2^{32}, bits shifted off the left end are unconstrained. Thus, the first conjunct of $\widehat{\alpha}(\psi)$ captures the relationship between the low-order two bytes of $\texttt{ebx}'$, the low-order two bytes of $\texttt{ecx}'$, and the low-order byte of $\texttt{eax}'$. ■

My Contributions

In this thesis, I have developed various algorithms for symbolic abstraction, which can be used to synthesize operations needed by an abstract interpreter. Thus, the use of symbolic abstraction lessens the burden on analysis designers by raising the level of automation in abstraction interpretation. Moreover, the use of symbolic abstraction provides help along the following four dimensions:

soundness: Symbolic abstraction provides a way to create analyzers that are correct-by-construction, while requiring an analysis designer to implement only a small number of operations. Consequently, each instantiation of the approach only relies on a small "trusted computing base".

precision: Unlike most conventional approaches to creating analyzers, the use of symbolic abstraction achieves the fundamental limits of precision that abstract-interpretation theory establishes.

scalability: Algorithms for performing symbolic abstraction can be implemented as "anytime" algorithms—i.e., the algorithms can be equipped with a monitor, and if too much time or space is being used, the algorithms can be stopped at any time, and a safe (over-approximating) answer returned. By this means, when the analyzer is applied to a suite of programs that require successively more analysis resources to be used, precision can degrade gracefully.

extensibility: If an additional abstract domain is added to an analyzer to track additional information, information can be exchanged automatically between domains via symbolic abstraction to produce the most-precise abstract values in each domain. This feature is illustrated in greater detail in Section 1.3.1.

Symbolic Abstraction and Quantifier Elimination

I now contrast two approaches to computing abstract transformers: (i) the use of symbolic abstraction, and (ii) the use of quantifier-elimination techniques (Gulwani and Musuvathi, 2008; Monniaux, 2009, 2010).

Gulwani and Musuvathi (2008) defined what they termed the "cover problem", which addresses *approximate existential quantifier elimination*:

> Given a formula φ in logic $\mathcal{L}$, and a set of variables V, find the strongest quantifier-free formula $\overline{\varphi}$ in $\mathcal{L}$ such that $[\![\exists V : \varphi]\!] \subseteq [\![\overline{\varphi}]\!]$.

I use $\text{Cover}_V(\varphi)$ to denote the cover of φ and the set of variables V.

Gulwani and Musuvathi (2008) presented cover algorithms for the theories of uninterpreted functions and linear arithmetic, and showed that covers exist in some theories that do not support quantifier elimination.

The notion of a cover has similarities to the notion of symbolic abstraction, but the latter is more general. If we think of an abstract domain $\mathcal{A}$ as a logic fragment $\mathcal{L}'$, then, in purely logical terms, symbolic abstraction addresses the following problem of performing an *over-approximating translation to an impoverished fragment*:

Given a formula φ in logic $\mathcal{L}$, find the strongest formula ψ in logic-fragment $\mathcal{L}'$ such that $[\![\varphi]\!] \subseteq [\![\psi]\!]$.

Both cover and symbolic abstraction (deliberately) lose information from a given formula φ, and hence both result in over-approximations of $[\![\varphi]\!]$. In general, however, they yield *different* over-approximations of $[\![\varphi]\!]$.

1. The information loss from the cover operation only involves the removal of variable set V from the vocabulary of φ. The resulting formula $\overline{\varphi}$ is still allowed to be an *arbitrarily complex* $\mathcal{L}$ formula; $\overline{\varphi}$ can use all of the (interpreted) operators and (interpreted) relation symbols of $\mathcal{L}$.

2. The information loss from symbolic abstraction involves finding a formula ψ in the fragment $\mathcal{L}'$: ψ must be a *restricted* $\mathcal{L}$ formula; it can only use the operators and relation symbols of $\mathcal{L}'$, and must be written using the syntactic restrictions of $\mathcal{L}'$.

One of the uses of information-loss capability 2 is to bridge the gap between the concrete semantics and an abstract domain. In particular, it may be necessary to use the full power of logic $\mathcal{L}$ to state the concrete semantics of a transformer τ. However, the corresponding abstract transformer *must* be expressed in $\mathcal{L}'$. When $\mathcal{L}'$ is something other than the restriction of $\mathcal{L}$ to a sub-vocabulary, cover is not guaranteed to return an answer in $\mathcal{L}'$, and thus does not yield a suitable *abstract* transformer. This difference is illustrated using the scenario described in Example 1.3.

Example 1.4. In Example 1.3, the abstract transformer for τ is obtained by computing $\widehat{\alpha}(\psi) \in \mathcal{E}_{2^{32}}$, where $\mathcal{E}_{2^{32}}$ is the domain of affine equalities over the 32-bit registers `eax`, `ebx`, and `ecx`; $\psi \equiv (\texttt{ebx} = \texttt{ecx} \wedge \varphi_\tau)$; and φ_τ is defined in Equation (1.1). In particular, $\widehat{\alpha}(\psi) \equiv (2^{16}\texttt{ebx}' = 2^{16}\texttt{ecx}' + 2^{24}\texttt{eax}') \wedge (2^{24}\texttt{ebx}' = 2^{24}\texttt{ecx}')$.

Let R be the set of registers at the input; that is, $R = \{\texttt{eax}, \texttt{ebx}, \texttt{ecx}\}$. The cover of ψ using the set R is

$$\text{Cover}_{R'}(\psi) \equiv \texttt{ebx}' = \begin{pmatrix} (\texttt{ecx}' \ \& \ 0\text{xFFFF00FF}) \\ |\ ((\texttt{ecx}' + 256 * (\texttt{eax}' \ \& \ 0\text{xFF})) \ \& \ 0\text{xFF00}) \end{pmatrix} \tag{1.2}$$

Equation (1.2) shows that even though the cover does not contain any of the input (unprimed) registers, it is not yield an abstract value in the domain $\mathcal{E}_{2^{32}}$. ■

Note that, the notion of symbolic abstraction subsumes the notion of cover: if $\mathcal{L}'$ is the logic $\mathcal{L}$ restricted to the variables not contained in V, then $\widehat{\alpha}_{\mathcal{L}'}(\varphi) = \text{Cover}_V(\varphi), \varphi \in \mathcal{L}$.

1.2 Abstraction in Decision Procedures

In addition to the contributions summarized in Section 1.1, the thesis also describes Satisfiability Modulo Abstraction (SMA), a new approach that uses abstraction to design and implement decision procedures.

This work got started with the result presented in Chapter 6. In that work, I demonstrated the use of abstraction in decision procedures by showing how Stålmarck's method (Sheeran and Stålmarck, 2000), an algorithm for satisfiability checking of propositional formulas, can be explained using abstract-interpretation terminology—in particular, as an instantiation of a more general algorithm, Stålmarck[$\mathcal{A}$], that is parameterized on a (Boolean) abstract domain $\mathcal{A}$ and operations on $\mathcal{A}$. The algorithm that goes by the name "Stålmarck's method" is one instantiation of Stålmarck[$\mathcal{A}$] with a certain abstract domain.

At each step, Stålmarck[$\mathcal{A}$] holds some $A \in \mathcal{A}$; each of the proof rules employed in Stålmarck's method improves A by finding a semantic reduction of A with respect to φ. The advantage of viewing Stålmarck's method using abstract-interpretation terminology is that it brings out a new connection between Stålmarck's method and symbolic abstraction. In essence, to check whether a formula φ is unsatisfiable, Stålmarck[$\mathcal{A}$] computes $\widehat{\alpha}_{\mathcal{A}}(\varphi)$ and performs the test "$\widehat{\alpha}_{\mathcal{A}}(\varphi) = \bot_{\mathcal{A}}$?" If the test succeeds, it establishes that $[\![\varphi]\!] \subseteq \gamma(\bot_{\mathcal{A}}) = \emptyset$, and hence that φ is unsatisfiable.

Using this abstraction-based view, Stålmarck's method can be lifted from propositional logic to *richer logics*, such as quantifier-free linear rational arithmetic (LRA): to obtain a method for richer logics, instantiate the parameterized version of Stålmarck's method with *richer abstract domains*, such as the polyhedral domain (Cousot and Halbwachs, 1978). By this means, we obtained algorithms for computing $\widehat{\alpha}$ for these richer abstract domains. The bottom line is that our algorithm is "dual-use":

- it can be used by an abstract interpreter to compute $\widehat{\alpha}$ (and perform other symbolic abstract operations), and
- it can be used in an SMT solver to determine whether a formula is satisfiable.

One of the main advantages of the SMA approach is that it is able to reuse abstract-interpretation machinery to implement decision procedures. For instance, in in Chapter 6, the polyhedral abstract domain—implemented in PPL (Bagnara et al., 2008)—is used to implement an SMA solver for the logic of linear rational arithmetic. Similarly in Chapter 11, the abstract domain of shapes—implemented in TVLA (Sagiv et al., 2002)—is used in a novel way to implement an SMA solver for a new fragment of separation logic for which existing approaches do not apply.

1.3 The different hats of $\widehat{\alpha}$

In this section, I present some other applications of symbolic abstraction, and draw connections between symbolic abstraction and concepts used in Machine Learning and Artificial Intelligence.

1.3.1 Communication of Information Between Abstract Domains

Apart from computing abstract transformers, symbolic abstraction also provides a way to combine the results from multiple analyses automatically—thereby enabling the construction of new, more-precise analyzers that use multiple abstract domains simultaneously.

Figures 1.1(a) and 1.1(b) show what happens if we want to communicate information between abstract domains *without* symbolic abstraction. Because it is necessary to create explicit conversion

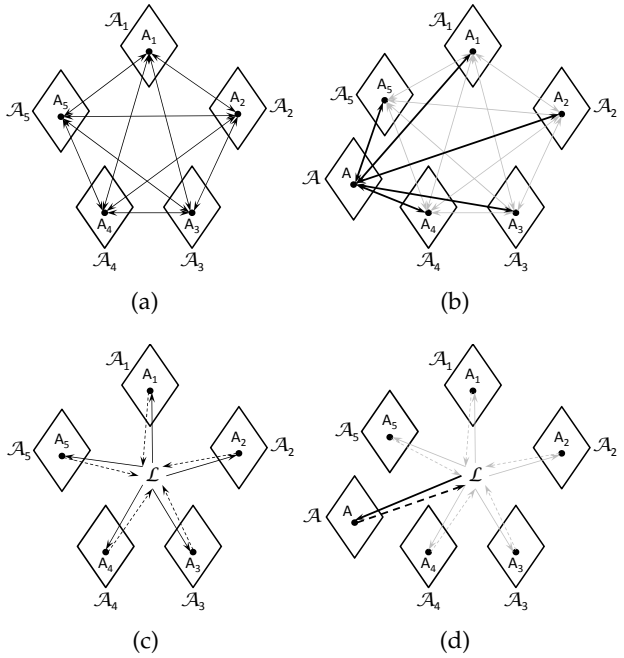


Figure 1.1: Conversion between abstract domains with the clique approach ((a) and (b)) versus the symbolic-abstraction approach ((c) and (d)). In particular, (b) and (d) show what is required under the two approaches when one starts to work with a new abstract domain $\mathcal{A}$.

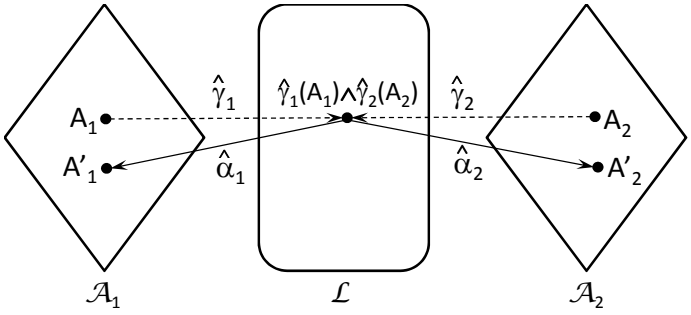


Figure 1.2: Combining information from multiple abstract domains via symbolic abstraction.

routines for each pair of abstract domains, we call this approach the "clique approach". As shown in Figure 1.1(b), when a new abstract domain $\mathcal{A}$ is introduced, the clique approach requires that a conversion method be developed for each prior domain $\mathcal{A}_i$. In contrast, as shown in Figure 1.1(d), the symbolic-abstraction approach only requires that we have $\widehat{\alpha}$ and $\widehat{\gamma}$ methods that relate $\mathcal{A}$ and $\mathcal{L}$.

Moreover, if each analysis i is sound, each result A_i represents an over-approximation of the

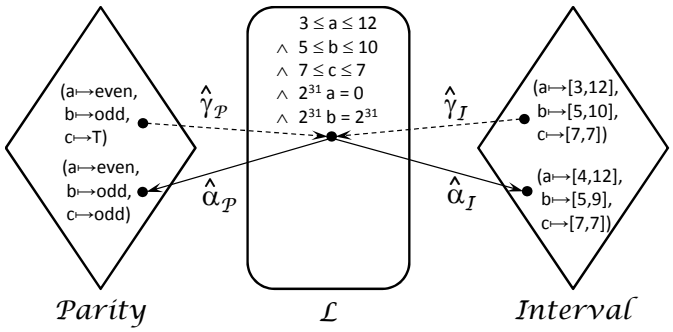


Figure 1.3: An example of how information from the Parity and Interval domains can be used to improve each other via symbolic abstraction.

actual set of concrete states. Consequently, the collection of analysis results $\{A_i\}$ implicitly tells us that only the states in $\bigcap_i \gamma(A_i)$ can actually occur. However, this information is only implicit, and it can be hard to determine what the intersection value really is.

One way to address this issue is to perform a *semantic reduction* (Cousot and Cousot, 1979) of each of the A_i with respect to the set of abstract values $\{A_j \mid i \neq j\}$. Fortunately, symbolic abstraction provides a way to carry out such semantic reductions *without the need to develop pair-wise or clique-wise reduction operators*. The principle is illustrated in Figure 1.2 for the case of two abstract domains, $\mathcal{A}_1$ and $\mathcal{A}_2$. Given $A_1 \in \mathcal{A}_1$ and $A_2 \in \mathcal{A}_2$, we can improve the pair $\langle A_1, A_2 \rangle$ by first creating the formula $\varphi \stackrel{\text{def}}{=} \widehat{\gamma}_1(A_1) \wedge \widehat{\gamma}_2(A_2)$, and then applying $\widehat{\alpha}_1$ and $\widehat{\alpha}_2$ to φ to obtain $A_1' = \widehat{\alpha}_1(\varphi)$ and $A_2' = \widehat{\alpha}_2(\varphi)$, respectively. A_1' and A_2' can be smaller than the original values A_1 and A_2, respectively. We then use the pair $\langle A_1', A_2' \rangle$ instead of $\langle A_1, A_2 \rangle$. Figure 1.3 shows a specific example of how information from the Parity and Interval domains can be used to improve each other via the symbolic-abstraction approach to semantic reduction. Note that in this example

- the Parity value is improved from $(a \mapsto \mathit{even}, b \mapsto \mathit{odd}, c \mapsto \top)$ to $(a \mapsto \mathit{even}, b \mapsto \mathit{odd}, c \mapsto \mathit{odd})$.
- the Interval value is improved from $(a \mapsto [3, 12], b \mapsto [5, 10], c \mapsto [7, 7])$ to $(a \mapsto [4, 12], b \mapsto [5, 9], c \mapsto [7, 7])$.

When there are more than two abstract domains, we form the conjunction $\varphi \stackrel{\text{def}}{=} \bigwedge_i \widehat{\gamma}_i(A_i)$, and then apply each $\widehat{\alpha}_i$ to obtain $A'_i = \widehat{\alpha}_i(\varphi)$.

1.3.2 Symbolic Abstraction and Interpolation in Logic

Symbolic abstraction also leads to a generalization of *interpolation* (Craig, 1957), a concept in mathematical logic. Interpolation has many uses in program verification (Henzinger et al., 2004; McMillan, 2010).

Definition 1.5. Given $\varphi_1, \varphi_2 \in \mathcal{L}$ such that $\varphi_1 \Rightarrow \varphi_2$, $I \in \mathcal{L}$ is said be an **interpolant** of φ_1 and φ_2 if and only if (i) $\varphi_1 \Rightarrow I$, (ii) $I \Rightarrow \varphi_2$, and (iii) I uses only symbols in the shared vocabulary of φ_1 and φ_2. ■

A logic $\mathcal{L}$ supports interpolation if for all $\varphi_1, \varphi_2 \in \mathcal{L}$, there exists an interpolant I. Many logics used in verification support interpolation.

In many applications, interpolant I is used as a heuristic for obtaining a *simple* explanation of why $\varphi_1 \Rightarrow \varphi_2$. The fact that the interpolant is expressed in terms of the common vocabulary is what supports the claim that the result is a simple explanation.

Extending our mantra of connecting concepts in logic to those in abstract interpretation, I introduce the notion of *abstract interpolation*:

Definition 1.6. Given $\varphi_1, \varphi_2 \in \mathcal{L}$ such that $\varphi_1 \Rightarrow \varphi_2$, and an abstract domain $\mathcal{A}$, $\iota \in \mathcal{A}$ is said be an **abstract interpolant** of φ_1 and φ_2 if and only if (i) $\varphi_1 \Rightarrow \widehat{\gamma}(\iota)$, and (ii) $\widehat{\gamma}(\iota) \Rightarrow \varphi_2$. ■

Here there is no common-vocabulary restriction à la Definition 1.5(iii); however, the fact that ι must be an element of abstract domain $\mathcal{A}$ serves as an alternative heuristic for obtaining a simple explanation.

Note that even if the logic $\mathcal{L}$ supports interpolation, it may not necessarily support abstract interpolation for a given abstract domain $\mathcal{A}$. The existence of an abstract interpolant depends on the expressiveness of $\mathcal{A}$. When $\mathcal{A}$ is defined as the logic $\mathcal{L}$ restricted to the symbols common to φ_1 and φ_2, abstract interpolation reduces to standard interpolation. Abstract interpolation

generalizes *uniform interpolation* (Visser, 1996), which is a strengthening of the notion of Craig interpolation. In uniform interpolation, the interpolant has to be expressed in a given vocabulary Σ. Thus, uniform interpolation concerns the problem of keeping the vocabulary Σ, and dropping the rest, which is similar to the cover problem discussed in Section 1.1. Jhala and McMillan (2006) present an algorithm for computing a uniform interpolant for a limited theory consisting of restricted use of array operators and rational constraints.

Symbolic abstraction can be used to compute an abstract interpolant. In particular, to compute an abstract interpolant of φ_1 and φ_2, compute $\iota = \widehat{\alpha}(\varphi_1)$, and verify whether $\widehat{\gamma}(\iota) \Rightarrow \varphi_2$. In fact, this method is guaranteed to compute the strongest abstract interpolant if any abstract interpolant exists. Thus, the method is sound and complete for abstract domains for which $\widehat{\alpha}$ is computable, and logics for which validity of implication is decidable. Though I introduce the notion of abstract interpolation, the rest of the thesis does not further explore this concept.

1.3.3 Symbolic Abstraction in Other Areas of Computer Science

The concept of symbolic abstraction can also be found in the literature on Machine Learning and Artificial Intelligence.

In (Reps et al., 2004), a connection was made between symbolic abstraction and the problem of *concept learning* in (classical) machine learning. In machine-learning terms, an abstract domain $\mathcal{A}$ is a *hypothesis space*; each domain element corresponds to a *concept*. A hypothesis space has an *inductive bias*, which means that it has a limited ability to express sets of concrete objects. In abstract-interpretation terms, inductive bias corresponds to the image of γ on $\mathcal{A}$ not being the full power set of the concrete objects—or, equivalently, the image of $\widehat{\gamma}$ on $\mathcal{A}$ being only a fragment of $\mathcal{L}$. Given a formula φ, the symbolic-abstraction problem is to find the most specific concept that explains the meaning of φ.

There are, however, some differences between the problems of symbolic abstraction and concept learning. These differences mostly stem from the fact that an algorithm for performing symbolic abstraction already starts with a precise statement of the concept in hand, namely, the formula φ. In the machine-learning context, usually no such finite description of the con-

cept exists, which imposes limitations on the types of queries that the learning algorithm can make to an oracle (or teacher); see, for instance, (Angluin, 1987, Section 1.2). The power of the oracle also affects the guarantees that a learning algorithm can provide. In particular, in the machine-learning context, the learned concept is not guaranteed or even required to be an overapproximation of the underlying concrete concept. During the past three decades, the machine-learning theory community has shifted their focus to learning algorithms that only provide probabilistic guarantees. This approach to learning is called *probably approximately correct learning (PAC learning)* (Valiant, 1984; Kearns and Vazirani, 1994). The PAC guarantee also enables a learning algorithm to be applicable to concept lattices that are not complete lattices (Definition 3.1).

These differences between symbolic abstraction and concept learning serve to highlight the novelty of the algorithms and applications discussed in the thesis. Furthermore, they also open up the opportunity for a richer exchange of ideas between the two areas. In particular, one can imagine situations in which it is appropriate for the overapproximation requirement for abstract transformers to be relaxed to a PAC guarantee—for example, if abstract interpretation is being used only to find errors in programs, instead of proving programs correct (Bush et al., 2000), or if we are analyzing programs with a probabilistic concrete semantics (Kozen, 1981; Monniaux, 2000; Cousot and Monerau, 2012).

The problem of symbolic abstraction is closely related to the problems of *restricted consequence finding* (RCF) (McIlraith and Amir, 2001) and *approximate knowledge compilation* in Artificial Intelligence (Selman and Kautz, 1996; Del Val, 1999; Simon and Del Val, 2001). Both of these problems involve translating a formula in a source logic to one in a given (less expressive) target logic. In the context of knowledge compilation, translating to the target logic allows for efficient queries to the knowledge base. Most existing techniques for RCF are applicable to propositional logic or relational first-order logic. Symbolic abstraction also has a connection to another related problem of *forgetting* in Artificial Intelligence (Lin and Reiter, 1994, 1997; Eiter and Wang, 2008; Konev et al., 2009; Wang et al., 2010), which has applications in reuse and combination of ontologies in Semantic Web applications, cognitive robotics, and logic programming.

There has been some work already in connecting some of these problems in Artificial Intelligence to problems in program verification. Jhala and McMillan (2006) apply RCF techniques to compute interpolants to be used for predicate abstraction. I believe it would be fruitful to continue exploring this connection between problems in Artificial Intelligence and symbolic abstraction.

1.4 Thesis Outline

This section provides an outline for each of the chapters in this thesis. Section 1.5 lists the dependencies the various chapters, and suggests several possible orders in which the chapters of the thesis can be read. A **Chapter Notes** section can be found at the end of most chapters. Those sections provide technical and non-technical background on the research described in the respective chapter. Chapters 5–12 contain the results developed during my dissertation research, and present my contributions to advancing the field. Chapter 13 concludes by summarizing the results in the three research threads discussed in this thesis: abstract interpretation, machine-code verification, and decision procedures.

Chapter 2: There's Plenty of Room at the Bottom

In this chapter, I introduce the reader to the unique opportunities and challenges involved in analyzing machine code. Consequently, this chapter puts into context the tools and techniques for machine-code analysis and verification developed in this thesis. I also give an overview of the design space of tools and techniques for machine-code analysis.

Chapter 3: Preliminaries

This chapter introduces and defines concepts in (classical) abstract interpretation and symbolic abstract interpretation, introduces terminology related to decision procedures, and describes Stålmarck's method.

Chapter 4: Symbolic Abstraction from Below

In this chapter, I review two prior algorithms for performing symbolic abstraction:

- The RSY algorithm: a framework for computing $\widehat{\alpha}$ that applies to any logic and abstract domain that satisfies certain conditions (Reps et al., 2004).
- The KS algorithm: an algorithm for computing $\widehat{\alpha}$ that applies to QFBV logic and the domain $\mathcal{E}_{2^w}$ of affine equalities (Elder et al., 2011).

I also present the results of an experiment I carried out to compare the performance of the two algorithms.

Chapter 5: A Bilateral Algorithm for Symbolic Abstraction

This chapter uses the insights gained from the algorithms presented in Chapter 4 to design a new *Bilateral framework* for symbolic abstraction that combines the best features of the KS and RSY algorithms, but also has benefits that none of these previous algorithms have. The Bilateral framework maintains sound under- and over-approximations of the answer, and hence the procedure can return the over-approximation if it is stopped at any point (unlike the RSY and KS algorithms). I compare the performance of the KS algorithm and an instantiation of the Bilateral framework.

Chapter 6: A Generalization of Stålmarck's Method

In this chapter, I give a new account of Stålmarck's method by explaining each of the key components in terms of concepts from the field of abstract interpretation. In particular, I show that Stålmarck's method can be viewed as a general framework, which I call, Stålmarck[$\mathcal{A}$], that is parameterized by an abstract domain $\mathcal{A}$ and operations on $\mathcal{A}$. This abstraction-based view allows Stålmarck's method to be lifted from propositional logic to *richer logics*, such as LRA. Furthermore, the generalized Stålmarck's method falls into the *Satisfiability Modulo Abstraction*

(SMA) paradigm: an SMA solver is designed and implemented using concepts from abstract interpretation.

I also present a connection between symbolic abstraction and Stålmarck's method for checking satisfiability, which leads to a new algorithm for symbolic abstraction. I present experimental results that illustrate the dual-use nature of this Stålmarck-based framework. One experiment uses it to compute abstract transformers, which are then used to generate invariants; another experiment uses it for checking satisfiability.

Chapter 7: Computing Best Inductive Invariants

In this chapter, I show how symbolic abstraction can be used to compute *best inductive invariants* for an entire program. This chapter provides insight on fundamental limits in abstract interpretation, because the best inductive invariant represents the limit of obtainable precision for a given abstract domain.

Chapter 8: Bit-Vector Inequality Domain

In this chapter, I describe how symbolic abstraction enables us to define a new abstract domain, called the $\mathcal{BVI}$ domain, that addresses the following challenges: (1) identifying affine-inequality invariants while handling overflow in arithmetic operations over bit-vector data-types, and (2) holding onto invariants about values in memory during machine-code analysis.

Chapter 9: Symbolic Abstraction for Machine-Code Verification

This chapter describes a model-checking algorithm for stripped machine-code called MCVETO (**M**achine-**C**ode **VE**rification **TO**ol). I describe how MCVETO adapts *directed proof generation* (DPG) (Gulavani et al., 2006) for model checking stripped machine-code. MCVETO implements a new technique called *trace-based generalization*, which enables MCVETO to handle instruction aliasing and self-modifying code. I describe how MCVETO uses *speculative trace refinement* to use *candidate invariants* to speed up the convergence of DPG and to refine the abstraction of the program.

Chapter 10: A Distributed SAT Solver

In this chapter, I describe a new distributed SAT algorithm, called `DiSSolve`, that combines concepts from Stålmarck's method with those found in modern SAT solvers. I evaluate the performance of `DiSSolve` when deployed on a multi-core machine and on the cloud.

Chapter 11: Satisfiability Modulo Abstraction for Separation Logic with Linked Lists

This chapter describes a procedure for checking the unsatisfiability of formulas in a fragment of separation logic. Separation logic (Reynolds, 2002) is an expressive logic for reasoning about heap-allocated data structures in programs. The unsatisfiability checker described in this chapter is designed using concepts from abstract interpretation, and is thus an SMA solver. I present an experimental evaluation of the procedure, and show that it is able to establish the unsatisfiability of formulas that cannot be handled by previous approaches.

Chapter 12: Property-Directed Symbolic Abstraction

This chapter presents a new framework for computing inductive invariants for a program that are sufficient to prove that a given pre/post-condition holds, which I call the *property-directed inductive-invariant (PDII) framework*. The PDII framework computes an inductive invariant that might not be the best (or most precise), but is sufficient to prove a given program property.

1.5 Suggested Order of Reading

This thesis touches on a variety of topics related to automated reasoning, and the reader is encouraged to read the thesis in its entirety in the order presented to fully appreciate the connections among the topics. The reader, however, is welcome to pick the chapters and topics in which they have a greater interest.

Dependencies between chapters are illustrated in the directed graph in Figure 1.4: each node is annotated with a chapter number, and an edge from node i to node j denotes that Chapter i should be read before Chapter j; solid arrows denote strong dependencies, and dashed arrows

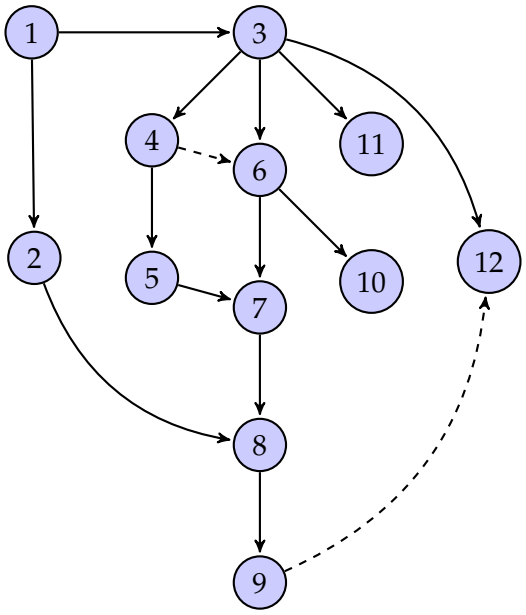


Figure 1.4: Dependencies among chapters

denote weak dependencies. Chapter 13, which is not shown in Figure 1.4, summarizes the results developed during my dissertation research.

The thesis can also read in orders that follow (individually) three different threads: abstract interpretation, decision procedures, and machine-code verification. The chapters relevant to these three topics are listed in Table 1.1. (The results of the thesis in each of these three topics are also summarized in Chapter 13 at a more technical level than is appropriate for the present chapter.)

Topic	Relevant chapters
Abstract Interpretation	Chapters 3, 4, 5, 6, 7, 8, and 12
Decision Procedures	Chapters 3, 6, 10, and 11
Machine-Code Verification	Chapters 2, 8, and 9

Table 1.1: Chapters relevant to a particular topic

1.6 Chapter Notes

The beginning is always the hardest. When I started writing this thesis, I was stuck with the question of what the first chapter should actually "introduce." This thesis covers a wide range of topics in automated reasoning that include theoretical results in abstract interpretation, practical techniques for machine-code verification, decision procedures for propositional logic, and an unsatisfiability checker for separation logic. Introducing all the problems and solution spaces relevant to this thesis in a single chapter appeared daunting and near impossible.

I then came across an article by Karp (2011) that explores the changing relationship between computer science and the natural and social sciences. This new relationship, referred to as the *computational lens*, sees computation as a kind of lens through which to view the world of science. This computational view enables the use of concepts of computer science to give new insights and new ways of thinking about the physical and social sciences. Though the specific research goals described in that article have little to do with this thesis, the title and message of the article resonated with me: the use and benefits of viewing the physical and social sciences though the computational lens paralleled the use and benefits of viewing program verification and decision procedures through the abstraction lens, albeit on a more modest scale. This thought process led to the title of this first chapter.

Chapter 2

There's Plenty of Room at the Bottom

> Now the name of this talk is "There is Plenty of Room at the Bottom"—not just "There is Room at the Bottom."
> — Richard Feynmann, *There is Plenty of Room at the Bottom*

This chapter introduces the reader to the unique opportunities and challenges involved in analyzing machine code. Consequently, this chapter puts into context the tools and techniques for machine-code analysis and verification developed in this thesis, especially McVeto, the **M**achine-**C**ode **VE**rification **TO**ol (Chapter 9).

The purpose and contributions of the chapter can be summarized as follows:

1. I state the need for and advantages of machine-code analysis and verification (Section 2.1).
2. I state the challenges in analyzing machine code, as compared to source-code analysis (Section 2.2).
3. I give an overview of the design space of tools and techniques for machine-code analysis (Section 2.3).

Though applicable to a wide-range of instruction sets, the examples and instantiations used in this thesis use the the 32-bit Intel x86 instruction set (also called IA32). For readers who

need a brief introduction to IA32, it has six 32-bit general-purpose registers (`eax`, `ebx`, `ecx`, `edx`, `esi`, and `edi`), plus two additional registers: `ebp`, the frame pointer, and `esp`, the stack pointer. By convention, register `eax` is used to pass back the return value from a function call. In Intel assembly syntax, the movement of data is from right to left (e.g., `mov eax,ecx` sets the value of `eax` to the value of `ecx`). Arithmetic and logical instructions are primarily two-address instructions (e.g., `add eax,ecx` performs `eax := eax + ecx`). An operand in square brackets denotes a dereference (e.g., if `v` is a local variable stored at offset -12 off the frame pointer, `mov [ebp-12],ecx` performs `v := ecx`). Branching is carried out according to the values of condition codes ("flags") set by an earlier instruction. For instance, to branch to `L1` when `eax` and `ebx` are equal, one performs `cmp eax,ebx`, which sets `ZF` (the zero flag) to 1 iff `eax` $-$ `ebx` $= 0$. At a subsequent jump instruction `jz L1`, control is transferred to `L1` if `ZF` $= 1$; otherwise, control falls through.

2.1 The Need for Machine-Code Analysis

Traditionally, verification efforts have focused on analyzing source code. The need for investigating new techniques for performing machine-code analysis and verification can be summarized as follows:

- In certain situations, source code is not available, and the only available artifact for analysis is machine code.
- Machine code is an artifact that is closer to what actually executes on the machine; models derived from machine code can be more accurate than models derived from source code.
- Analysis and verification techniques that have typically been applied to source code would be unsound if applied to machine code.

Source code is not available when analyzing malware and viruses, or third-party libraries and drivers. Thus, when trying to understand the behavior of malware, or trying to vet third-party binaries for security vulnerabilities, the only option is to analyze machine code.

Computers do not execute source-code programs, they execute machine-code programs. Consequently, there is a mismatch between what the programmer sees in the source code and what is executed. Furthermore, analyses that are performed on source code can fail to detect certain bugs and security vulnerabilities (particularly because compilation, optimization, and link-time transformation can change how the code behaves). Balakrishnan et al. (2007) call this phenomenon WYSINWYX for "What You See Is Not What You eXecute".

When source code is compiled, the compiler and optimizer make certain choices that eliminate some possible behaviors—hence there is sometimes the opportunity to obtain more precise answers from machine-code analysis than from source-code analysis. For instance, security exploits, such as buffer-overflow vulnerabilities, can depend on the layout of variables in memory, which in turn depends on the idiosyncrasies of the compiler and optimizer used to generate the machine code from the source code (Howard, 2002); security exploits could depend on the specific libraries that are statically and dynamically linked to the program; security exploits could depend on the (possibly malicious) instrumentation code inserted in programs subsequent to compilation.

Often it is challenging for the programmer to understand how the compiler transforms the source code. For instance, compilers perform optimizations that exploit undefined behavior in programs; though perfectly sound with respect to the language semantics, such optimizations have unexpected consequences, such as incorrect functionality and deleting security checks (Wang et al., 2013).

The previous paragraph assumes that the compiler itself does not contain bugs. But compilers do have bugs (Yang et al., 2011). In fact, so does the CPU (Cipra, 1995; AMD, 2012). How far down do we go? There is *plenty* of room at the bottom.

2.2 The Challenges in Machine-Code Analysis

Machine-code analysis presents many new challenges. For instance, at the machine-code level, memory is one large byte-addressable array, and an analyzer must handle computed—and

possibly non-aligned—addresses. It is crucial to track array accesses and updates accurately; however, the task is complicated by the fact that arithmetic and dereferencing operations are both pervasive and inextricably intermingled. For instance, consider the load of a local variable `v`, located at offset `-12` in the current activation record, into register `eax`: `mov eax,[ebp-12]`. This instruction involves a *numeric* operation (`ebp-12`) to calculate an address whose value is then *dereferenced* (`[ebp-12]`) to fetch the value of `v`, after which the value is placed in `eax`. Source-code analysis tools often use separate phases of (i) points-to/alias analysis (analysis of addresses) and (ii) analysis of arithmetic operations. Because numeric and address-dereference operations are inextricably intertwined, as discussed above, only very imprecise information would result with the same organization of analysis phases.

Compared with analysis of source code, the challenge is to drop all assumptions about having certain kinds of information available (variables, control-flow graph, call-graph, etc.) In particular, standard approaches to source-code analysis assume that certain information is available—or at least obtainable by separate analysis phases with limited interactions between phases—e.g.,

- a control-flow graph (CFG), or interprocedural CFG (ICFG)
- a call-graph
- a set of variables, split into disjoint sets of local and global variables
- a set of non-overlapping procedures
- type information
- points-to information or alias information

The availability of such information permits the use of techniques that can greatly aid the analysis task. For instance, when one can assume that (i) the program's variables can be split into (a) global variables and (b) local variables that are encapsulated in a conceptually protected environment,

and (ii) a procedure's return address is never corrupted, analyzers often tabulate and reuse explicit summaries that characterize a procedure's behavior.

Source-code-analysis tools sometimes also use questionable techniques, such as interpreting operations in integer arithmetic, rather than bit-vector arithmetic. They also usually make assumptions about the semantics that are not true at the machine-code level—for instance, they usually assume that the area of memory beyond the top-of-stack is not part of the execution state at all (i.e., they adopt the fiction that such memory does not exist).

Machine-code analyzers also need to address new kinds of behaviors that are not seen at the source-code level, such as jumps to "hidden" instructions starting at positions that are out of registration with the instruction boundaries of a given reading of an instruction stream (Linn and Debray, 2003), and self-modifying code.

To summarize, the challenges that analysis of machine code presents include:

absence of information about variables: In stripped executables, no information is provided about the program's global and local variables.

a semantics based on a flat memory model: With machine code, there is no notion of separate "protected" storage areas for the local variables of different procedure invocations, nor any notion of protected fields of an activation record. For instance, a procedure's return address is stored on the stack; an analyzer must prove that it is not corrupted, or discover what new values it could have.

absence of type information: In particular, int-valued and address-valued quantities are indistinguishable at runtime.

arithmetic on addresses is used extensively: Moreover, numeric and address-dereference operations are inextricably intertwined, even during simple operations.

instruction aliasing: Programs written in instruction sets with varying-length instructions, such as x86, can have "hidden" instructions starting at positions that are out of registration with

the instruction boundaries of a given reading of an instruction stream (Linn and Debray, 2003).

self-modifying code: With self-modifying code there is no fixed association between an address and the instruction at that address.

2.3 The Design Space for Machine-Code Analysis

Machine-code-analysis problems come in at least three varieties:

1. In addition to the executable, the program's source code is also available.
2. The source code is unavailable, but the executable includes symbol-table/debugging information ("unstripped executables").
3. The executable has no symbol-table/debugging information ("stripped executables").

The appropriate variant to work with depends on the intended application. Some analysis techniques apply to multiple variants, but other techniques are severely hampered when symbol-table/debugging information is absent. In this thesis, I have primarily been concerned with the analysis of stripped executables, both because it is the most challenging situation, and because it is what is needed in the common situation where one needs to install a device driver or commercial off-the-shelf application delivered as stripped machine code. If an individual or company wishes to vet such programs for bugs, security vulnerabilities, or malicious code (e.g., back doors, time bombs, or logic bombs) analysis tools for stripped executables are required.

In this section, I describe the design space for machine-code analysis by contrasting two previous machine-code-analysis tools—CodeSurfer/x86 (Balakrishnan et al., 2005; Balakrishnan and Reps, 2010) and DDA/x86 (Balakrishnan and Reps, 2008)—with the machine-code-verification tool MCVETO developed in this thesis (Chapter 9). Those three tools represent several firsts:

- CodeSurfer/x86 is the first program-slicing tool for machine code that is able to track the flow of values through memory, and thus help with understanding dependences transmitted via memory loads and stores.

```
void foo() {
  int arr[2], n;
  void (*addr_bar)() = bar;
  if(MakeChoice() == 7) n = 4; // (*)
  else n = 2;
  for(int i = 0; i < n; i++)
    arr[i] = (int)addr_bar; // (**)
  return; // can return to the entry of
bar
 }
```

```
void bar() {
  ERR:
return;
 }

int main() {
  foo();
  return 0;
 }
```

Figure 2.1: A program that, on some executions, can modify the return address of `foo` so that `foo` returns to the beginning of `bar`, thereby reaching `ERR`. (`MakeChoice` is a primitive that returns a random 32-bit number.)

- DDA/x86 is the first automatic program-verification tool that is able to check whether a stripped executable—such as a device driver—conforms to an API-usage rule (specified as a finite-state machine).

- MCVETO is the first automatic program-verification tool capable of verifying (or detecting flaws in) self-modifying code.

Figure 2.1 is an example that will be used to illustrate two points in the design space of machine-code-analysis tools with respect to the question of corruption of a procedure's return address. When the program shown in Figure 2.1 is compiled with Visual Studio 2005, the return address is located two 4-byte words beyond `arr`—in essence, at `arr[3]`. When `MakeChoice` returns 7 at line (*), `n` is set to 4, and thus in the loop `arr[3]` is set to the starting address of procedure `bar`. Consequently, the execution of `foo` can modify `foo`'s return address so that `foo` returns to the beginning of `bar`.

In general, tools that represent different points in the design space have different answers to the question

What properties are checked, and what is expected of the analyzer after the first anomalous action is detected?

First, consider the actions of a typical source-code analyzer, which would propagate abstract states through an interprocedural control-flow graph (ICFG). The call on `foo` in `main` causes it to begin analyzing `foo`. Once it is finished analyzing `foo`, it would follow the "return-edge"

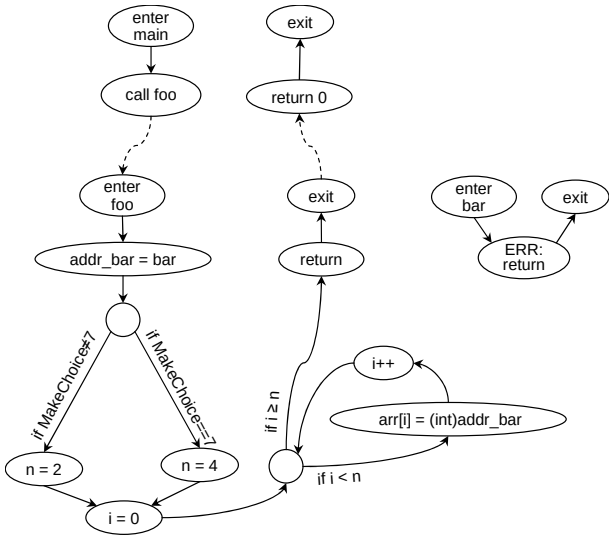


Figure 2.2: Conventional ICFG for the program shown in Figure 2.1. Note that the CFG for `bar` is disconnected from the rest of the ICFG.

in the ICFG back to the point in `main` after the call on `foo`. However, a typical source-code analyzer does not represent the return address explicitly in the abstract state and relies on an unsound assumption that the return address cannot be modified. The analyzer would never analyze the path from `main` to `foo` to `bar`, and would thus miss one of the program's possible behaviors—although the analyzer might report an array-out-of-bounds error at line (**).

The analysis problems in CodeSurfer/x86[1] resemble standard source-code analysis problems, to a considerable degree. CodeSurfer/x86 only follows behaviors expected from a standard compilation model. By a "standard compilation model", I mean that the executable has procedures, activation records (ARs), a global data region, and a free-storage pool; might use virtual functions and DLLs; maintains a runtime stack; each global variable resides at a fixed offset in memory; each local variable of a procedure f resides at a fixed offset in the ARs for f; actual parameters of f are pushed onto the stack by the caller so that the corresponding formal parameters reside at fixed offsets in the ARs for f; the program's instructions occupy a fixed area of memory, and are not self-modifying.

[1]Henceforth, I will not refer to DDA/x86 explicitly. Essentially all of the observations made about CodeSurfer/x86 apply to DDA/x86 as well.

CodeSurfer/x86 is, however, prepared to detect and report deviations from behaviors expected from a standard compilation model, but it is not prepared to explore the consequences of deviant behavior. When such deviations are detected, an error report is issued, and the analysis proceeds by continuing to explore behaviors that stay within those of the desired execution model. For instance, when the analysis discovers that the return address might be modified within the procedure `foo` in Figure 2.1, CodeSurfer/x86 reports the potential violation, but proceeds according to the original return address—i.e., by returning from `foo` to `main`. Similar to source-code analyzers, they would not analyze the path from `main` to `foo` to `bar`. Although it could miss one of the program's possible behaviors, it reports that there is possibly an anomalous overwrite of the return address. In the case of self-modifying code, either a write into the code will be reported or a jump or call to data will be reported.

In contrast, McVeto uses some techniques that permit it not only to detect the presence of "deviant behaviors", but also to explore them as well. The state-space-exploration method used in McVeto discovers that the execution of `foo` can modify `foo`'s return address. It uses the modified return address to discover that `foo` actually returns to the beginning of `bar`, and correctly reports that `ERR` is reachable. The reader is referred to Chapter 9 for more details regarding the internals of McVeto.

2.4 Chapter Notes

I came up with the idea for the title of this chapter when thinking of a title for an invited paper for CAV 2010. I was rather excited about it, until Tom Reps, my adviser, mentioned that Richard Feynman has a (famous) talk with the same title (Feynman, 1960), which I was unaware of at the time. Tom believes I must have seen a reference to Feynman's talk, and subconsciously remembered it.

Chapter 3

Preliminaries

> The advanced reader who skips parts that appear to him too elementary may miss more than the less advanced reader who skips parts that appear to him too complex.
>
> — G. Polya

This chapter introduces and defines concepts in (classical) abstract interpretation (Sections 3.1 and 3.1.2) and symbolic abstract interpretation (Sections 3.2.3 and 3.2.4), introduces terminology related to decision procedures (Section 3.3), and describes Stålmarck's method (Section 3.4).

3.1 Abstract Interpretation

Abstract interpretation (Cousot and Cousot, 1977) is a powerful framework for specifying static program analyses. In this section, I formalize the abstract-interpretation concepts introduced in Section 1.1. I assume familiarity with the basic concepts of abstract interpretation; a detailed description can be found in (Nielson et al., 1999, chapter 4).

Definition 3.1. A partially ordered set $(L, \leq)$ is said to be a *complete lattice* if every subset M of L has both a greatest lower bound (also called *meet*, denoted by $\bigsqcap M$) and a least upper bound (also called *join*, denoted by $\bigsqcup M$) in $(L, \leq)$. A complete lattice has a greatest element, denoted by $\top$, and a least element, denoted by $\bot$, such that $\bot \leq m \leq \top$, for each $m \in L$. ■

Let $(\mathcal{C}, \sqsubseteq_{\mathcal{C}})$ be a complete lattice representing the concrete *collecting semantics* of the program. Usually calculations using $\mathcal{C}$ may be too costly or even uncomputable. This situation motivates using a simpler lattice $(\mathcal{A}, \sqsubseteq_{\mathcal{A}})$. The *abstraction function* $\alpha : \mathcal{C} \to \mathcal{A}$ and the *concretization function* $\gamma : \mathcal{A} \to \mathcal{C}$ are used to express the relationship between $\mathcal{C}$ and $\mathcal{A}$. In particular, α expresses what element of $\mathcal{A}$ is used to represent an element of $\mathcal{C}$, and γ expresses the meaning of an element of $\mathcal{A}$ in terms of an element of $\mathcal{C}$.

Definition 3.2. We define $\mathcal{G} = \mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$ to be a *Galois connection* between complete lattices $(\mathcal{C}, \sqsubseteq_{\mathcal{C}})$ and $(\mathcal{A}, \sqsubseteq_{\mathcal{A}})$ if and only if $\alpha : \mathcal{C} \to \mathcal{A}$ and $\gamma : \mathcal{A} \to \mathcal{C}$ are total functions that satisfy

$$\alpha(c) \sqsubseteq_{\mathcal{A}} a \Leftrightarrow c \sqsubseteq_{\mathcal{C}} \gamma(a) \tag{3.1}$$

■

Intuitively, Equation (3.1) expresses that α and γ "respect" the orderings of the two lattices: If an element $c \in \mathcal{C}$ is described by the element $\alpha(c) \in \mathcal{A}$, then all elements $a \in \mathcal{A}$ that are higher than $\alpha(c)$ in the $\mathcal{A}$-lattice order—$\alpha(c) \sqsubseteq_{\mathcal{A}} a$—should be mapped to elements $\gamma(a) \in \mathcal{C}$ that are higher that c in the $\mathcal{C}$-lattice order—$c \sqsubseteq_{\mathcal{C}} \gamma(a)$.

The next example illustrates the concepts of abstraction and concretization functions using the abstract domain of intervals introduced in Section 1.1.

Example 3.3. Let $\mathcal{C}$ be the the powerset of states over integer variables x and y, and $\mathcal{A}$ could be the lattice of states over intervals.

$$\begin{aligned} c_1 &= \{[x \mapsto 2, y \mapsto 200], [x \mapsto 5, y \mapsto 120], [x \mapsto 10, y \mapsto 20]\} \\ \alpha(c_1) = a_1 &= [x \mapsto [2, 10], y \mapsto [20, 200]] \\ \gamma(a_1) = c_2 &= \{[x \mapsto 2, y \mapsto 20], [x \mapsto 2, y \mapsto 21], [x \mapsto 2, y \mapsto 200], \ldots \\ &\qquad [x \mapsto 10, y \mapsto 20], [x \mapsto 10, y \mapsto 21], [x \mapsto 10, y \mapsto 200], \ldots\} \end{aligned}$$

The abstract value $a_1 = \alpha(c_1)$ is the interval representation of concrete states in $c_1 \in \mathcal{C}$. Notice that there is a loss of precision when moving from $\mathcal{C}$ to $\mathcal{A}$; however, the abstract value a_1 is a

more succinct representation of c_1. Applying the concretization function γ to the abstract value a_1 results in the set of concrete states c_2. Note that $c_1 \sqsubseteq_{\mathcal{C}} c_2$. ■

I will make use of the following properties of a Galois connection $\mathcal{G} = \mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$ (Nielson et al., 1999, section 4.3.1):

- γ uniquely determines α by

$$\alpha(c) = \bigsqcap \{a \mid c \sqsubseteq_{\mathcal{C}} \gamma(a)\} \tag{3.2}$$

- α is completely additive; that is, given $C \subseteq \mathcal{C}$,

$$\alpha\left(\bigsqcup C\right) = \bigsqcup \{\alpha(c) \mid c \in C\} \tag{3.3}$$

Definition 3.4. The *representation function* β maps a singleton concrete state σ such that $\{\sigma\} \in \mathcal{C}$ to the least value in $\mathcal{A}$ that over-approximates $\{\sigma\}$. ■

In other words, β returns the abstraction of a singleton concrete state; i.e.,

$$\beta(\sigma) = \alpha\left(\{\sigma\}\right). \tag{3.4}$$

Experience shows that, for most abstract domains, it is easy to implement a β function. Note that implementing the α function can still be challenging, especially when a formula is used to specify an infinite (or large) state set.

If $\mathcal{C}$ is the powerset of some set of states and $c \in \mathcal{C}$, then using Equations (3.4) and (3.3) we have that

$$\alpha(c) = \bigsqcup_{\sigma \in c} \beta(\sigma) \tag{3.5}$$

3.1.1 Conjunctive Abstract Domains

Many common abstract domains, and all the domains used in this thesis, fall into the category of *conjunctive domains*.

Domain	Φ
Integer Interval domain	inequalities of the form $c_1 \leq v$ and $v \leq c_2$ over integers
Polyhedral domain	linear inequalities over reals or rationals
Bit-Vector Interval domain $\mathcal{I}_{2^w}$	inequalities of the form $c_1 \leq v$ and $v \leq c_2$ integers mod 2^w
Affine-equalities domain $\mathcal{E}_{2^w}$	affine equalities over integers mod 2^w

Table 3.1: Examples of conjunctive domains. v, v_i represent program variables and c, c_i represent constants

Definition 3.5. Let Φ be a given set of formulas expressed in $\mathcal{L}$. A *conjunctive domain* over Φ is an abstract domain $\mathcal{A}$ such that:

- For any $a \in \mathcal{A}$, there exists a finite subset $\Psi \subseteq \Phi$ such that $\gamma(a) = [\![\bigwedge \Psi]\!]$.
- For any finite $\Psi \subseteq \Phi$, there exists an $a \in \mathcal{A}$ such that $\gamma(a) = [\![\bigwedge \Psi]\!]$.
- There is an algorithm that, for all $a_1, a_2 \in \mathcal{A}$, checks whether $a_1 \sqsubseteq a_2$ holds. ■

Table 3.1 lists the conjunctive domains used in this thesis.

The polyhedral domain and its associated abstract operations are defined in (Cousot and Halbwachs, 1978). An element of the polyhedral domain (Cousot and Halbwachs, 1978) is a convex polyhedron, bounded by hyperplanes; that is, each polyhedron can be expressed as some conjunction of linear inequalities ("half-spaces") from the set $\mathcal{F} = \{\sum_{v \in \mathcal{V}} c_v v \geq c \mid c, c_v \text{ are constants}\}$, where $\mathcal{V}$ is the set of program variables.

The affine-equalities domain $\mathcal{E}_{2^w}$ and its associated abstract operations are defined in Elder et al. (2011). An element of the affine-equalities domain $\mathcal{E}_{2^w}$ is a set of affine equalities, where each equality C_i is of the form $\Sigma_j a_{ij} v_j + b_i = 0$, $a_{ij}, b_i \in \mathbb{Z}_{2^w}$, $v_j \in \mathcal{V}$. In this thesis, we are mainly interested in versions of the $\mathcal{E}_{2^w}$ that are based on machine arithmetic, e.g., arithmetic modulo 2^8, 2^{16}, 2^{32}, or 2^{64}, and are able to take care of arithmetic overflow. The domain of affine-equalities *over integers* $\mathcal{E}_{\mathbb{Z}}$ was first studied by Karr (1976). The abstract domain $\mathcal{E}_{\mathbb{Z}}$ does not take into account overflow; consequently, the analysis results of an $\mathcal{E}_{\mathbb{Z}}$ analysis can be unsound. Furthermore, $\mathcal{E}_{2^w}$ is more expressive than $\mathcal{E}_{\mathbb{Z}}$. For instance, it is possible to express the parity of

a variable in $\mathcal{E}_{2^w}$, but not in $\mathcal{E}_{\mathbb{Z}}$. In arithmetic mod 2^w, multiplying by a power of 2 serves to shift bits to the left, with the effect that bits shifted off the left end are unconstrained. For example, in $\mathcal{E}_{2^{32}}$ the affine relation $2^{31}x = 2^{31}$ places a constraint solely on the least-significant bit of x; consequently, $2^{31}x = 2^{31}$ is satisfied by all states in which x is odd. Similarly, $2^{31}x = 0$ is satisfied by all states in which x is even.

3.1.2 Abstract Operations

Abstract Transformer

Suppose that $\mathcal{G} = \mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$ is a Galois connection between concrete domain $\mathcal{C}$ and abstract domain $\mathcal{A}$. Then the "best transformer" (Cousot and Cousot, 1979), or best abstract post operator for transition τ, denoted by $\widehat{\mathrm{Post}}[\tau] : \mathcal{A} \to \mathcal{A}$, is the most-precise abstract operator possible, given $\mathcal{A}$, for the concrete post operator for τ, $\mathrm{Post}[\tau] : \mathcal{C} \to \mathcal{C}$. $\widehat{\mathrm{Post}}[\tau]$ can be expressed in terms of α, γ, and $\mathrm{Post}[\tau]$, as follows (Cousot and Cousot, 1979):

$$\widehat{\mathrm{Post}}[\tau] = \alpha \circ \mathrm{Post}[\tau] \circ \gamma \tag{3.6}$$

Equation (3.6) defines the limit of precision obtainable using abstraction $\mathcal{A}$. However, it is non-constructive; it does not provide an *algorithm*, either for applying $\widehat{\mathrm{Post}}[\tau]$ or for finding a representation of the function $\widehat{\mathrm{Post}}[\tau]$. In particular, in many cases, the application of γ to an abstract value would yield an intermediate result—a set of concrete states—that is either infinite or too large to fit in computer memory.

In practice, analysis designers typically give up on Equation (3.6) and manually write, for each concrete operation, an abstract transformer that satisfies the weaker condition

$$\mathrm{Post}^\sharp[\tau] \sqsupseteq \alpha \circ \mathrm{Post}[\tau] \circ \gamma. \tag{3.7}$$

Furthermore, an analysis designer needs to prove that Equation (3.7) holds for the abstract transformers that he has defined.

One common approach for implementing abstract transformers is via operator-by-operator reinterpretation. The analysis developer writes a sound abstract operator—$\&^\sharp$, $+^\sharp$, $*^\sharp$, $|^\sharp$, and $=^\sharp$—for each concrete operator—$\&$, $+$, $*$, $|$, and $=$, respectively. These abstract operators are used in place of the concrete operators occurring in the term used to express the concrete transformer.

Abstract Transformers via Reinterpretation

TSL (for " **T**ransformer **S**pecification **L**anguage") (Lim and Reps, 2008, 2013), is a language and system that uses the reinterpretation approach to automate the creation of abstract-interpretation systems (primarily for machine-code analysis). With TSL, one specifies an instruction set's concrete operational semantics by writing an interpreter using a first-order functional language. The interpreter specifies how each instruction transforms a concrete state. To define an abstract interpretation for an abstract domain $\mathcal{A}$, one defines "replacement" datatypes for TSL's numeric/bit-vector and map datatypes, along with 42 replacement numeric/bit-vector operations, 12 replacement map-access/update operations, plus a few additional operations, such as $\sqcup$, $\sqcap$, and widen. From such a reinterpretation of the TSL meta-language, which is extended automatically to TSL expressions and functions, TSL creates sound over-approximating transformers.

In general, such an operator-by-operator reinterpretation approach is sound, but is not guaranteed to compute $\widehat{\mathrm{Post}}$. Furthermore, the task of manually defining the abstract operations can be tedious and error-prone, especially for machine code where most instructions involve bit-wise operations. Example 1.3 illustrated that applying $\widehat{\mathrm{Post}}$ can be complex even for a single instruction. For this example, $\widehat{\mathrm{Post}}[\tau](a) \stackrel{\text{def}}{=} (2^{16}\mathtt{ebx}' = 2^{16}\mathtt{ecx}' + 2^{24}\mathtt{eax}') \wedge (2^{24}\mathtt{ebx}' = 2^{24}\mathtt{ecx}')$, while the reinterpretation approach results in $\mathrm{Post}^\sharp[\tau] \stackrel{\text{def}}{=} 2^{24}\mathtt{ebx}' = 2^{24}\mathtt{ecx}'$, which is a strict over-approximation of $\widehat{\mathrm{Post}}[\tau](a)$.

Example 3.6, given below, motivates why a technique for applying $\widehat{\mathrm{Post}}$ for a sequence of instructions is desirable.

Example 3.6. Consider the two Intel x86 instructions $\tau_1 \stackrel{\text{def}}{=} \texttt{push 42}$ and $\tau_1 \stackrel{\text{def}}{=} \texttt{pop eax}$. The instruction τ_1 pushes the value 42 onto the stack, and τ_2 pops the value on top of the stack into the register `eax`.

As in Example 1.3, we consider the abstract domain $\mathcal{E}_{2^{32}}$ of relational affine equalities among 32-bit registers. We would like to compute $\widehat{\text{Post}}[\tau_1;\tau_2](a)$, where $a = \top$; that is, we want to apply the abstract transformer for the sequential composition of τ_1 and τ_2. One approach is to compute the value $\widehat{\text{Post}}[\tau_2](\widehat{\text{Post}}[\tau_1](a))$. $\widehat{\text{Post}}[\tau_1](a) \stackrel{\text{def}}{=} \top$, because $\mathcal{A}$ is able to only capture relations between registers, and is incapable of holding onto properties of values on the stack. Consequently,

$$\widehat{\text{Post}}[\tau_2](\widehat{\text{Post}}[\tau_1](a)) = \widehat{\text{Post}}[\tau_2](\top) = \top.$$

In contrast,

$$\widehat{\text{Post}}[\tau_1;\tau_2](A) \stackrel{\text{def}}{=} \texttt{eax}' = 42.$$

■

Examples 1.3 and 3.6 illustrate the fact that an abstract domain can be expressive enough to capture invariants before and after a sequence of operations or instructions, but not capable of capturing invariants at some intermediate point. As illustrated in Example 1.3, the application of a sequence of sound abstract *operations* can lose precision because it is necessary to express the intermediate result in the limited language supported by the abstract domain. Example 3.6 illustrates that a similar phenomenon holds at the level of a sequence of *instructions*: again, the need to express an intermediate result in the limited language supported by the abstract domain can cause a loss of precision.

Semantic Reduction

The semantic reduction of an abstract value $A \in \mathcal{A}$ is the lowest value, according to the lattice order of $\mathcal{A}$, that has the same concretization as that of A (Cousot and Cousot, 1979).

Definition 3.7. Given an abstract value $A \in \mathcal{A}$, the *semantic reduction* of A, denoted by $\rho(A)$, is defined as:

$$\rho(A) \stackrel{\text{def}}{=} \bigsqcap\{A' \mid \gamma(A') = \gamma(A)\}$$ ■

Working with semantic reductions of abstract values often improves the overall precision of the abstract interpretation. The use of semantic reduction in computing the reduced product (Section 1.3.1) is one such example. Furthermore, when inferring whether a program point is unreachable, an abstract interpreter checks whether the abstract value A at that program point is $\bot$. It is possible that A is not $\bot$, but $\rho(A)$ is $\bot$.

This thesis also uses the following variant of the notion of semantic reduction, which provides a way to refine an abstract value with respect to a formula.

Definition 3.8. Abstract value A' is a *semantic reduction of A with respect to φ* if (i) $\gamma(A') \cap [\![\varphi]\!] = \gamma(A) \cap [\![\varphi]\!]$, and (ii) $A' \sqsubseteq A$. ■

Example 3.9. Let $I \equiv [x \mapsto [4, 100]]$ be an element of the interval domain $\mathcal{I}_{2^{32}}$, and $\varphi \equiv (x\&1 = 1)$. The semantic reduction of I with respect to φ is $I' \equiv [x \mapsto [5, 99]]$, because φ restricts x to be an odd number. ■

The relation between Definitions 3.7 and 3.8 is explained in Section 3.2.3.

Communication Among Abstract Domains

It is often beneficial to perform analysis using multiple abstract domains (Cousot et al., 2006). Section 1.3.1 illustrated the concepts and issues associated with combining the results from multiple analyses automatically.

3.2 Symbolic Abstract Interpretation

The aforementioned problems with applying γ can be side-stepped by working with *symbolic* representations of sets of states (i.e., using formulas in some logic $\mathcal{L}$). The use of $\mathcal{L}$ formulas to

represent sets of states is convenient because logic can also be used for specifying a language's concrete semantics; i.e., the concrete semantics of a transformer $\mathrm{Post}[\tau]$ can be stated as a formula $\varphi_\tau \in \mathcal{L}$ that specifies the relation between input states and output states. However, the symbolic approach introduces a new challenge: how to bridge the gap between $\mathcal{L}$ and $\mathcal{A}$. In particular, we need to develop (i) concepts and algorithms to handle interconversion between formulas of $\mathcal{L}$ and abstract values in $\mathcal{A}$, and (ii) symbolic versions of the operations that form the core repertoire at the heart of an abstract interpreter.

3.2.1 Moving from $\mathcal{A}$ to $\mathcal{L}$

Definition 3.10. Given an abstract value $A \in \mathcal{A}$, the *symbolic concretization* of A, denoted by $\widehat{\gamma}(A)$, maps A to a formula $\widehat{\gamma}(A)$ such that A and $\widehat{\gamma}(A)$ represent the same set of concrete states (i.e., $\gamma(A) = [\![\widehat{\gamma}(A)]\!]$). ■

Experience shows that, for most abstract domains, it is easy to write a $\widehat{\gamma}$ function, as illustrated in Example 1.1.

3.2.2 Moving from $\mathcal{L}$ to $\mathcal{A}$

Definition 3.11. Given $\varphi \in \mathcal{L}$, the symbolic abstraction of φ, denoted by $\widehat{\alpha}(\varphi)$, maps φ to the best value in $\mathcal{A}$ that over-approximates $[\![\varphi]\!]$ (i.e., $\widehat{\alpha}(\varphi) = \alpha([\![\varphi]\!])$). ■

In other words, $\widehat{\alpha}(\varphi)$ is the strongest consequence of φ expressible in $\mathcal{A}$. Examples 1.2 and 1.3 illustrate the concept of $\widehat{\alpha}$.

3.2.3 Symbolic Versions of Abstract Operations

The symbolic operations of $\widehat{\gamma}$ and $\widehat{\alpha}$ can be used to implement the abstract operations of Section 3.1.2.

Abstract Transformer

Given an abstract value $a \in \mathcal{A}$, computing $\widehat{\mathrm{Post}}[\tau](a)$ corresponds to computing $\widehat{\alpha}(\widehat{\gamma}(a) \wedge \varphi_\tau)$, where φ_τ is a formula in logic $\mathcal{L}$ that expresses the semantics of τ. Examples 1.2 and 1.3 illustrate this concept.

Apart from being able to compute $\widehat{\mathrm{Post}}[\tau](a)$, it is sometimes useful to create a representation of $\widehat{\mathrm{Post}}[\tau]$. Some intraprocedural (Graham and Wegman, 1976) and many interprocedural (Sharir and Pnueli, 1981; Knoop and Steffen, 1992) dataflow-analysis algorithms operate on instances of an abstract datatype $\mathcal{T}$ that (i) represents a family of abstract functions (or relations), and (ii) is closed under composition and join. By "creation of a representation of $\widehat{\mathrm{Post}}[\tau]$", we mean finding the best instance in $\mathcal{T}$ that over-approximates $\mathrm{Post}[\tau]$. $\widehat{\alpha}(\varphi_\tau)$ computes the best instance in $\mathcal{T}$ that over-approximates $\mathrm{Post}[\tau]$.

Example 3.12. Consider the Intel x86 instruction $\tau \equiv$ `add bh,al`, which was used in Example 1.3. The formula φ_τ (Equation (1.1)) expresses the semantics of τ. Let $\mathcal{E}_{2^{32}}$ be the abstract domain of affine equalities over the 32-bit registers `eax`, `ebx`, `ecx`, `eax'`, `ebx'`, and `ecx'`. $\widehat{\alpha}(\varphi_\tau)$ computes the representation of $\widehat{\mathrm{Post}}[\tau]$; in particular,

$$\widehat{\alpha}(\varphi_\tau) \equiv (2^{16}\texttt{ebx}' = 2^{16}\texttt{ebx} + 2^{24}\texttt{eax}) \wedge (\texttt{eax}' = \texttt{eax}) \wedge (\texttt{ecx}' = \texttt{ecx}). \qquad \blacksquare$$

Semantic Reduction

In symbolic abstract interpretation, $\widehat{\mathrm{Assume}}$ plays the role of the semantic-reduction operation (Definition 3.8). Given $\varphi \in \mathcal{L}$ and $A \in \mathcal{A}$, $\widehat{\mathrm{Assume}}[\varphi](A)$ returns the best value in $\mathcal{A}$ that over-approximates the meaning of φ in concrete states described by A. That is, $\widehat{\mathrm{Assume}}[\varphi](A)$ equals $\alpha([\![\varphi]\!] \cap \gamma(A))$, and can be computed as $\widehat{\alpha}(\varphi \wedge \widehat{\gamma}(A))$.

Example 3.13. Consider the scenario discussed in Example 3.9. The semantic reduction of I with respect to φ is

$$\begin{aligned} I' &\equiv \widehat{\mathrm{Assume}}[\varphi](I) \\ &\equiv \widehat{\alpha}\big((x\&1 = 1) \wedge 4 \leq x \wedge x \leq 100\big) \\ &\equiv [x \mapsto [5, 99]] \end{aligned}$$

■

The semantic-reduction operator ρ (Definition 3.7) can be computed using $\widehat{\mathrm{Assume}}$: $\rho(A) = \widehat{\mathrm{Assume}}[\widehat{\gamma}(A)](A)$.

Communication Among Abstract Domains

Suppose that there are two Galois connections $\mathcal{G}_1 = \mathcal{C} \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \mathcal{A}_1$ and $\mathcal{G}_2 = \mathcal{C} \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \mathcal{A}_2$, and one wants to work with the reduced product of $\mathcal{A}_1$ and $\mathcal{A}_2$ (Cousot and Cousot, 1979, Section 10.1), which I denote by $\mathcal{A}_1 \star \mathcal{A}_2$. The semantic reduction of a pair $\langle A_1, A_2 \rangle$ can be performed by letting ψ be the formula $\widehat{\gamma}_1(A_1) \wedge \widehat{\gamma}_2(A_2)$, and creating the pair $\langle \widehat{\alpha}_1(\psi), \widehat{\alpha}_2(\psi) \rangle$. Section 1.3.1 illustrates how symbolic abstraction can be used to compute the reduced product.

Symbolic abstraction can also be used to translate an abstract value in one abstract domain to the best abstract value in another abstract domain. This capability would allow an abstract interpreter to use different abstract domains for different portions of a program being analyzed. Given $A_1 \in \mathcal{A}_1$, one can find the most-precise value $A_2 \in \mathcal{A}_2$ that over-approximates A_1 in $\mathcal{A}_2$ as follows: $A_2 = \widehat{\alpha}_2(\widehat{\gamma}_1(A_1))$.

Notation for Over-Approximating Operators

We replace "$\widehat{\ }$" with "$\widetilde{\ }$" to denote over-approximating operators—e.g., $\widetilde{\alpha}(\varphi)$, $\widetilde{\mathrm{Assume}}[\varphi](A)$, and $\widetilde{\mathrm{Post}}[\tau](A)$.

3.2.4 Properties of Symbolic Abstraction

In this section, I give two theorems that characterize $\widehat{\alpha}(\varphi)$ in terms of underapproximating elements (Theorem 3.14) and in terms of overapproximating elements (Theorem 3.15). These theorems will be used later to justify algorithms that, given φ, find $\widehat{\alpha}(\varphi)$ from below (Chapter 4), from above (Chapter 6), or both (Chapter 5).

Theorem 3.14. $\widehat{\alpha}(\varphi) = \bigsqcup\{\beta(S) \mid S \models \varphi\}$

Proof. By definition,

$$\widehat{\alpha}(\varphi) = \alpha([\![\varphi]\!]) \tag{3.8}$$

$$S \models \varphi \text{ if and only if } \{S\} \subseteq [\![\varphi]\!] \tag{3.9}$$

From Equation (3.9), we have that

$$[\![\varphi]\!] = \bigcup\{S \mid S \models \varphi\} \tag{3.10}$$

Using Equation (3.10) in Equation (3.8), we have that

$$\widehat{\alpha}(\varphi) = \alpha\left(\bigcup\{S \mid S \models \varphi\}\right) \tag{3.11}$$

Using Equation (3.5) in Equation (3.11), we have that

$$\widehat{\alpha}(\varphi) = \bigsqcup\{\beta(S) \mid S \models \varphi\} \qquad \square$$

Theorem 3.15. $\widehat{\alpha}(\varphi) = \bigsqcap\{a \mid \varphi \Rightarrow \widehat{\gamma}(a)\}$

Proof. By definition,

$$\widehat{\alpha}(\varphi) = \alpha([\![\varphi]\!]) \tag{3.12}$$

By Equation (3.2), we have that

$$\alpha([\![\varphi]\!]) = \bigsqcap\{a \mid [\![\varphi]\!] \sqsubseteq_{\mathcal{C}} \gamma(a)\} \tag{3.13}$$

Using Equations (3.12) and (3.13), we have that

$$\widehat{\alpha}(\varphi) = \bigsqcap \{a \mid [\![\varphi]\!] \sqsubseteq_{\mathcal{C}} \gamma(a)\} \tag{3.14}$$

From the definition of $\widehat{\gamma}$, we have that

$$\gamma(a) = [\![\widehat{\gamma}(a)]\!] \tag{3.15}$$

Using Equation (3.15) and from the relation between $\sqsubseteq_{\mathcal{C}}$ and $\Rightarrow$, we have that

$$[\![\varphi]\!] \sqsubseteq_{\mathcal{C}} \gamma(a) \text{ if and only if } \varphi \Rightarrow \widehat{\gamma}(a) \tag{3.16}$$

Using Equation (3.16) in Equation (3.14), we have that

$$\widehat{\alpha}(\varphi) = \bigsqcap \{a \mid \varphi \Rightarrow \widehat{\gamma}(a)\} \qquad \square$$

3.3 Decision Procedures

This section introduces basic terminology related to decision procedures.

A formula ψ is *satisfiable* if and only if there exists an interpretation S that makes ψ true. S is called a *model* of ψ, and this fact is denoted by $S \models \psi$. Conversely, a formula is *unsatisfiable* if all interpretations make the formula false. A formula is *valid* if all models make the formula true. ψ is valid if and only if $\neg\psi$ is unsatisfiable. A valid formula is also called a *tautology*.

A *theory* T for a signature of predicate and function symbols is a set of sentences. A formula ψ is *satisfiable modulo theory* T if and only if there exists a interpretation S of the theory T such that $S \models \psi$. In essence, T restricts the possible meanings of the predicate and function symbols. Unless otherwise stated, "valid" means T-valid and "satisfiable" means T-satisfiable.

A *decision procedure* for a logic $\mathcal{L}$ is an algorithm that can determine in finite time whether or not a given formula $\psi \in \mathcal{L}$ is satisfiable. A decision procedure for Boolean or propositional logic is called a *SAT solver*. A decision procedure determining satisfiability modulo a theory is called

a *Satisfiability Modulo Theory (SMT) solver*. Note that if $\mathcal{L}$ is closed under negation, a decision procedure for $\mathcal{L}$-satisfiability can be used for deciding $\mathcal{L}$-validity.

Most modern implementations of SAT and SMT solvers compute a satisfying model, if one exists, when given a formula ψ. They also typically take a timeout parameter that bounds the amount of time the implementation is allowed to spend on determining the answer. Given a formula $\psi \in \mathcal{L}$, we use "$\texttt{Model}(\psi)$" to denote a function that returns (i) a satisfying model S if a decision procedure is able to determine that ψ is satisfiable (typically in a given time limit), (ii) "None" if a decision procedure is able to determine that ψ is unsatisfiable in the given time limit, and (iii) "TimeOut" otherwise. This terminology is used in the algorithms presented in Chapters 4 and 5.

3.4 Stålmarck's Method for Propositional Logic

This section describes Stålmarck's method (Sheeran and Stålmarck, 2000), a fast validity checker (and hence a fast SAT solver) for propositional logic.

We use $\mathbf{0}$ and $\mathbf{1}$ to denote the propositional constants false and true, respectively. Propositional variables, negations of propositional variables, and propositional constants are referred to collectively as *literals*. Stålmarck's method manipulates *formula relations*, which are equivalence relations over literals. A formula relation R will be denoted by $\equiv_R$, although we generally omit the subscript when R is understood. We use $\mathbf{0} \equiv \mathbf{1}$ to denote the universal (and contradictory) equivalence relation $\{l_i \equiv l_j \mid l_i, l_j \in \text{Literals}\}$.

In this section, we review the two types of proof rules used in Stålmarck's method with the help of examples: (i) Example 3.16 describes the simple deductive rules, called *propagation rules*, used in Stålmarck's method, and (ii) Example 3.17 describes the *Dilemma rule*, a branching and merging proof rule employed by Stålmarck's method. The algorithm is described in Section 3.4.2.

$$u_1 \Leftrightarrow (u_2 \vee u_3) \tag{3.17}$$
$$u_2 \Leftrightarrow (a \wedge b) \tag{3.18}$$
$$u_3 \Leftrightarrow (\neg a \vee \neg b) \tag{3.19}$$

Figure 3.1: Integrity constraints corresponding to the formula $\varphi = (a \wedge b) \vee (\neg a \vee \neg b)$. The root variable of φ is u_1.

$$\frac{p \Leftrightarrow (q \vee r) \qquad p \equiv \mathbf{0}}{q \equiv \mathbf{0} \qquad r \equiv \mathbf{0}}\ \textsc{Or1}$$

$$\frac{p \Leftrightarrow (q \wedge r) \qquad q \equiv \mathbf{1} \qquad r \equiv \mathbf{1}}{p \equiv \mathbf{1}}\ \textsc{And1}$$

Figure 3.2: Propagation rules.

3.4.1 Examples

Example 3.16. Consider the tautology $\varphi = (a \wedge b) \vee (\neg a \vee \neg b)$. It expresses the pigeonhole principle for two pigeons ($a \wedge b$) and one hole ($\neg a \vee \neg b$). This example shows that the simpler component of the two components of Stålmarck's method (application of "simple deductive rules") is sufficient to establish that φ is valid.

Stålmarck's method first assigns to every subformula of φ a unique Boolean variable in a set of propositional variables $\mathcal{U}$, and generates a list of *integrity constraints* as shown in Figure 3.1. An *assignment* is a function in $\mathcal{U} \to \{0, 1\}$. The integrity constraints limit the set of assignments in which we are interested. Here the integrity constraints encode the structure of the formula.

Stålmarck's method establishes the validity of the formula φ by showing that $\neg\varphi$ leads to a contradiction (which means that $\neg\varphi$ is unsatisfiable). Thus, the second step of Stålmarck's method is to create a formula relation that contains the assumption $u_1 \equiv \mathbf{0}$. Figure 3.2 lists some *propagation rules* that enable Stålmarck's method to refine a formula relation by inferring new equivalences. For instance, rule Or1 says that if $p \Leftrightarrow (q \vee r)$ is an integrity constraint and $p \equiv \mathbf{0}$ is in the formula relation, then $q \equiv \mathbf{0}$ and $r \equiv \mathbf{0}$ can be added to the formula relation.

Figure 3.3 shows how, starting with the assumption $u_1 \equiv \mathbf{0}$, the propagation rules derive the explicit contradiction $\mathbf{0} \equiv \mathbf{1}$, thus proving that φ is valid. ■

The process of repeatedly using the propagation rules until no new information is deduced is called *0-saturation*, and is described in Algorithm 2 (Figure 3.6). As demonstrated by the following example, 0-saturation is not sufficient to prove all tautologies.

$u_1 \equiv \mathbf{0}$		...	by assumption
$u_2 \equiv \mathbf{0}$,	$u_3 \equiv \mathbf{0}$	...	by rule OR1 using Equation (3.17)
$\neg a \equiv \mathbf{0}$,	$\neg b \equiv \mathbf{0}$	...	by rule OR1 using Equation (3.19)
$a \equiv \mathbf{1}$,	$b \equiv \mathbf{1}$	...	interpretation of logical negation
$u_2 \equiv \mathbf{1}$		...	by rule AND1 using Equation (3.18)
$\mathbf{0} \equiv \mathbf{1}$		...	$u_2 \equiv \mathbf{0}, u_2 \equiv \mathbf{1}$

Figure 3.3: Proof that φ is valid.

Example 3.17. Consider the tautology $\psi = \big(a \wedge (b \vee c)\big) \Leftrightarrow \big((a \wedge b) \vee (a \wedge c)\big)$, which expresses the distributivity of $\wedge$ over $\vee$. The integrity constraints for ψ are:

$$u_1 \;\Leftrightarrow\; (u_2 \Leftrightarrow u_3) \qquad u_2 \;\Leftrightarrow\; (a \wedge u_4) \qquad u_3 \;\Leftrightarrow\; (u_5 \vee u_6)$$

$$u_4 \;\Leftrightarrow\; (b \vee c) \qquad u_5 \;\Leftrightarrow\; (a \wedge b) \qquad u_6 \;\Leftrightarrow\; (a \wedge c)$$

The root variable of ψ is u_1. Assuming $u_1 \equiv \mathbf{0}$ and then performing 0-saturation does not result in a contradiction; all we can infer is $u_2 \equiv \neg u_3$.

To prove that ψ is a tautology, we need to use the *Dilemma Rule*, which is a special type of branching and merging rule. It is shown schematically in Figure 3.4. After two literals u_i and u_j are chosen, the current formula relation R is split into two formula relations, based on whether we assume $u_i \equiv u_j$ or $u_i \equiv \neg u_j$, and transitive closure is performed on each variant of R. Next, the two relations are 0-saturated, which produces the two formula relations R'_1 and R'_2. Finally, the two proof branches are merged by intersecting the set of tuples in R'_1 and R'_2. The correctness of the Dilemma Rule follows from the fact that equivalences derived from *both* of the

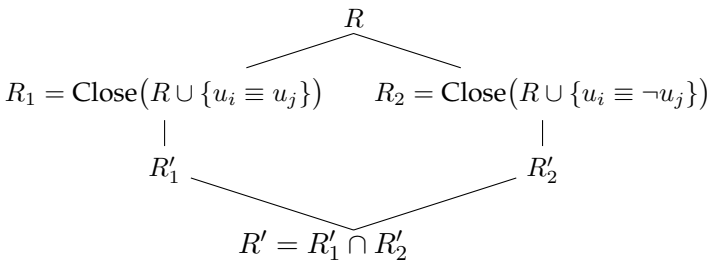


Figure 3.4: The Dilemma Rule.

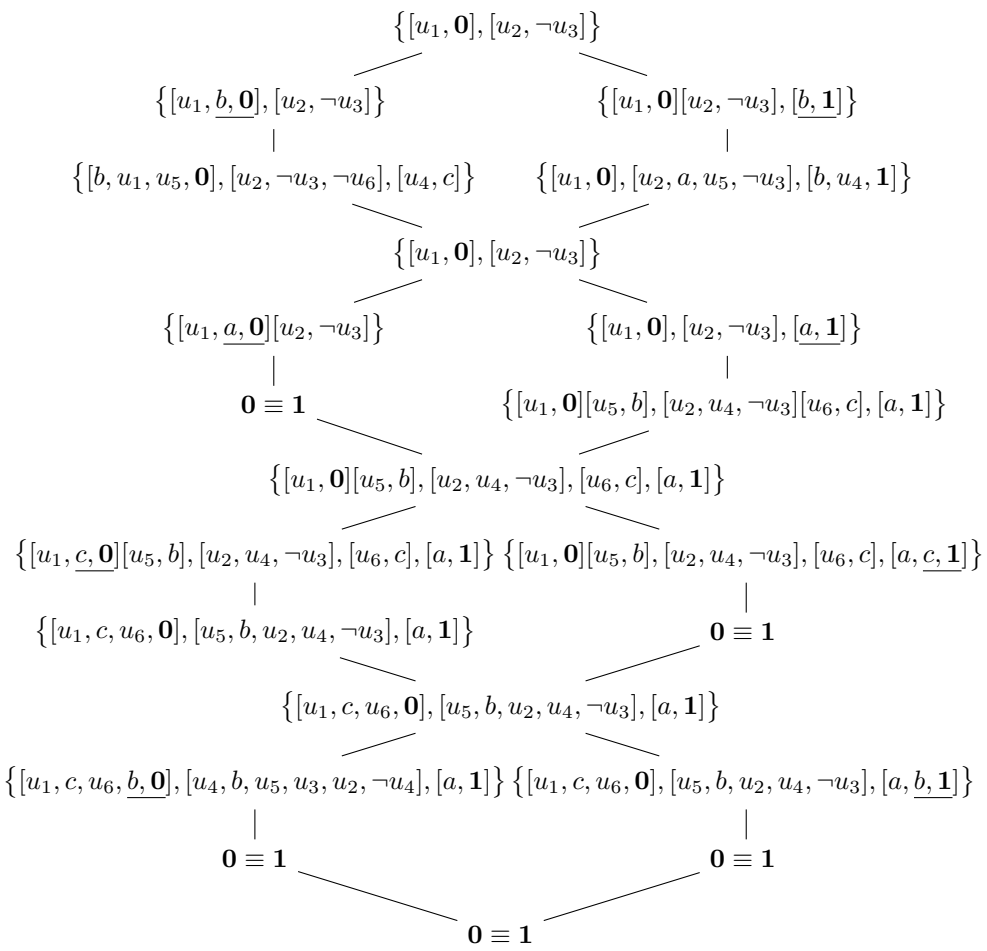


Figure 3.5: Sequence of Dilemma Rules in a proof that ψ is valid. (Details of 0-saturation steps are omitted. For brevity, singleton equivalence-classes are not shown.)

(individual) assumptions $u_i \equiv u_j$ and $u_i \equiv \neg u_j$ hold irrespective of whether $u_i \equiv u_j$ holds or whether $u_i \equiv \neg u_j$ holds.

The formula ψ in Example 3.17 can be proved valid using repeated application of the Dilemma rule, as shown in Figure 3.5. The first application of the Dilemma Rule, which splits on the value of b, does not make any progress; i.e., no new information is obtained after the intersection. The next two applications of the Dilemma Rule, which split on the values of a and c, respectively, each deduce a contradiction on one of their branches. The (universal) relation $\mathbf{0} \equiv \mathbf{1}$ is the identity element for intersection, and hence the intersection result is the equivalence relation from the

Algorithm 1: propagate$(J, R_1, R, \mathcal{I})$

$R_2 = \text{ApplyRule}[\mathcal{I}](J, R_1)$
return *Close*$(R \cup R_2)$

Algorithm 2: 0-saturation$(R, \mathcal{I})$

repeat
  $R' \leftarrow R$
  foreach $J \in I, R_1 \subseteq R$ **do**
    $R \leftarrow \text{propagate}(J, R_1, R, \mathcal{I})$
until $(R = R') \parallel \text{contradiction}(R)$
return R

Algorithm 3: 1-saturation$(R, \mathcal{I})$

repeat
  $R' \leftarrow R$
  foreach v_i, v_j **such that** $v_i \equiv v_j \notin R$ **and** $v_i \equiv \neg v_j \notin R$ **do**
    $R_1 \leftarrow \text{Close}(R \cup \{v_i \equiv v_j\})$
    $R_2 \leftarrow \text{Close}(R \cup \{v_i \equiv \neg v_j\})$
    $R'_1 \leftarrow 0\text{-saturation}(R_1)$
    $R'_2 \leftarrow 0\text{-saturation}(R_2)$
    $R \leftarrow R'_1 \cap R'_2$
until $(R = R') \parallel \text{contradiction}(R)$
return R

Algorithm 4: k-saturation$(R, \mathcal{I})$

repeat
  $R' \leftarrow R$
  foreach v_i, v_j **such that** $v_i \equiv v_j \notin R$ **and** $v_i \equiv \neg v_j \notin R$ **do**
    $R_1 \leftarrow \text{Close}(R \cup \{v_i \equiv v_j\})$
    $R_2 \leftarrow \text{Close}(R \cup \{v_i \equiv \neg v_j\})$
    $R'_1 \leftarrow (\text{k}–1)\text{-saturation}(R_1, \mathcal{I})$
    $R'_2 \leftarrow (\text{k}–1)\text{-saturation}(R_2, \mathcal{I})$
    $R \leftarrow R'_1 \cap R'_2$
until $(R = R') \parallel \text{contradiction}(R)$
return R

Algorithm 5: k-Staalmarck(φ)

$(v_\varphi, \mathcal{I}) \leftarrow \text{integrity}(\varphi)$
$R \leftarrow \{v_\varphi \equiv \mathbf{0}\}$
$R' \leftarrow \text{k-saturation}(R, \mathcal{I})$
if $R' = \mathbf{0} \equiv \mathbf{1}$ **then return** *valid*
else return *unknown*

Figure 3.6: Stålmarck's method. The operation Close performs transitive closure on a formula relation after new tuples are added to the relation.

non-contradictory branch. Finally, splitting on the variable b leads to a contradiction on both branches. ■

Repeatedly applying the Dilemma rule until no new information is deduced is called *1-saturation*, and is described in Algorithm 3 (Figure 3.6). Unfortunately, 1-saturation may not be sufficient to prove certain tautologies. The 1-saturation procedure can be generalized to the k-saturation procedure shown in Algorithm 4 (Figure 3.6).

3.4.2 Pseudo-Code for Stålmarck's Method

Algorithm 1 (Figure 3.6) implements the propagation rules of Figure 3.2. Given an integrity constraint $J \in \mathcal{I}$ and a set of equivalences $R_1 \subseteq R$, line 1 calls the function $\mathrm{ApplyRule}$, which instantiates and applies the derivation rules of Figure 3.2 and returns the deduced equivalences in R_2. The new equivalences in R_2 are incorporated into R and the transitive closure of the resulting equivalence relation is returned. We implicitly assume that if Close derives a contradiction then it returns $\mathbf{0} \equiv \mathbf{1}$.

Algorithm 2 (Figure 3.6) describes *0-saturation*, which calls $\mathrm{propagate}$ repeatedly until no new information is deduced, or a contradiction is derived. If a contradiction is derived, then the given formula is proved to be valid.

The Dilemma Rule is applied repeatedly until no new information is deduced by a process called *1-saturation*, shown in Algorithm 3 (Figure 3.6). 1-saturation uses two literals v_i and v_j, and splits the formula relation with respect to $v_i \equiv v_j$ and $v_i \equiv \neg v_j$ (lines 4 and 5). 1-saturation finds a contradiction when both 0-saturation branches identify contradictions (in which case $R = R'_1 \cap R'_2$ equals $\mathbf{0} \equiv \mathbf{1}$).

The 1-saturation procedure can be generalized to the k-saturation procedure shown in Algorithm 4. Stålmarck's method (Algorithm 5) is structured as a procedure for validity checking. The actions of the algorithm are parameterized by a certain parameter k that is fixed by the user. For a given tautology, if k is large enough Stålmarck's method can prove validity, but if k is too small the answer returned is "unknown". In the latter case, one can increment k and try again. However, for each k, $(k+1)$-saturation is significantly more expensive than k-saturation: the running time of Algorithm 5 as a function of k is $O(|\varphi|^k)$ (Sheeran and Stålmarck, 2000).

3.5 Chapter Notes

This chapter omits discussion of algorithms to compute fixpoints; the reader if referred to sections 2.4 and 4.2 of Nielson et al. (1999). Reps et al. (2007) discuss the use of weighted pushdown systems (WPDSs) to perform inter-procedural analysis of programs. The WALi library (Kidd

et al., 2007) implements such a WPDS-based framework. The various analyses described in this thesis are implemented using the WALi library.

Stålmarck's method was protected by Swedish, European, and U.S. patents (Stålmarck, 1989), which may have discouraged experimentation by researchers. Indeed, one finds relatively few publications that concern Stålmarck's method—some of the exceptions being Harrison (1996); Cook and Gonthier (2005); Björk (2009). Kunz and Pradhan (1994) discuss an algorithm closely related to Stålmarck's method.

More information on SAT solvers can be found in (Harrison, 2009, Chapter 2); (Kroening and Strichman, 2008, Chapter 2); Darwiche and Pipatsrisawat (2009); and Marques-Silva et al. (2009). More information on SMT solvers can be found in (Kroening and Strichman, 2008, Chapters 3–7); and Barrett et al. (2009).

Chapter 4

Symbolic Abstraction from Below

> Started from the bottom, now we are here,
> Started from the bottom, now the whole team here.
> — Drake

In this chapter, I review two prior algorithms for performing symbolic abstraction:

- The RSY algorithm: a framework for computing $\widehat{\alpha}$ that applies to any logic and abstract domain that satisfies certain conditions (Reps et al., 2004).
- The KS algorithm: an algorithm for computing $\widehat{\alpha}$ that applies to QFBV logic and the domain $\mathcal{E}_{2^w}$ of affine equalities. King and Søndergaard (2010) gave a specific $\widehat{\alpha}$ algorithm for an abstract domain of Boolean affine relations. Elder et al. (2011) extended the King and Søndergaard algorithm to the affine-equalities domain $\mathcal{E}_{2^w}$. Because the generalized algorithm is similar to the Boolean one, we refer to it as KS.

I also present the results of an experiment I carried out to compare the performance of the two algorithms. Both algorithms compute $\widehat{\alpha}(\varphi)$ via successive approximation from "below", computing a sequence of successively "larger" approximations to the set of states described by φ.

Algorithm 6: $\widetilde{\alpha}^{\uparrow}_{\text{RSY}}\langle\mathcal{L},\mathcal{A}\rangle(\varphi)$

```
1  lower ← ⊥
2
3  while true do
4
5    S ← Model(φ ∧ ¬γ̂(lower))
6    if S is TimeOut then
7      return ⊤
8    else if S is None then
9      break                          // φ ⇒ γ̂(lower)
10   else                             // S ⊭ γ̂(lower)
11     lower ← lower ⊔ β(S)
12 ans ← lower
13 return ans
```

Algorithm 7: $\widetilde{\alpha}^{\uparrow}_{\text{KS}}(\varphi)$

```
1  lower ← ⊥
2  i ← 1
3  while i ≤ rows(lower) do
4    p ← Row(lower, −i)               // p ⊒ lower
5    S ← Model(φ ∧ ¬γ̂(p))
6    if S is TimeOut then
7      return ⊤
8    else if S is None then
9      i ← i + 1                      // φ ⇒ γ̂(p)
10   else                             // S ⊭ γ̂(p)
11     lower ← lower ⊔ β(S)
12 ans ← lower
13 return ans
```

The purpose and contributions of this chapter can be summarized as follows:

1. To present algorithms for performing symbolic abstraction that are representative of the state-of-the-art prior to this thesis (Sections 4.1 and 4.2).

2. To present the RSY and KS algorithms in a common setting (Algorithms 6 and 7).

3. To present an empirical comparison of the RSY and KS algorithms (Section 4.3).

The results of item 3 served as motivation for the new algorithms that I developed, described in Chapters 5 and 6.

4.1 RSY Algorithm

Reps et al. (2004) presented a framework for computing $\widehat{\alpha}$—which we call the RSY algorithm—that applies to any logic $\mathcal{L}$ and abstract domain $\mathcal{A}$ that satisfy certain conditions. The key insight of the algorithm is the use an SMT solver for $\mathcal{L}$ as a black-box to query for models of φ. Recall the following property of $\widehat{\alpha}$ (Theorem 3.14):

$$\widehat{\alpha}(\varphi) = \bigsqcup\{\beta(S) \mid S \models \varphi\} \tag{4.1}$$

Equation (4.1) states that $\widehat{\alpha}(\varphi)$ is the result of performing a join of the abstraction of models of φ. Unfortunately, Equation (4.1) does not directly lead to an algorithm for computing $\widehat{\alpha}(\varphi)$, because, as stated, it involves finding *all* models of φ, which would be impractical. The RSY algorithm does not find all models of φ; instead the algorithm queries the SMT solver to compute a *finite* sequence $S_1, S_2, \ldots, S_{k-1}, S_k$ of models of φ. This sequence of models is used to compute a sequence of abstract values $A_0, A_1, A_2, \ldots, A_{k-1}, A_k$ in the following way:

$$A_0 = \bot \tag{4.2}$$

$$A_i = A_{i-1} \sqcup \beta(S_i), S_i \models \varphi, \;\; 1 \leq i \leq k \tag{4.3}$$

Merely sampling k arbitrary models of φ would not work. In particular, it is possible that $A_{i-1} = A_i$ in Equation (4.3) in which case step i has not made progress. Thus, to ensure progress we require S_i to be a model of φ such that $S_i \notin \gamma(A_{i-1})$ in Equation (4.3). In other words, S_i should be a model satisfying $\varphi \wedge \neg\widehat{\gamma}(A_{i-1})$. Computing S_i in such a way ensures that $A_{i-1} \sqsubsetneq A_i$ in Equation (4.3). Equations (4.2) and (4.3) can be restated as:

$$A_0 = \bot \tag{4.4}$$

$$A_i = A_{i-1} \sqcup \beta(S_i), \; S_i \models \varphi \wedge \neg\widehat{\gamma}(A_{i-1}), \;\; 1 \leq i \leq k \tag{4.5}$$

Furthermore, provided the abstract domain $\mathcal{A}$ has no infinite ascending chains, there exists a k such that $A_k = \widehat{\alpha}(\varphi)$, because each step is guaranteed to make progress up the lattice. That is, using Equations (4.4) and (4.5), we can construct a finite sequence of abstract values that form an ascending chain that converges to $\widehat{\alpha}(\varphi)$:

$$\bot = A_0 \sqsubset A_1 \sqsubset A_2 \sqsubset \ldots \sqsubset A_{k-1} \sqsubset A_k = \widehat{\alpha}(\varphi). \tag{4.6}$$

Algorithm 6 shows the general RSY algorithm: $\widetilde{\alpha}^{\uparrow}_{\text{RSY}}\langle\mathcal{L}, \mathcal{A}\rangle$ is parameterized on logic $\mathcal{L}$ and abstract domain $\mathcal{A}$. The algorithm assumes there is a decision procedure for logic $\mathcal{L}$, and uses notation introduced in Section 3.3.

The algorithm implements Equations (4.4) and (4.5): the algorithm maintains an under-approximation of the final answer in the variable *lower*, which is initialized to $\bot$ on line 1, and updated by performing a join (line 11). $\widehat{\gamma}(\mathit{lower})$ is used as a "blocking clause", and thus the model S of φ returned on line 5 is one that is not already represented by *lower* (i.e., S is not in the concretization of *lower*). Consequently, $\mathit{lower} \sqsubsetneq \mathit{lower} \sqcup \beta(S)$, and the updated value of *lower* on line 11 is a strictly larger approximation to the set of states described by φ. Thus, provided the abstract domain $\mathcal{A}$ has no infinite ascending chains, Algorithm 6 eventually terminates when the call to `Model` on line 5 returns `None`. In this case, $\mathit{lower} = \widehat{\alpha}(\varphi)$.

Note that if the call to `Model` returns `TimeOut`, then Algorithm 6 returns $\top$ (line 6), which is a trivial, but sound, over-approximation of $\widehat{\alpha}(\varphi)$. In other words, the RSY algorithm implements $\widetilde{\alpha}(\varphi)$; however, if in a given run of the algorithm there is no timeout, the value returned equals $\widehat{\alpha}(\varphi)$. In this sense, all of the algorithms described in the thesis are implementations of $\widehat{\alpha}$.

By looking at the pseudo-code in Algorithm 6, we can determine the assumptions on the logic and the abstract domain made by the RSY algorithm:

1. There is a Galois connection $\mathcal{C} \xleftarrow[\alpha]{\gamma} \mathcal{A}$ between $\mathcal{A}$ and concrete domain $\mathcal{C}$, and an implementation of the corresponding representation function β (see line 11).
2. There is an algorithm to evaluate $a_1 \sqcup a_2$ for all $a_1, a_2 \in \mathcal{A}$ (see line 11).
3. There is a symbolic-concretization operation $\widehat{\gamma}$ that maps an abstract value $a \in \mathcal{A}$ to a formula $\widehat{\gamma}(a)$ in $\mathcal{L}$ (see line 5).
4. $\mathcal{A}$ has no infinite ascending chains.
5. There is a decision procedure for the logic $\mathcal{L}$ that is also capable of returning a model satisfying a formula in $\mathcal{L}$ (see line 5).
6. The logic $\mathcal{L}$ is closed under conjunction and negation (see line 5).

I omit the formal proof of the correctness of the algorithm. The proof can be found in Reps et al. (2004). Furthermore, the correctness of Bilateral algorithm, which is a generalization of the RSY algorithm, will be given in Chapter 5.

Algorithm 8: $\widehat{\text{Post}}^{\uparrow}[\tau](v)$

```
1  lower' ← ⊥
2  while true do
3      ⟨S, S'⟩ ← Model(γ̂(v) ∧ φ_τ ∧ ¬γ̂(lower'))
4      if ⟨S, S'⟩ is TimeOut then
5          return ⊤
6      else if ⟨S, S'⟩ is None then
7          break                                   // Post̂[τ](v) = lower'
8      else                                        //  S' ⊭ γ̂(lower')
9          lower' ← lower' ⊔ β(S')
10 v' ← lower'
11 return v'
```

4.1.1 Computing $\widehat{\text{Post}}$

Algorithm 6, which computes the symbolic abstraction of a formula, can be adapted to compute $\widehat{\text{Post}}$ for a concrete transformer τ and input abstract value $a \in AbsDomain$. Recall that $\widehat{\text{Post}}$ satisfies $\widehat{\text{Post}}[\tau](a) = (\alpha \circ \text{Post}[\tau] \circ \gamma)(a)$. Algorithm 8 shows the algorithm that computes $\widehat{\text{Post}}$. φ_τ represents the formula in logic $\mathcal{L}$ that captures the input-output relation for the concrete transformer τ. The main difference between Algorithm 6 and Algorithm 8 is call to `Model` on line 3. Algorithm 8 will be used in Chapter 7.

4.2 KS Algorithm

Algorithm 7 presents the $\widetilde{\alpha}^{\uparrow}_{\text{KS}}$ algorithm for computing $\widehat{\alpha}$ that applies to the QFBV logic and affine-equalities domain $\mathcal{E}_{2^w}$. King and Søndergaard (2010) first presented the algorithm for abstract domain for Boolean affine relations, which was generalized to the domain of affine equalities by Elder et al. (2011).

An abstract value in the $\mathcal{E}_{2^w}$ domain is a conjunction of affine equalities, which can be represented in a normal form as a matrix in which each row expresses a non-redundant affine equality (Elder et al., 2011). (Rows are 0-indexed.) Given a matrix m, $\texttt{rows}(m)$ returns the number of rows of m (as in line 3 in Algorithm 7), and $\texttt{Row}(m, -i)$, for $1 \leq i \leq \texttt{rows}(m)$, returns row $(\texttt{rows}(m) - i)$ of m (as in line 4 in Algorithm 7).

The presentation in Algorithm 7 differs significantly from that in (King and Søndergaard, 2010, figure 2); in particular, Algorithm 7 is stated using the terminology of symbolic abstract interpretation (Section 3.2.3). The similarities between the RSY algorithm (Algorithm 6) and the KS algorithm are easier to see in this presentation. Both algorithms have a similar overall structure. Both are successive approximation algorithms: they compute a sequence of successively "larger" approximations to the set of states described by φ. Both maintain an under-approximation of the final answer in the variable *lower*, which is initialized to $\bot$ on line 1. Both call a decision procedure (line 5), and if a model S is found that satisfies the query, the under-approximation is updated by performing a join (line 11).

The RSY and KS algorithms have a similar overall structures (Algorithms 6 and 7). The differences between Algorithms 6 and 7 are highlighted in gray. The key difference is the nature of the decision-procedure query on line 5. $\widetilde{\alpha}^{\uparrow}_{\text{RSY}}$ uses *all* of *lower* to construct the query, while $\widetilde{\alpha}^{\uparrow}_{\text{KS}}$ uses only a single row from *lower* (line 4)—i.e., just a *single affine equality*, which has two consequences. First, $\widetilde{\alpha}^{\uparrow}_{\text{KS}}$ should issue a larger number of queries than $\widetilde{\alpha}^{\uparrow}_{\text{RSY}}$, for the following reason. Suppose that the value of *lower* has converged to the final answer via a sequence of joins performed by the algorithm. To discover that convergence has occurred, $\widetilde{\alpha}^{\uparrow}_{\text{RSY}}$ has to issue just a single decision-procedure query, whereas $\widetilde{\alpha}^{\uparrow}_{\text{KS}}$ has to confirm it by issuing $\texttt{rows}(\textit{lower}) - i$ number of queries, proceeding row-by-row. Second, each individual query issued by $\widetilde{\alpha}^{\uparrow}_{\text{KS}}$ is simpler than the ones issued by $\widetilde{\alpha}^{\uparrow}_{\text{RSY}}$. Thus, *a priori*, it is not clear which algorithm will perform better in practice; an empirical comparison of the RSY and KS algorithms is provided in Section 4.3.

I omit the formal proof of the correctness of the KS algorithm. The proof can be found in Elder et al. (2014). Furthermore, the correctness of Bilateral algorithm, which is a generalization of the KS algorithm, will be given in Chapter 5.

4.3 Empirical Comparison of the RSY and KS Algorithms

In this section, I provide an quantitative comparison of the performance of the KS algorithm and the RSY framework instantiated for the affine-equalities domain $\mathcal{E}_{2^{32}}$. We use RSY$[\mathcal{E}_{2^{32}}]$ to

name	instrs	procs	BBs	brs
finger	532	18	298	48
subst	1093	16	609	74
label	1167	16	573	103
chkdsk	1468	18	787	119
convert	1927	38	1013	161
route	1982	40	931	243
logoff	2470	46	1145	306
setup	4751	67	1862	589

Table 4.1: The characteristics of the x86 binaries of Windows utilities used in the experiments. The columns show the number of instructions (instrs); the number of procedures (procs); the number of basic blocks (BBs); the number of branch instructions (brs).

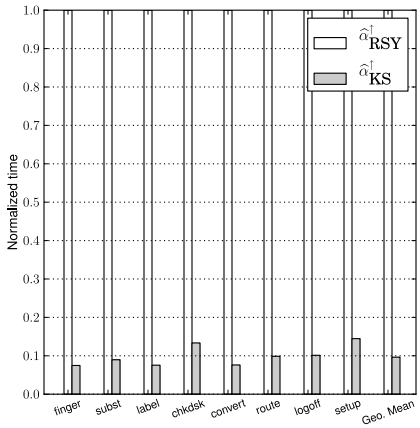


Figure 4.1: Total time taken by all invocations of $\widetilde{\alpha}^{\uparrow}_{\mathrm{RSY}}[\mathcal{E}_{2^{32}}]$ compared to that taken by $\widetilde{\alpha}^{\uparrow}_{\mathrm{KS}}$ for each of the benchmark executables. The running time is normalized to the corresponding time taken by $\widetilde{\alpha}^{\uparrow}_{\mathrm{RSY}}[\mathcal{E}_{2^{32}}]$; lower numbers are better.

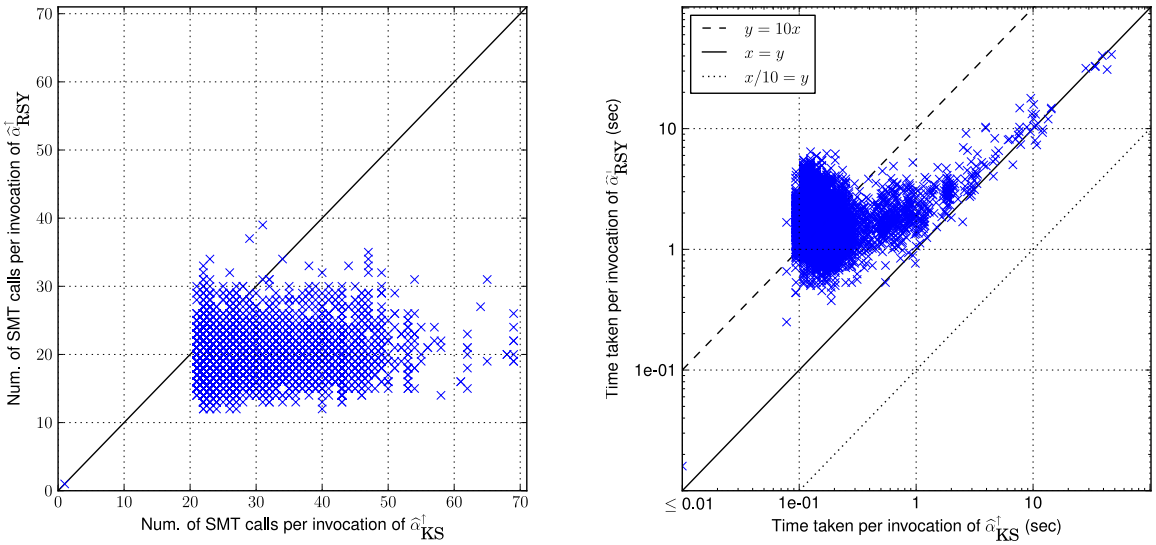
denote the RSY framework instantiated for the affine-equalities domain $\mathcal{E}_{2^{32}}$.

The experiments in this section were designed to answer the following questions:

1. How does the speed of $\widetilde{\alpha}^{\uparrow}_{\mathrm{KS}}$ compare with that of $\widetilde{\alpha}^{\uparrow}_{\mathrm{RSY}}[\mathcal{E}_{2^{32}}]$?

2. What are the reasons for the differences between the running times of $\widetilde{\alpha}^{\uparrow}_{\mathrm{KS}}$ and $\widetilde{\alpha}^{\uparrow}_{\mathrm{RSY}}[\mathcal{E}_{2^{32}}]$?

To address these questions, we performed affine-relations analysis (ARA) on x86 machine code, computing affine relations over the x86 registers. Our experiments were run on a single



(a) (b)

Figure 4.2: (a) Scatter plot showing of the number of decision-procedure queries during each pair of invocations of $\widetilde{\alpha}^{\uparrow}_{\mathrm{RSY}}$ and $\widetilde{\alpha}^{\uparrow}_{\mathrm{KS}}$, when neither invocation had a decision-procedure timeout. (b) Log-log scatter plot showing the times taken by each pair of invocations of $\widetilde{\alpha}^{\uparrow}_{\mathrm{RSY}}$ and $\widetilde{\alpha}^{\uparrow}_{\mathrm{KS}}$, when neither invocation had a decision-procedure timeout.

core of a quad-core 3.0 GHz Xeon computer running 64-bit Windows XP (SP2), configured so that a user process has 4GB of memory. We analyzed a corpus of Windows utilities (Table 4.1) using the WALi (Kidd et al., 2007) system for weighted pushdown systems (WPDSs). [1] For the $\widetilde{\alpha}^{\uparrow}_{\mathrm{KS}}$-based ($\widetilde{\alpha}^{\uparrow}_{\mathrm{RSY}}$-based) analysis we used a weight domain of $\widetilde{\alpha}^{\uparrow}_{\mathrm{KS}}$-generated ($\widetilde{\alpha}^{\uparrow}_{\mathrm{RSY}}$-generated) ARA transformers. The weight on each WPDS rule encodes the ARA transformer for a basic block B of the program, including a jump or branch to a successor block. A formula φ_B is created that captures the concrete semantics of B, and then the ARA weight for B is obtained by performing $\widehat{\alpha}(\varphi_B)$. We compared the time for $\widetilde{\alpha}^{\uparrow}_{\mathrm{RSY}}[\mathcal{E}_{2^{32}}]$ and $\widetilde{\alpha}^{\uparrow}_{\mathrm{KS}}$ to compute basic-block transformers for this set of x86 executables. There was no overall timeout imposed on the invocation of the procedures, but each invocation of the decision procedure (line 5 in Algorithms 6 and 7) had a timeout of 3

[1] Due to the high cost of the ARA-based WPDS construction, all analyses excluded the code for libraries. Because register `eax` holds the return value from a call, library functions were modeled approximately (albeit unsoundly, in general) by "`havoc(eax)`".

seconds (as in Elder et al. (2011)).

Figure 4.1 shows the total time taken by all invocations of $\widetilde{\alpha}^{\uparrow}_{\text{RSY}}$ compared to that taken by $\widetilde{\alpha}^{\uparrow}_{\text{KS}}$ for each of the benchmark executables. The running time is normalized to the corresponding time taken by $\widetilde{\alpha}^{\uparrow}_{\text{RSY}}$; lower numbers are better. Overall, $\widetilde{\alpha}^{\uparrow}_{\text{KS}}$ is about ten times faster compared to $\widetilde{\alpha}^{\uparrow}_{\text{RSY}}$; this answers the question posed in item 1 above.

Figure 4.2(a) shows a scatter-plot of the *number of decision-procedure calls* in each invocation of $\widetilde{\alpha}^{\uparrow}_{\text{RSY}}$ versus the corresponding invocation of $\widetilde{\alpha}^{\uparrow}_{\text{KS}}$, when neither of the procedures had a decision-procedure timeout. $\widetilde{\alpha}^{\uparrow}_{\text{RSY}}$ issues fewer decision-procedure queries: on average (computed as an arithmetic mean), $\widetilde{\alpha}^{\uparrow}_{\text{KS}}$ invokes 42% more calls to the decision procedure. Figure 4.2(b) shows a log-log scatter-plot of the *total time* taken by each invocation of $\widetilde{\alpha}^{\uparrow}_{\text{RSY}}$ versus the time taken by $\widetilde{\alpha}^{\uparrow}_{\text{KS}}$. $\widetilde{\alpha}^{\uparrow}_{\text{KS}}$ is much faster than $\widetilde{\alpha}^{\uparrow}_{\text{RSY}}$: overall, computed as the geometric mean of the speedups on each of the x86 executables, $\widetilde{\alpha}^{\uparrow}_{\text{KS}}$ is about ten times faster compared to $\widetilde{\alpha}^{\uparrow}_{\text{RSY}}[\mathcal{E}_{2^{32}}]$.

The order-of-magnitude speedup can be attributed to the fact that each of the $\widetilde{\alpha}^{\uparrow}_{\text{KS}}$ decision-procedure queries is less expensive than the ones issued by $\widetilde{\alpha}^{\uparrow}_{\text{RSY}}$ even though $\widetilde{\alpha}^{\uparrow}_{\text{KS}}$ issues more decision-procedure calls. At line 4 in $\widetilde{\alpha}^{\uparrow}_{\text{KS}}$, p is a single constraint; consequently, the decision-procedure query contains the *single* conjunct $\neg\widehat{\gamma}(p)$ (line 5). In contrast, at line 5 in $\widetilde{\alpha}^{\uparrow}_{\text{RSY}}$, $lower$ is a *conjunction* of constraints, and consequently the decision-procedure query contains $\neg\widehat{\gamma}(lower)$, which is a *disjunction* of constraints.

Chapter 5

A Bilateral Algorithm for Symbolic Abstraction

> Great results can be achieved with small forces.
>
> — Sun Tzu, *The Art of War*

In this chapter, I use the insights gained from the algorithms presented in Chapter 4 to design a new *framework* for symbolic abstraction that

- is *parametric* and is applicable to any abstract domain that satisfies certain conditions (similar to the RSY algorithm)
- uses a successive-approximation algorithm that is *parsimonious* in its use of the decision procedure (similar to the KS algorithm)
- is *bilateral*; that is, it maintains both an under-approximation and a (non-trivial) over-approximation of the desired answer, and hence is *resilient to timeouts*: the procedure can return the over-approximation if it is stopped at any point (unlike the RSY and KS algorithms).

In contrast, neither the RSY algorithm nor KS algorithm is resilient to timeouts. A decision-procedure query—or the cumulative time for $\widetilde{\alpha}^{\uparrow}$—might take too long, in which case the only safe answer that can be returned is $\top$ (line 6 in Algorithms 6 and 7).

Algorithm		Parametric	Resilient	Parsimonious
RSY	(Section 4.1)	✓	✗	✗
KS	(Section 4.2)	✗	✗	✓
Bilateral	(Section 5.2)	✓	✓	✓

Table 5.1: Qualitative comparison of symbolic-abstraction algorithms. A ✓ indicates that the algorithm has a property described by a column, and a ✗ indicates that the algorithm does not have the property.

Table 5.1 compares the features of the various algorithms for symbolic abstraction. Thus, the Bilateral algorithm combines the best features of the KS and RSY algorithms, but also has benefits that none of these previous algorithms have.

The contributions of this chapter can be summarized as follows:

- I show how the KS algorithm can be modified into the KS$^+$ algorithm, which maintains sound under- and over-approximations of the answer (Section 5.1).
- I present a framework for symbolic abstraction based on a *bilateral algorithm* for computing $\widehat{\alpha}$ (Section 5.2).
- I extend the bilateral algorithm to handle abstract domains with infinite descending chains (Section 5.3).
- I give several instantiations of the bilateral framework (Section 5.4).
- I compare the performance of the KS algorithm and an instantiation of the Bilateral framework (Section 5.5).

Section 5.6 describes a Bilateral algorithm for computing $\widehat{\mathrm{Post}}$. Section 5.7 presents related work.

5.1 Towards a Bilateral Algorithm

In this section, I describe how the KS algorithm (Algorithm 7) can be made resilient to timeouts. In particular, I show how $\widetilde{\alpha}^{\uparrow}_{\mathrm{KS}}$ can be modified to maintain a non-trivial over-approximation

Algorithm 9: $\widetilde{\alpha}^{\uparrow}_{\text{KS}}(\varphi)$

```
1
2  lower ← ⊥
3  i ← 1
4  while i ≤ rows(lower) do
5    p ← Row(lower, −i)
// p ⊒ lower
6    S ← Model(φ ∧ ¬γ̂(p))
7    if S is TimeOut then
8      return ⊤
9    else if S is None then
                                   // φ ⇒ γ̂(p)
10     i ← i + 1
11   else                          // S ⊭ γ̂(p)
12     lower ← lower ⊔ β(S)
13 ans ← lower
14 return ans
```

Algorithm 10: $\widetilde{\alpha}^{\updownarrow}_{\text{KS}^+}(\varphi)$

```
1  upper ← ⊤
2  lower ← ⊥
3  i ← 1
4  while i ≤ rows(lower) do
5    p ← Row(lower, −i)
// p ⊒ lower,  p ⋣ upper
6    S ← Model(φ ∧ ¬γ̂(p))
7    if S is TimeOut then
8      return upper
9    else if S is None then
10     upper ← upper ⊓ p  // φ ⇒ γ̂(p) i ← i + 1
11   else                          // S ⊭ γ̂(p)
12     lower ← lower ⊔ β(S)
13 ans ← lower
14 return ans
```

of the desired answer. Algorithm 10 is thus a *bilateral* algorithm: it maintains both an under-approximation and over-approximation of $\widehat{\alpha}(\varphi)$. The original $\widetilde{\alpha}^{\uparrow}_{\text{KS}}$ is shown in Algorithm 9 for comparison; the differences in the algorithms are highlighted in gray. (Note that line numbers are different in Algorithms 7 and 9.)

The $\widetilde{\alpha}^{\updownarrow}_{\text{KS}^+}$ algorithm (Algorithm 10) initializes the over-approximation (*upper*) to $\top$ on line 1. At any stage in the algorithm $\varphi \Rightarrow \widehat{\gamma}(\mathit{upper})$. On line 10, it is sound to update *upper* by performing a meet with p because $\varphi \Rightarrow \widehat{\gamma}(p)$. Progress is guaranteed because $p \not\sqsupseteq \mathit{upper}$. In case of a decision-procedure timeout (line 7), Algorithm 10 returns *upper* as the answer (line 8). Thus, $\widetilde{\alpha}^{\updownarrow}_{\text{KS}^+}(\varphi)$ can return a non-trivial over-approximation of $\widehat{\alpha}(\varphi)$ in case of a timeout. However, if the loop exits without a timeout, then $\widetilde{\alpha}^{\updownarrow}_{\text{KS}^+}(\varphi)$ returns $\widehat{\alpha}(\varphi)$.

5.2 A Parametric Bilateral Algorithm

Like the original KS algorithm, $\widetilde{\alpha}^{\updownarrow}_{\text{KS}^+}$ applies only to the affine-equalities domain $\mathcal{E}_{2^w}$. The results presented in Section 4.3 provide motivation to generalize $\widetilde{\alpha}^{\updownarrow}_{\text{KS}^+}$ so that we can take advantage of its benefits with domains other than $\mathcal{E}_{2^w}$. In this section, I present the bilateral framework,

which applies to any abstract domain that satisfies the interface defined below. (Note that the interface for the Bilateral framework is slightly different than the interface for the RSY framework presented in Section 4.1.)

I first introduce the *abstract-consequence* operation, which is the key operation in our generalized algorithm:

Definition 5.1. An operation $\texttt{AbstractConsequence}(\cdot, \cdot)$ is an *acceptable abstract-consequence operation* iff for all $a_1, a_2 \in \mathcal{A}$ such that $a_1 \sqsubsetneq a_2$,
$a = \texttt{AbstractConsequence}(a_1, a_2)$ implies $a_1 \sqsubseteq a$ and $a \not\sqsupseteq a_2$. ■

Figure 5.1 illustrates Definition 5.1 graphically, using the concretizations of a_1, a_2, and a.

Algorithm 11 presents the parametric bilateral algorithm $\widetilde{\alpha}^{\updownarrow}\langle\mathcal{L}, \mathcal{A}\rangle(\varphi)$, which performs symbolic abstraction of $\varphi \in \mathcal{L}$ for abstract domain $\mathcal{A}$. The differences between Algorithms 11 and 10 are highlighted in gray.

The assumptions placed on the logic and the abstract domain are as follows:

1. There is a Galois connection $\mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$ between $\mathcal{A}$ and concrete domain $\mathcal{C}$, and an implementation of the corresponding representation function β.

2. Given $a_1, a_2 \in \mathcal{A}$, there are algorithms to evaluate $a_1 \sqcup a_2$ and $a_1 \sqcap a_2$, and to check $a_1 = a_2$.

3. There is a symbolic-concretization operation $\widehat{\gamma}$ that maps an abstract value $a \in \mathcal{A}$ to a formula $\widehat{\gamma}(a)$ in $\mathcal{L}$.

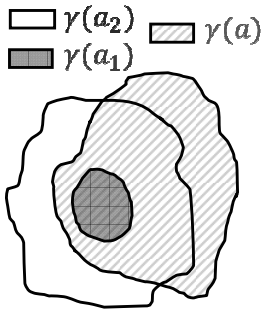


Figure 5.1: Abstract Consequence: For all $a_1, a_2 \in \mathcal{A}$ where $\gamma(a_1) \subsetneq \gamma(a_2)$, if $a = \texttt{AbstractConsequence}(a_1, a_2)$, then $\gamma(a_1) \subseteq \gamma(a)$ and $\gamma(a) \not\supseteq \gamma(a_2)$.

Algorithm 11: $\widehat{\alpha}^{\updownarrow}\langle\mathcal{L},\mathcal{A}\rangle(\varphi)$

```
1  upper ← ⊤
2  lower ← ⊥
3  while lower ≠ upper ∧ ResourcesLeft do
   // lower ⊏ upper
4      p ← AbstractConsequence(lower, upper)
   // p ⊒ lower, p ⋣ upper
5      S ← Model(φ ∧ ¬γ̂(p))
6      if S is TimeOut then
7          return upper
8      else if S is None then                          // φ ⇒ γ̂(p)
9          upper ← upper ⊓ p
10     else                                            // S ⊭ γ̂(p)
11         lower ← lower ⊔ β(S)
12 ans ← upper
13 return ans
```

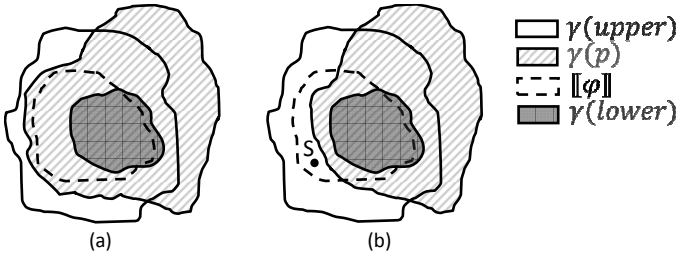


Figure 5.2: The two cases arising in Algorithm 11: $\varphi \wedge \neg\widehat{\gamma}(p)$ is either (a) unsatisfiable, or (b) satisfiable with $S \models \varphi$ and $S \not\models \widehat{\gamma}(p)$. (Note that although $\mathit{lower} \sqsubseteq \widehat{\alpha}(\varphi) \sqsubseteq \mathit{upper}$ and $[\![\varphi]\!] \subseteq \gamma(\mathit{upper})$ are invariants of Algorithm 11, $\gamma(\mathit{lower}) \subseteq [\![\varphi]\!]$ does not necessarily hold, as depicted above.)

4. There is a decision procedure for the logic $\mathcal{L}$ that is also capable of returning a model satisfying a formula in $\mathcal{L}$.

5. The logic $\mathcal{L}$ is closed under conjunction and negation.

6. There is an acceptable abstract-consequence operation for $\mathcal{A}$ (Definition 5.1).

The abstract value p returned by `AbstractConsequence` (line 4 of Algorithm 11) is used to generate the decision-procedure query (line 5); Figure 5.2 illustrates the two cases arising based on whether $\varphi \wedge \neg\widehat{\gamma}(p)$ is satisfiable or unsatisfiable.

If a model S is found satisfying the query, the Bilateral algorithm updates the under-approximation by performing a join (line 11). The correctness of this update of *lower* via join is based on Theorem 3.14.

The Bilateral algorithm also maintains an over-approximation of the final answer in the variable *upper*, which is initialized to $\top$ (line 1). When the decision procedure determines that the query on line 5 is unsatisfiable the Bilateral algorithm refines the value of *upper* by performing a meet, as shown on line 9. The correctness of this update of *upper* via a meet is based on Theorem 3.15.

The overall resources used by Algorithm 11, such as time, can be controlled via the ResourcesLeft flag (line 3). If resources run out, the Bilateral algorithm returns the over-approximation computed in *upper* (line 7).

Theorem 5.2. *Suppose that $\mathcal{L}$ and $\mathcal{A}$ satisfy requirements 1–6, and $\varphi \in \mathcal{L}$. Let $a \in \mathcal{A}$ be the value returned by $\widetilde{\alpha}^{\updownarrow}\langle \mathcal{L}, \mathcal{A} \rangle(\varphi)$. Then*

1. *a over-approximates $\widehat{\alpha}(\varphi)$; i.e., $\widehat{\alpha}(\varphi) \sqsubseteq a$.*
2. *If $\mathcal{A}$ has neither infinite ascending nor infinite descending chains and the run of $\widetilde{\alpha}^{\updownarrow}\langle \mathcal{L}, \mathcal{A} \rangle(\varphi)$ returns without ever having a timeout, then $a = \widehat{\alpha}(\varphi)$.*

Proof. To prove part 1, we show that at each stage of Algorithm 11 *lower* $\sqsubseteq \widehat{\alpha}(\varphi) \sqsubseteq$ *upper* holds. This invariant is trivially true after *upper* and *lower* are initialized in lines 1 and 2, respectively.

If control reaches line 4, then *lower* $\neq$ *upper* and *timeout* is false. Hence, at line 4, *lower* $\sqsubsetneq$ *upper*. Thus, the precondition for the call to `AbstractConsequence`(*lower*, *upper*) is satisfied, and the abstract value p returned is such that *lower* $\sqsubseteq p$ and $p \not\sqsubseteq$ *upper* (by Definition 5.1).

A Galois connection $\mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$ obeys the adjointness condition

$$\text{for all } c \in \mathcal{C}, a \in \mathcal{A} : c \sqsubseteq \gamma(a) \text{ iff } \alpha(c) \sqsubseteq a. \tag{5.1}$$

The counterpart of Equation (5.1) for symbolic abstraction is

$$\text{for all } \varphi \in \mathcal{L}, a \in \mathcal{A} : \varphi \Rightarrow \widehat{\gamma}(a) \text{ iff } \widehat{\alpha}(\varphi) \sqsubseteq a. \tag{5.2}$$

If control reaches line 9, then $\varphi \wedge \neg\widehat{\gamma}(p)$ is unsatisfiable (Figure 5.2(a)), which means that $\varphi \Rightarrow \widehat{\gamma}(p)$ holds. Consequently, by Equation (5.2), we know that $\widehat{\alpha}(\varphi) \sqsubseteq p$ holds. By properties of meet ($\sqcap$), we can combine the latter inequality with the invariant $\widehat{\alpha}(\varphi) \sqsubseteq \mathit{upper}$ to obtain $\widehat{\alpha}(\varphi) \sqsubseteq \mathit{upper} \sqcap p$. Hence it is safe to update *upper* by performing a meet with p; that is, after the assignment $\mathit{upper} \leftarrow \mathit{upper} \sqcap p$ on line 9, the invariant $\widehat{\alpha}(\varphi) \sqsubseteq \mathit{upper}$ still holds.

On the other hand, if $\varphi \wedge \neg\widehat{\gamma}(p)$ is satisfiable (Figure 5.2(b)), then at line 11 $S \models \varphi$. Thus, $\beta(S) \sqsubseteq \widehat{\alpha}(\varphi)$. By properties of join ($\sqcup$), we can combine the latter inequality with the invariant $\mathit{lower} \sqsubseteq \widehat{\alpha}(\varphi)$ to obtain $\mathit{lower} \sqcup \beta(S) \sqsubseteq \widehat{\alpha}(\varphi)$. Hence it is safe to update *lower* by performing a join with $\beta(S)$; that is, after the assignment $\mathit{lower} \leftarrow \mathit{lower} \sqcup \beta(S)$ on line 11, the invariant $\mathit{lower} \sqsubseteq \widehat{\alpha}(\varphi)$ still holds.

In both cases, $\mathit{lower} \sqsubseteq \widehat{\alpha}(\varphi) \sqsubseteq \mathit{upper}$ holds, and thus $\mathit{lower} \sqsubseteq \widehat{\alpha}(\varphi) \sqsubseteq \mathit{upper}$ holds throughout the loop on lines 3–11.

On exiting the loop, we have $\widehat{\alpha}(\varphi) \sqsubseteq \mathit{upper}$. At line 12, *ans* is assigned the value of *upper*, which is the value returned by Algorithm 11 at line 13. This finishes the proof of part 1.

We now prove part 2 of the theorem.

- At line 9, because $p \not\sqsupseteq \mathit{upper}$, $\mathit{upper} \sqcap p$ does not equal *upper*; that is, $\mathit{upper} \sqcap p \sqsubsetneq \mathit{upper}$.
- At line 11, $S \not\models \widehat{\gamma}(p)$. Because $p \sqsupseteq \mathit{lower}$, $S \not\models \widehat{\gamma}(p)$ implies that $S \not\models \widehat{\gamma}(\mathit{lower})$, and hence $\beta(S) \not\sqsubseteq \mathit{lower}$. Therefore, $\mathit{lower} \sqcup \beta(S)$ is not equal to *lower*; that is, $\mathit{lower} \sqcup \beta(S) \sqsupsetneq \mathit{lower}$.

Consequently, progress is made no matter which branch of the if-then-else on lines 8–11 is taken, and hence Algorithm 11 makes progress during each iteration of the while-loop.

By part 1, $\mathit{lower} \sqsubseteq \widehat{\alpha}(\varphi) \sqsubseteq \mathit{upper}$. Consequently, if $\mathcal{A}$ has neither infinite ascending nor infinite descending chains, then eventually *lower* will be equal to *upper*, and both *lower* and *upper* will

Algorithm 12: `AbstractConsequence`(a_1, a_2) for conjunctive domains

```
1 if a_1 = ⊥ then return ⊥
2 Let Ψ ⊆ Φ be the set of formulas such that γ̂(a_1) = ⋀ Ψ
3 foreach ψ ∈ Ψ do
4     a ← μα̂(ψ)
5     if a ⋢ a_2 then return a
```

have the value $\widehat{\alpha}(\varphi)$ (provided the loop exits without a timeout). Thus, for a run of Algorithm 11 on which the loop exits without a timeout, the answer returned is $\widehat{\alpha}(\varphi)$. □

5.2.1 Relation Between RSY and Bilateral Algorithms

Definition 5.1 allows `AbstractConsequence`(a_1, a_2) to return any $a \in \mathcal{A}$ as long as a satisfies $a_1 \sqsubseteq a$ and $a \not\sqsupseteq a_2$. Thus, for a given abstract domain $\mathcal{A}$ there could be multiple implementations of the `AbstractConsequence` operation. In particular, `AbstractConsequence`(a_1, a_2) can return a_1, because $a_1 \sqsubseteq a_1$ and $a_1 \not\sqsupseteq a_2$. If this particular implementation of `AbstractConsequence` is used, then Algorithm 11 reduces to the RSY algorithm (Algorithm 6). However, as illustrated in Section 4.3, the decision-procedure queries issued by the RSY algorithm can be very expensive.

5.2.2 Abstract Consequence for Conjunctive Domains

Algorithm 12 presents an implementation of `AbstractConsequence` for *conjunctive domains* (Definition 3.5). The benefit of Algorithm 12 is that it causes Algorithm 11 to issue the kind of inexpensive queries that we see in $\widetilde{\alpha}^{\uparrow}_{\mathrm{KS}}$. Note that $a_1 \sqcup a_2 = a_2$ iff $a_1 \sqsubseteq a_2$ iff $a_1 \sqcap a_2 = a_1$, so by Assumption 2 of the bilateral framework, a comparison test for use in line 5 of Algorithm 12 is always available in a conjunctive domain that satisfies the requirements of the bilateral framework.

Theorem 5.3. *When $\mathcal{A}$ is a conjunctive domain over Φ, Algorithm 12 is an acceptable abstract-consequence operation.*

Proof. Suppose that $a_1 \sqsubsetneq a_2$, and let $\widehat{\gamma}(a_1) = \bigwedge \Psi$, where $\Psi \subseteq \Phi$. If for each $\psi \in \Psi$ we have $\widehat{\gamma}(a_2) \Rightarrow \psi$, then $\widehat{\gamma}(a_2) \Rightarrow \bigwedge\{\psi\}$, or equivalently $\widehat{\gamma}(a_2) \Rightarrow \bigwedge \Psi$; i.e., $\widehat{\gamma}(a_2) \Rightarrow \widehat{\gamma}(a_1)$, or equivalently $a_2 \sqsubseteq a_1$, which contradicts $a_1 \sqsubsetneq a_2$. Thus, there must exist some $\psi \in \Psi$ such that $\widehat{\gamma}(a_2) \not\Rightarrow \psi$. The

latter is equivalent to $\psi \not\Leftarrow \widehat{\gamma}(a_2)$, which can be written as $a \not\sqsupseteq a_2$ (where $a = \mu\widehat{\alpha}(\psi)$). Therefore, Algorithm 12 will return some $a \in \mathcal{A}$ such that $a_1 \sqsubseteq a$ and $a \not\sqsupseteq a_2$. □

If there are also algorithms for join and meet in conjunctive domain $\mathcal{A}$, and a decision procedure for the logic $\mathcal{L}$ that supplies models for satisfiable formulas, then $\mathcal{A}$ satisfies the bilateral framework, and therefore supports the $\widetilde{\alpha}^{\updownarrow}$ algorithm.

5.2.3 Increasing Resilience to Timeouts

As presented, Algorithm 11 exits and returns the value of *upper* the first time the decision procedure times out. We can improve the precision of Algorithm 11 by not exiting after the first timeout, and instead trying other abstract consequences. The algorithm will exit and return *upper* only if it cannot find an abstract consequence for which the decision-procedure terminates within the time bound. For conjunctive domains, Algorithm 11 can be modified to enumerate all conjuncts of *lower* that are abstract consequences; to implement this strategy, lines 4–7 of Algorithm 11 are replaced with

```
progress ← false                                          // Initialize progress
foreach p such that p = AbstractConsequence(lower, upper) do
  S ← Model(φ ∧ ¬γ̂(p))
  if S is not TimeOut then
    progress ← true                                       // Can make progress
    break
if ¬progress then return upper                            // Could not make progress
```

Henceforth, when we refer to $\widetilde{\alpha}^{\updownarrow}$, we mean Algorithm 11 with the above two changes.

5.2.4 Relationship of Abstract Consequence to Interpolation

To avoid the potential for confusion, we now discuss how the notion of abstract consequence differs from the well-known concept of *interpolation* (Craig, 1957):

> A logic $\mathcal{L}$ *supports interpolation* if for all $\varphi_1, \varphi_2 \in \mathcal{L}$ such that $\varphi_1 \Rightarrow \varphi_2$, there exists a formula I such that (i) $\varphi_1 \Rightarrow I$, (ii) $I \Rightarrow \varphi_2$, and (iii) I uses only symbols in the shared vocabulary of φ_1 and φ_2.

Although condition (i) is part of Definition 5.1, the restrictions imposed by conditions (ii) and (iii) are not part of Definition 5.1. To highlight the differences, we restate Definition 5.1 in terms of formulas.

> An operation $\texttt{AbstractConsequence}(\cdot, \cdot)$ is an acceptable abstract-consequence operation iff for all $a_1, a_2 \in \mathcal{A}$ such that $\widehat{\gamma}(a_1) \Rightarrow \widehat{\gamma}(a_2)$ and $\widehat{\gamma}(a_1) \not\Leftarrow \widehat{\gamma}(a_2)$,
> $a = \texttt{AbstractConsequence}(a_1, a_2)$ implies $\widehat{\gamma}(a_1) \Rightarrow \widehat{\gamma}(a)$ and $\widehat{\gamma}(a) \not\Leftarrow \widehat{\gamma}(a_2)$.

From an operational standpoint, condition (iii) in the definition of interpolation serves as a heuristic that generally allows interpolants to be expressed as small formulas. In the context of $\widetilde{\alpha}^{\updownarrow}$, we are interested in obtaining small formulas to use in the decision-procedure query (line 5 of Algorithm 11). Thus, given $a_1, a_2 \in \mathcal{A}$, it might appear plausible to use an interpolant I of $\widehat{\gamma}(a_1)$ and $\widehat{\gamma}(a_2)$ in $\widetilde{\alpha}^{\updownarrow}$ instead of the abstract consequence of a_1 and a_2. However, there are a few problems with such an approach:

- There is no guarantee that I will indeed be simple; for instance, if the vocabulary of $\widehat{\gamma}(a_1)$ is a subset of the vocabulary of $\widehat{\gamma}(a_2)$, then I could be $\widehat{\gamma}(a_1)$ itself, in which case Algorithm 11 performs the more expensive RSY iteration step.
- Converting the formula I into an abstract value $p \in \mathcal{A}$ for use in line 9 of Algorithm 11 itself requires performing $\widehat{\alpha}$ on I.

As discussed above, many domains are conjunctive domains, and for conjunctive domains is it always possible to find a *single conjunct* that is an abstract consequence (see Theorem 5.3). Moreover, such a conjunct is not necessarily an interpolant.

Algorithm 13: $\widetilde{\alpha}^{\updownarrow}_{+}\langle\mathcal{L},\mathcal{A}\rangle(\varphi)$

```
1  upper ← ⊤
2  lower ← ⊥
3  k ← 0                                                    // initialize k
4  while lower ≠ upper do
//   lower ⋢ upper
5    if k < N then
6      p ← AbstractConsequence(lower, upper)
7    else
8      p ← lower
//   p ⊒ lower, p ⋣ upper
9    S ← Model(φ ∧ ¬γ̂(p))
10   if S is TimeOut then
11     return upper
12   else if S is None then                                 // φ ⇒ γ̂(p)
13     upper ← upper ⊓ p
14     k ← k + 1                                            // increment k
15   else                                                   // S ⊭ γ̂(p)
16     lower ← lower ⊔ β(S)
17     k ← 0                                                // reset k
18 ans ← upper
19 return ans
```

5.3 Abstract Domains with Infinite Descending Chains

We can weaken part 2 of Theorem 5.2 to allow $\mathcal{A}$ to have infinite descending chains by modifying Algorithm 11 slightly. The modified algorithm has to ensure that it does not get trapped updating *upper* along an infinite descending chain, and that it exits when *lower* has converged to $\widehat{\alpha}(\varphi)$. Suppose that, for some fixed N, N consecutive iterations of the loop on lines 3–11 update *upper* (line 9) *without updating lower* (line 11). If this situation occurs, in the next iteration the algorithm can set p to *lower* so that the decision-procedure query at line 5 becomes $\texttt{Model}(\varphi \wedge \neg\widehat{\gamma}(\mathit{lower}))$—i.e., we force the algorithm to perform the basic iteration-step from the RSY algorithm. In this way, we force *lower* to be updated at least once every N iterations. Moreover, if on such an RSY-step the model S returned from the decision procedure is None, then we know that *lower* has converged to $\widehat{\alpha}(\varphi)$ and the algorithm can return. A version of Algorithm 11 that implements this strategy is presented as Algorithm 13.

Theorem 5.4. *If abstract domain $\mathcal{A}$ does not have infinite ascending chains, and Algorithm 13 does not timeout, then $\widetilde{\alpha}_{+}^{\updownarrow}\langle\mathcal{L},\mathcal{A}\rangle(\varphi)$ terminates and returns $\widehat{\alpha}(\varphi)$.*

Proof. The proof of Theorem 5.2 carries over, except that we must additionally argue that if $\mathcal{A}$ has infinite descending chains, $\widetilde{\alpha}_{+}^{\updownarrow}$ (Algorithm 13) does not get trapped refining *upper* along an infinite descending chain, and that the algorithm returns $\widehat{\alpha}(\varphi)$ after *lower* has converged to $\widehat{\alpha}(\varphi)$.

Suppose that N consecutive iterations of the loop on lines 3–11 update *upper* (line 13) *without updating lower* (line 16). In the next iteration, the algorithm can set p to *lower* (line 8 so that the decision-procedure query at line 9 becomes $\texttt{Model}(\varphi \wedge \neg\widehat{\gamma}(\mathit{lower}))$—i.e., we force the algorithm to perform the basic iteration-step from the RSY algorithm. In this way, we force *lower* to be updated at least once every N iterations.

Consequently, because $\mathcal{A}$ has no infinite ascending chains, *lower* must eventually be set to $\widehat{\alpha}(\varphi)$. After that happens, within the next N iterations of the loop body, Algorithm 13 must execute line 8, which sets p to *lower* (i.e., $p = \widehat{\alpha}(\varphi)$). The model S returned from calling $\texttt{Model}(\varphi \wedge \neg\widehat{\gamma}(p))$ in line 9 must be `None`. As argued in the proof of Theorem 5.2, $\mathit{lower} \sqsubseteq \widehat{\alpha}(\varphi) \sqsubseteq \mathit{upper}$ holds on every iteration of the loop in lines 4–17. Because $\widehat{\alpha}(\varphi) \sqsubseteq \mathit{upper}$ and $p = \widehat{\alpha}(\varphi)$, the update $\mathit{upper} \leftarrow \mathit{upper} \sqcap p$ on line 13 assigns $\widehat{\alpha}(\varphi)$ to *upper*. Because *lower* is equal to *upper*, the loop exits with $\mathit{upper} = \widehat{\alpha}(\varphi)$. At line 18, *ans* is thus assigned $\widehat{\alpha}(\varphi)$, which is the value returned by Algorithm 13 at line 19. □

5.3.1 Algorithm Complexity

In this section, I analyze the worst-case time complexity of Algorithm 13. I first state the time complexity of the algorithm for a generic abstract domain $\mathcal{A}$ and logic $\mathcal{L}$. I then talk about the time complexity for the affine-equalities abstract domain $\mathcal{E}_{2^w}$ and QFBV logic.

The worst-case time complexity T of Algorithm 13 for abstract domain $\mathcal{A}$ and logic $\mathcal{L}$ is

$$T = \mathcal{O}(C \times (T_{AC} + T_{DP} + T_{AO})) \tag{5.3}$$

where

- C is the maximum length of an ascending chain in $\mathcal{A}$, which is an upper bound for the number of times the loop in lines 4–17 executes,
- T_{AC} is the time complexity of computing the abstract consequence on line 6,
- T_{DP} is the time complexity of the call to the decision procedure for logic $\mathcal{L}$ on line 9,
- T_{AO} is the time complexity of the abstract operations, meet, on line 13, and join, on line 16).

Consider the instantiation of Algorithm 13 with the affine-equalities abstract domain $\mathcal{E}_{2^w}$ and QFBV logic. Let k be the number of variables that the abstract domain $\mathcal{E}_{2^w}$ keeps track of. I now list the various parameters in Equation (5.3) for this specific instantiation of the bilateral algorithm:

- The length of the longest ascending chain in $\mathcal{E}_{2^w}$ is $\mathcal{O}(wk)$ (Elder et al., 2014, Theorem 2.7).
- The abstract consequence is computed using Algorithm 12. The loop on lines 3–5 can execute at most k times. The time complexity of the subsumption check on line 5 is $\mathcal{O}(k^3)$ (Elder et al., 2014).Thus, the worst-case time complexity for computing the abstract consequence is $T_{AC} = \mathcal{O}(k^4)$.
- The decision procedure for QFBV logic is NEXPTIME-complete (Kovásznai et al., 2012).
- The worst-case time complexity of the meet and join operations for $\mathcal{E}_{2^w}$ are $\mathcal{O}(k^3)$ (Elder et al., 2014).

Note that, technically, the time complexity of subsumption, meet, and join for $\mathcal{E}_{2^w}$ depends on an operation that has the same asymptotic complexity as matrix multiplication (Storjohann, 2000). The operation has a straightforward $\mathcal{O}(k^3)$ implementation that is similar to a standard Gaussian-elimination algorithm (Elder et al., 2014, Algorithm 1).

5.4 Instantiations

In this section, we describe instantiations of the bilateral framework for several abstract domains.

5.4.1 Herbrand-Equalities Domain

Herbrand equalities are used in analyses for partial redundancy elimination, loop-invariant code motion (Steffen et al., 1990), and strength reduction (Steffen et al., 1991). In these analyses, arithmetic operations (e.g., + and *) are treated as term constructors. Two program variables are known to hold equal values if the analyzer determines that the variables hold equal terms. Herbrand equalities can also be used to analyze programs whose types are user-defined algebraic data-types.

Basic definitions. Let $\mathcal{F}$ be a set of function symbols. The function $arity\colon \mathcal{F} \to \mathbb{N}$ yields the number of parameters of each function symbol. *Terms* over $\mathcal{F}$ are defined in the usual way; each function symbol $f \in \mathcal{F}$ always requires $arity(f)$ parameters. Let $\mathcal{T}(\mathcal{F}, X)$ denote the set of finite terms generated by $\mathcal{F}$ and variable set X. The *Herbrand universe* of $\mathcal{F}$ is $\mathcal{T}(\mathcal{F}, \emptyset)$, the set of *ground terms* over $\mathcal{F}$.

A *Herbrand state* is a mapping from program variables $\mathcal{V}$ to ground terms (i.e., a function in $\mathcal{V} \to \mathcal{T}(\mathcal{F}, \emptyset)$). The concrete domain consists of all sets of Herbrand states: $\mathcal{C} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{V} \to \mathcal{T}(\mathcal{F}, \emptyset))$. We can apply a Herbrand state σ to a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ as follows:

$$\sigma[t] \stackrel{\text{def}}{=} \begin{cases} \sigma(t) & \text{if } t \in \mathcal{V} \\ f(\sigma[t_1], \ldots, \sigma[t_k]) & \text{if } t = f(t_1, \ldots, t_k) \end{cases}$$

The Herbrand-equalities domain. Sets of Herbrand states can be abstracted in several ways. One way is to use conjunctions of equations among terms (whence the name "Herbrand-equalities domain"). Such systems of equations can be represented using Equivalence DAGs (Steffen et al., 1990). A different, but equivalent, approach is to use a representation based on *idempotent substitutions*: $\mathcal{A} = (\mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V}))_\perp$. Idempotence means that for each $\sigma \neq \perp$ and $v \in \mathcal{V}$, $\sigma[\sigma(v)] = \sigma(v)$. The meaning of an idempotent substitution $\sigma \in \mathcal{A}$ is given by its concretization

$\gamma(\sigma)$, where $\gamma\colon \mathcal{A} \to \mathcal{C}$, $\gamma(\bot) = \emptyset$, and otherwise

$$\gamma(\sigma) = \{\rho\colon \mathcal{V} \to \mathcal{T}(\mathcal{F}, \emptyset) \mid \forall v \in \mathcal{V}\colon \rho(v) = \rho[\sigma(v)]\}\,. \tag{5.4}$$

I now show that the Herbrand-equalities domain satisfies the requirements of the bilateral framework. I will assume that the logical language $\mathcal{L}$ has all the function symbols and constant symbols from $\mathcal{F}$, equality, and a constant symbol for each element from $\mathcal{V}$[1]. (In a minor abuse of notation, the set of such constant symbols will also be denoted by $\mathcal{V}$.) The logic's universe is the Herbrand universe of $\mathcal{F}$ (i.e., $\mathcal{T}(\mathcal{F}, \emptyset)$). An interpretation maps the constants in $\mathcal{V}$ to terms in $\mathcal{T}(\mathcal{F}, \emptyset)$. To be able to express $\widehat{\gamma}(p)$ and $\neg\widehat{\gamma}(p)$ (see item 3 below), we assume that $\mathcal{L}$ contains at least the following productions:

$$\begin{aligned} F &::= F \wedge F \mid \neg F \mid v = T \text{ for } v \in \mathcal{V} \mid \text{ false} \\ T &::= v \in \mathcal{V} \mid f(T_1, \ldots, T_k) \text{ when } \mathit{arity}(f) = k \end{aligned} \tag{5.5}$$

1. There is a Galois connection $\mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$:
   - The ordering on $\mathcal{C}$ is the subset relation on sets of Herbrand states.
   - $\gamma(\sigma)$ is given in Equation (5.4).
   - $\alpha(S) = \bigsqcap\{a \mid \gamma(a) \supseteq S\}$.
   - For $a, b \in \mathcal{A}$, $a \sqsubseteq b$ iff $\gamma(a) \subseteq \gamma(b)$.
2. Meet is most-general unification of substitutions, computed by standard unification techniques (Lassez et al., 1988, Thm. 3.1). Join is most-specific generalization, computed by "dual unification" or "anti-unification" (Plotkin, 1970; Reynolds, 1970), (Lassez et al., 1988, Thm. 5.8). Equality checking is described by Lassez et al. (Lassez et al., 1988, Prop. 4.10).
3. $\widehat{\gamma}$: $\widehat{\gamma}(\bot) =$ false; otherwise, $\widehat{\gamma}(\sigma)$ is $\bigwedge_{v \in \mathcal{V}} v = \sigma(v)$.

[1] Because the set of program variables is finite, $\mathcal{V}$ is finite.

4. A decision procedure for $\mathcal{L}$, with models, is given by Lassez et al. (Lassez et al., 1988, Thm. 6.12). In practice, one can obtain a decision procedure for $\mathcal{L}$ formulas using the built-in datatype mechanism of, e.g., Z3 (de Moura and Bjørner, 2008) or Yices (Dutertre and de Moura, 2006), and obtain the necessary decision procedure using an existing SMT solver.

5. $\mathcal{L}$ is closed under conjunction and negation.

6. `AbstractConsequence`: The domain is a conjunctive domain, as can be seen from the definition of $\widehat{\gamma}$.

Theorem 5.2 ensures that Algorithm 11 returns $\widehat{\alpha}(\varphi)$ when abstract domain $\mathcal{A}$ has neither infinite ascending nor infinite descending chains. The Herbrand-equalities domain has no infinite ascending chains (Lassez et al., 1988, Lem. 3.15). The domain described here also has no infinite descending chains, essentially because every right-hand term in every Herbrand state has no variables but those in $\mathcal{V}$.[2]

I now present a pair of worked examples of $\widetilde{\alpha}^{\updownarrow}$ (Algorithm 11) for the Herbrand-equalities domain.

Example 5.5. Consider the following code fragment, which uses two different methods for checking the parity of `i` (`i&1` and `i << 31 == 0`):

```
int i;
cons_tree x, y;
x = (i&1) ? nil : cons (y,x);
if ( i << 31 == 0 ) { /* (*) ... */ }
```

Suppose that we want an element of the Herbrand-equalities domain that relates the values of x and y at the fragment's start to x' and y', the values of those variables at program point `(*)`. A straightforward abstract interpretation of this program in the Herbrand-equalities domain would yield no information about x', because $\{x' \mapsto \texttt{nil}\} \sqcup \{x' \mapsto \text{cons}(y, x)\} = \{x' \mapsto x'\}$.

[2]One can instead define a domain of Herbrand equalities in which fresh variables may occur in the terms of an idempotent substitution's range. Everything said in this section remains true of this alternative domain, except that it has infinite descending chains, and so Algorithm 11 is only guaranteed to return an over-approximation of $\widehat{\alpha}(\varphi)$. However, as discussed in Section 5.2, Algorithm 13, presented in Section 5.3, is a version of Algorithm 11 that converges to $\widehat{\alpha}(\varphi)$, even when the abstract domain has infinite descending chains.

lower	*upper*	$\widehat{\gamma}(p)$	model, or unsatisfiable
$\bot$	$\top$	false	$i \mapsto 0$ $x \mapsto \texttt{nil}$ $x' \mapsto \text{cons}(\texttt{nil}, \texttt{nil})$ $y \mapsto \texttt{nil}$ $y' \mapsto \texttt{nil}$
$x \mapsto \texttt{nil}$ $x' \mapsto \text{cons}(\texttt{nil}, \texttt{nil})$ $y \mapsto \texttt{nil}$ $y' \mapsto \texttt{nil}$	$\top$	$y = \texttt{nil}$	$i \mapsto 0$ $x \mapsto \texttt{nil}$ $x' \mapsto \text{cons}(\text{cons}(\texttt{nil}, \texttt{nil}), \texttt{nil})$ $y \mapsto \text{cons}(\texttt{nil}, \texttt{nil})$ $y' \mapsto \text{cons}(\texttt{nil}, \texttt{nil})$
$x \mapsto \texttt{nil}$ $x' \mapsto \text{cons}(y, \texttt{nil})$ $y \mapsto y'$	$\top$	$y = y'$	unsatisfiable
$x \mapsto \texttt{nil}$ $x' \mapsto \text{cons}(y, \texttt{nil})$ $y \mapsto y'$	$y \mapsto y'$	$x = \texttt{nil}$	$i \mapsto 0$ $x \mapsto \text{cons}(\texttt{nil}, \texttt{nil})$ $x' \mapsto \text{cons}(\texttt{nil}, \text{cons}(\texttt{nil}, \texttt{nil}))$ $y \mapsto \texttt{nil}$ $y' \mapsto \texttt{nil}$
$x' \mapsto \text{cons}(y, x)$ $y \mapsto y'$	$y \mapsto y'$	$x' = \text{cons}(y, x)$	unsatisfiable
$x' \mapsto \text{cons}(y, x)$ $y \mapsto y'$	$x' \mapsto \text{cons}(y, x)$ $y \mapsto y'$		

Figure 5.3: Iterations of Algorithm 11 in Example 5.5. Self-mappings, e.g., $y \mapsto y$, are omitted.

Algorithm 11 can do better because, in essence, it accounts for the correlation between the conditions "i&1" and "i << 31 == 0". First, symbolic execution from the beginning of the code fragment to `(*)` yields the formula

$$\varphi \stackrel{\text{def}}{=} (x' = \mathit{ite}(i\&1, \texttt{nil}, \text{cons}(y, x))) \wedge (y' = y) \wedge (i \ll 31) = 0),$$

where $\mathit{ite}(\cdot, \cdot, \cdot)$ denotes the if-the-else operator. The values obtained just after line 5 during each iteration of Algorithm 11 are shown in Figure 5.3. Each row of Figure 5.3 displays, for a given iteration of Algorithm 11,

- the values of *lower* and *upper*,
- the value of $\widehat{\gamma}(p)$ computed from `AbstractConsequence`(*lower*, *upper*), and
- the model, if any, of $\varphi \wedge \neg\widehat{\gamma}(p)$ that Z3 returned.

lower	*upper*	$\widehat{\gamma}(p)$	model, or unsatisfiable	
$\bot$	$\top$	false	$a \mapsto \textbf{true}$ $x \mapsto \text{cons}(\texttt{nil}, \texttt{nil})$ $y \mapsto \texttt{nil}$	$x' \mapsto \texttt{nil}$ $y' \mapsto \text{cons}(\texttt{nil}, \texttt{nil})$
$x \mapsto \text{cons}(\texttt{nil}, \texttt{nil})$ $x' \mapsto \texttt{nil}$ $y \mapsto \texttt{nil}$ $y' \mapsto \text{cons}(\texttt{nil}, \texttt{nil})$	$\top$	$y = \texttt{nil}$	$a \mapsto \textbf{true}$ $x \mapsto \texttt{nil}$ $y \mapsto \text{cons}(\texttt{nil}, \texttt{nil})$	$x' \mapsto \text{cons}(\texttt{nil}, \texttt{nil})$ $y' \mapsto \texttt{nil}$
$x' \mapsto y$ $x \mapsto y'$	$\top$	$x' = y$	unsatisfiable	
$x' \mapsto y$ $x \mapsto y'$	$x' \mapsto y$	$x = y'$	unsatisfiable	
$x' \mapsto y$ $x \mapsto y'$	$x' \mapsto y$ $x \mapsto y'$			

Figure 5.4: Iterations of Algorithm 11 in Example 5.6. Self-mappings, e.g., $y \mapsto y$, are omitted.

Iterations that find a model increase the next iteration's *lower*, when *lower* $\leftarrow$ *lower* $\sqcup\, \beta(S)$. Iterations that return None decrease the next iteration's *upper*, when *upper* $\leftarrow$ *upper* $\sqcap\, p$. Each of the models shown is the model that Z3 returns when asked to satisfy $\varphi \wedge \neg\widehat{\gamma}(p)$, where $p = \texttt{AbstractConsequence}(\textit{lower}, \textit{upper})$. For example, each call to Z3 finished in 14 milliseconds or less. The final result is $\{x \mapsto x, y \mapsto y, x' \mapsto \text{cons}(y, x), y' \mapsto y\}$. ■

The next example shows that one can use the Herbrand-equalities domain without having to give *every* function symbol its Herbrand interpretation. This approach allows one to more faithfully model the language semantics—and still use the Herbrand-equalities domain—thereby increasing the set of equalities that the analysis is capable of detecting.

Example 5.6. Consider the following program fragment, which is in the same programming language as the fragment from Example 5.5, extended with selectors `car` and `cdr`:

```
bool a;
cons_tree x, y;
x = (a ? cons(y,x) : cons(x,y));
y = cdr(x);
x = car(x);
if (a) { /* (**) ... */ }
```

Suppose that we would like an element of the Herbrand-equalities domain that relates the values of x and y at the fragment's start to x' and y' at program point `(**)`. To reach `(**)`, `a` must be true; in this case, the code swaps the values of `x` and `y`.

As in Example 5.5, a straightforward abstract interpretation of the path to `(**)` in the Herbrand-equalities domain yields $\top$. As shown in Figure 5.4, symbolic abstraction, using the Herbrand-equalities domain, of the formula obtained from symbolic execution results in an abstract value that captures the swap effect.

In this example, the set of function symbols $\mathcal{F}$ over which we define the Herbrand universe $\mathcal{T}(\mathcal{F}, \emptyset)$ is $\mathcal{F} \stackrel{\text{def}}{=} \{\texttt{nil}, \text{cons}\}$, not $\mathcal{F} \stackrel{\text{def}}{=} \{\texttt{nil}, \text{cons}, \text{car}, \text{cdr}\}$. The functions car and cdr are not given their Herbrand interpretation; instead, they are interpreted as the deconstructors that select the first and second components, respectively, of a cons-term.

Symbolic execution from the beginning of the code fragment to `(**)` yields the following formula:

$$\begin{aligned}\varphi \stackrel{\text{def}}{=} & \; x' = \text{car}\,(ite\,(a, \text{cons}(y, x), \text{cons}(x, y))) \\ & \wedge\; y' = \text{cdr}\,(ite\,(a, \text{cons}(y, x), \text{cons}(x, y))) \\ & \wedge\; a\end{aligned}$$

As in Example 5.5, Figure 5.4 shows the values obtained just after line 5 on each iteration of Algorithm 11, applied to φ. Each call to Z3 finished in 10 milliseconds or less The final result is $x' = y$ and $x = y'$, which captures the fact that, when the program reaches `(**)`, it has swapped the values of `x` and `y`. ■

5.4.2 Polyhedral Domain

The polyhedral domain is a conjunctive domain (Section 3.1.1). In addition, there are algorithms for join, meet, and checking equality. The logic QF_LRA (quantifier-free linear real arithmetic) supported by SMT solvers provides a decision procedure for the fragment of logic that is required to express negation, conjunction, and $\widehat{\gamma}$ of a polyhedron. Consequently, the polyhedral domain

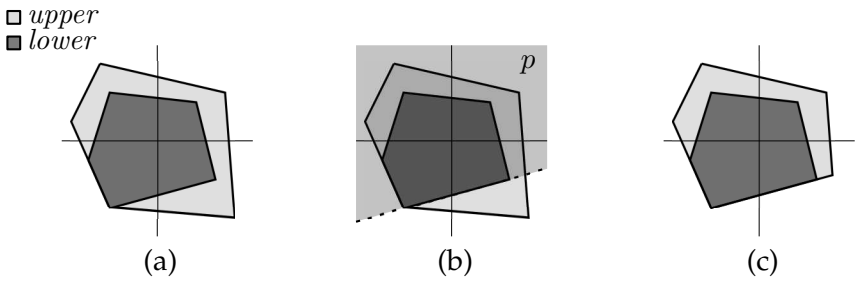


Figure 5.5: Abstract consequence on polyhedra. (a) Two polyhedra: *lower* $\sqsubseteq$ *upper*. (b) $p =$ `AbstractConsequence`$(lower, upper)$. (c) Result of *upper* $\leftarrow$ *upper* $\sqcap\, p$.

satisfies the bilateral framework, and therefore supports the $\widetilde{\alpha}^{\updownarrow}$ algorithm. The polyhedral domain has both infinite ascending chains and infinite descending chains, and hence Algorithm 13 is only guaranteed to compute an over-approximation of $\widehat{\alpha}(\varphi)$.

Because the polyhedral domain is a conjunctive domain, if *lower* $\sqsubsetneq$ *upper*, then some single constraint p of *lower* satisfies $p \not\sqsupseteq$ *upper*. For instance, if *lower* and *upper* are the polyhedra shown in Figure 5.5(a), then the region p above the dotted line in Figure 5.5(b) is an acceptable abstract consequence. Figure 5.5(c) shows the result after *upper* $\leftarrow$ *upper* $\sqcap\, p$ is performed at line 9 of Algorithm 11.

5.5 Empirical Comparison of the KS and Bilateral Algorithms

In this section, I compare two algorithms for performing symbolic abstraction for the affine-equalities domain $\mathcal{E}_{2^{32}}$:

- the $\widetilde{\alpha}^{\uparrow}_{\text{KS}}$ procedure of Algorithm 7.
- the $\widetilde{\alpha}^{\updownarrow}\langle\mathcal{E}_{2^{32}}\rangle$ procedure that is the instantiation of Algorithm 11 for the domain of affine-equalities $\mathcal{E}_{2^{32}}$ and QFBV logic.

Although the bilateral algorithm $\widetilde{\alpha}^{\updownarrow}\langle\mathcal{E}_{2^{32}}\rangle$ benefits from being resilient to timeouts, it maintains *both* an over-approximation and an under-approximation. Thus, the experiments were designed to understand the trade-off between performance and precision.

name	$\widetilde{\alpha}^{\uparrow}_{\mathrm{KS}}$	t/o	$\widetilde{\alpha}^{\updownarrow}\langle\mathcal{E}_{2^{32}}\rangle$	Better $\widetilde{\alpha}^{\updownarrow}$ precision
finger	104.0	4	138.9	**6.3%**
subst	196.7	4	214.6	0%
label	146.1	2	171.6	0%
chkdsk	377.2	16	417.9	0%
convert	287.1	10	310.5	0%
route	618.4	14	589.9	**2.5%**
logoff	611.2	16	644.6	**15.0%**
setup	1499	60	1576	**1.0%**

Table 5.2: Performance and precision comparison between the KS and Bilateral algorithms. The columns show the times, in seconds, for $\widetilde{\alpha}^{\uparrow}_{\mathrm{KS}}$ and $\widetilde{\alpha}^{\updownarrow}\langle\mathcal{E}_{2^{32}}\rangle$ WPDS construction; the number of invocations of $\widetilde{\alpha}^{\uparrow}_{\mathrm{KS}}$ that had a decision procedure timeout (t/o); and the degree of improvement gained by using $\widetilde{\alpha}^{\updownarrow}\langle\mathcal{E}_{2^{32}}\rangle$-generated weights rather than $\widetilde{\alpha}^{\uparrow}_{\mathrm{KS}}$ weights (measured as the percentage of control points whose inferred one-vocabulary affine relation was strictly more precise under $\widetilde{\alpha}^{\updownarrow}\langle\mathcal{E}_{2^{32}}\rangle$-based analysis).

In particular, the experiments were designed to answer the following questions:

1. How does the speed of $\widetilde{\alpha}^{\updownarrow}\langle\mathcal{E}_{2^{32}}\rangle$ compare with that of $\widetilde{\alpha}^{\uparrow}_{\mathrm{KS}}$?

2. How does the precision of $\widetilde{\alpha}^{\updownarrow}\langle\mathcal{E}_{2^{32}}\rangle$ compare with that of $\widetilde{\alpha}^{\uparrow}_{\mathrm{KS}}$?

To address these questions, we performed affine-relations analysis (ARA) on x86 machine code, computing affine relations over the x86 registers. The experimental setup is the same as that in Section 4.3.

Columns 2 and 4 of Table 5.2 list the time taken, in seconds, for $\widetilde{\alpha}^{\uparrow}_{\mathrm{KS}}$ and $\widetilde{\alpha}^{\updownarrow}\langle\mathcal{E}_{2^{32}}\rangle$ WPDS construction. We observe that on average $\widetilde{\alpha}^{\updownarrow}\langle\mathcal{E}_{2^{32}}\rangle$ is about 10% slower than $\widetilde{\alpha}^{\uparrow}_{\mathrm{KS}}$ (computed as the geometric mean), which answers question 1.

To answer question 2 we compared the precision of the WPDS analysis when using $\widetilde{\alpha}^{\uparrow}_{\mathrm{KS}}$ with the precision obtained using $\widetilde{\alpha}^{\updownarrow}\langle\mathcal{E}_{2^{32}}\rangle$. In particular, we compare the affine-relation invariants computed by the $\widetilde{\alpha}^{\uparrow}_{\mathrm{KS}}$-based and $\widetilde{\alpha}^{\updownarrow}\langle\mathcal{E}_{2^{32}}\rangle$-based analyses for each *control point*—i.e., the beginning of a basic block that ends with a branch. The last column of Table 5.2 shows the percentage of control points for which the $\widetilde{\alpha}^{\updownarrow}\langle\mathcal{E}_{2^{32}}\rangle$-based analysis computed a strictly more precise affine

Algorithm 14: $\widehat{\mathrm{Post}}^{\uparrow}[\tau](v)$

```
1
2  lower′ ← ⊥
3  while true do
4
5    ⟨S, S′⟩ ← Model(γ̂(v) ∧ τ ∧ ¬γ̂(lower′))
6    if ⟨S, S′⟩ is TimeOut then
7      return ⊤
8    else if ⟨S, S′⟩ is None then
9      break                          // P̂ost[τ](v) = lower′
10   else                                 // S′ ⊭ γ̂(lower′)
11     lower′ ← lower′ ⊔ β(S′)
12 v′ ← lower′
13 return v′
```

Algorithm 15: $\widehat{\mathrm{Post}}^{\updownarrow}[\tau](v)$

```
1  upper′ ← ⊤
2  lower′ ← ⊥
3  while lower′ ≠ upper′ ∧ ResourcesLeft do
4    p′ ← AbstractConsequence(lower′, upper′)
//   p′ ⊒ lower′, p′ ⋣ upper′
5    ⟨S, S′⟩ ← Model(γ̂(v) ∧ τ ∧ ¬γ̂(p′))
6    if ⟨S, S′⟩ is TimeOut then
7      break
8    else if ⟨S, S′⟩ is None then
9      upper′ ← upper′ ⊓ p′                 // P̂ost[τ](v) ⊑ p′
10   else                                       // S′ ⊭ γ̂(p′)
11     lower′ ← lower′ ⊔ β(S′)
12 v′ ← upper′
13 return v′
```

relation. Column 3 of Table 5.2 lists the number invocations of $\widetilde{\alpha}^{\uparrow}_{\mathrm{KS}}$ that had a decision-procedure timeout, and hence returned $\top$. We see that the $\widetilde{\alpha}^{\updownarrow}\langle\mathcal{E}_{2^{32}}\rangle$-based analysis improves precision at up to 15% of control points, and, on average, the $\widetilde{\alpha}^{\updownarrow}\langle\mathcal{E}_{2^{32}}\rangle$-based analysis is more precise for 3.1% of the control points (computed as the arithmetic mean), which answers question 2.

5.6 Computing $\widehat{\mathrm{Post}}$

Algorithm 15 shows the Bilateral algorithm for computing $\widehat{\mathrm{Post}}$. In the same way Algorithm 6, which computes symbolic abstraction, was adapted to give Algorithm 8, which computes $\widehat{\mathrm{Post}}$, Algorithm 11 is adapted to give Algorithm 15. The differences between $\widehat{\mathrm{Post}}^{\uparrow}$ and $\widehat{\mathrm{Post}}^{\updownarrow}$ are highlighted in gray. Most concern the variables *upper*′ or p'. Algorithm 8 will be used in Chapter 7.

5.7 Related Work

5.7.1 Related Work on Symbolic Abstraction

Previous work on symbolic abstraction falls into three categories:

1. algorithms for specific domains (Regehr and Reid, 2004; McMillan, 2005; Brauer and King, 2010; Barrett and King, 2010; King and Søndergaard, 2010; Elder et al., 2011; Li et al., 2014)

2. algorithms for parameterized abstract domains (Graf and Saïdi, 1997; Yorsh et al., 2004; Sankaranarayanan et al., 2005; Monniaux, 2010)

3. abstract-domain frameworks (Reps et al., 2004).

What distinguishes category 3 from category 2 is that each of the results cited in category 2 applies to a specific *family* of abstract domains, defined by a *parameterized Galois connection* (e.g., with an abstraction function equipped with a readily identifiable parameter for controlling the abstraction). In contrast, the results in category 3 are defined by an *interface*; for any abstract domain that satisfies the requirements of the interface, one has a method for symbolic abstraction. The approach presented in this paper falls into category 3.

Algorithms for specific domains. Regehr and Reid (2004) present a method that constructs abstract transformers for machine instructions, for interval and bitwise abstract domains. Their method does not call a SAT solver, but instead uses the physical processor (or a simulator of a processor) as a black box. To compute the abstract post-state for an abstract value a, the approach recursively divides a until an abstract value is obtained whose concretization is a singleton set. The concrete semantics are then used to derive the post-state value. The results of each division are joined as the recursion unwinds to derive the abstract post-state value.

Brauer and King (2010) developed a method that works from below to derive abstract transformers for the interval domain. Their method is based on an approach due to Monniaux (2010) (see below), but they changed two aspects:

1. They express the concrete semantics with a Boolean formula (via "bit-blasting"), which allows a formula equivalent to $\forall x.\varphi$ to be obtained from φ (in CNF) by removing the x and $\neg x$ literals from all of the clauses of φ.

2. Whereas Monniaux's method performs abstraction and then quantifier elimination, Brauer and King's method performs quantifier elimination on the concrete specification, and then performs abstraction.

The abstract transformer derived from the Boolean formula that results is a guarded update: the guard is expressed as an element of the octagon domain (Miné, 2001); the update operation is expressed as an element of the abstract domain of rational affine equalities (Karr, 1976). The abstractions performed to create the guard and the update are optimal for their respective domains. The algorithm they use to create the abstract value for the update operation is essentially the King-Søndergaard algorithm for $\widehat{\alpha}$ (King and Søndergaard, 2010, Fig. 2), which works from below (see Algorithm 7). Brauer and King show that optimal evaluation of such transfer functions requires linear programming. They give an example that demonstrates that an octagon-closure operation on a combination of the guard's octagon and the update's affine equality is sub-optimal.

Barrett and King (2010) describe a method for generating range and set abstractions for bit-vectors that are constrained by Boolean formulas. For range analysis, the algorithm separately computes the minimum and maximum value of the range for an n-bit bit-vector using $2n$ calls to a SAT solver, with each SAT query determining a single bit of the output. The result is the best over-approximation of the value that an integer variable can take on (i.e., $\widehat{\alpha}$).

Li et al. (2014) developed a symbolic-abstraction method for LRA, called SYMBA. The scenario considered by Li et al. (2014) differs from that considered in this thesis. Given a formula φ in LRA logic and a *finite* set of objectives $\{t_1, t_2, \ldots, t_n\}$, where t_i is a linear-rational expression, SYMBA computes the lower and upper bounds $l_1, l_2, \ldots, l_n$ and $u_1, u_2, \ldots, u_n$ such that $\varphi \Rightarrow (\bigwedge_{1 \leq i \leq n} l_i \leq t_i \leq u_i)$. Similar to the Bilateral framework described in this chapter, the SYMBA algorithm maintains and under-approximation and over-approximation of the final answer.

McMillan (2005) presents an algorithm for performing symbolic abstraction for propositional logic and the abstract domain of propositional clauses of length up to k. The algorithm can be viewed as an instance of the RSY algorithm: a SAT solver is used to generate samples and a trie data structure is used to perform the join of abstract values. The specific application that the algorithm is used for is to compute don't-care conditions for logic synthesis.

Algorithms for parameterized abstract domains. Graf and Saïdi (1997) showed that decision procedures can be used to generate best abstract transformers for predicate-abstraction domains. Other work has investigated more efficient methods to generate approximate transformers that

are not best transformers, but approach the precision of best transformers (Ball et al., 2001; Clarke et al., 2004).

Yorsh et al. (2004) developed a method that works from above to perform $\widetilde{\alpha}(\varphi)$ for the kind of abstract domains used in shape analysis (i.e., "canonical abstraction" of logical structures (Sagiv et al., 2002)).

Template Constraint Matrices (TCMs) are a parametrized family of linear-inequality domains for expressing invariants in linear real arithmetic. Sankaranarayanan et al. (2005) gave a meet, join, and set of abstract transformers for all TCM domains. Monniaux (2010) gave an algorithm that finds the best transformer in a TCM domain across a straight-line block (assuming that concrete operations consist of piecewise linear functions), and good transformers across more complicated control flow. However, the algorithm uses quantifier elimination, and no polynomial-time elimination algorithm is known for piecewise-linear systems.

Abstract-domain frameworks. As discussed in Section 5.2, the bilateral framework reduces to the RSY framework when using a particular (trivial) implementation of `AbstractConsequence`. Unlike the RSY framework, to compute $\widetilde{\alpha}(\varphi)$ the bilateral framework does not impose the requirement that the abstract domain have no infinite ascending chains. As shown in part 1 of Theorem 5.2, even when there are infinite ascending chains, the bilateral framework can return a *non-trivial* over-approximation of $\widehat{\alpha}(\varphi)$. In contrast, RSY gives no such guarantees. Consequently, compared to the RSY framework, the bilateral framework is applicable to a larger class of abstract domains.

5.7.2 Other Related Work

Logical abstract domains. Cousot et al. (2011a) define a method of abstract interpretation based on using particular sets of logical formulas as abstract-domain elements (so-called *logical abstract domains*). They face the problems of (i) performing abstraction from unrestricted formulas to the elements of a logical abstract domain (Cousot et al., 2011a, §7.1), and (ii) creating abstract transformers that transform input elements of a logical abstract domain to output elements of

the domain (Cousot et al., 2011a, §7.2). Their problems are particular cases of $\widehat{\alpha}(\varphi)$. They present heuristic methods for creating over-approximations of $\widehat{\alpha}(\varphi)$.

Connections to machine-learning algorithms.

$\widetilde{\alpha}^{\uparrow}_{\text{RSY}}$ (Algorithm 6) is related to the Find-S algorithm (Mitchell, 1997, Section 2.4) for concept learning. Both algorithms start with the most-specific hypothesis (i.e., $\bot$) and work bottom-up to find the most-specific hypothesis that is consistent with positive examples of the concept. Both algorithms generalize their current hypothesis each time they process a (positive) training example that is not explained by the current hypothesis. A major difference is that Find-S receives a sequence of positive and negative examples of the concept (e.g., from nature). It discards negative examples, and its generalization steps are based solely on the positive examples. In contrast, $\widetilde{\alpha}^{\uparrow}_{\text{RSY}}$ already starts with a precise statement of the concept in hand, namely, the formula φ, and on each iteration, calls a decision procedure to generate the next positive example; $\widetilde{\alpha}^{\uparrow}_{\text{RSY}}$ never sees a negative example.

A similar connection exists between $\widetilde{\alpha}^{\updownarrow}$ (Algorithm 11) and the Candidate-Elimination (CE) algorithm for concept learning (Mitchell, 1997, Section 2.5). Both algorithms maintain two approximations of the concept, one that is an over-approximation and one that is an under-approximation. The CE algorithm updates its under-approximation using positive examples in the same way that the Find-S algorithm updates its under-approximation. Similarly, the Bilateral algorithm updates its under-approximation (via a join) is the same way that the RSY algorithm updates its under-approximation. One key difference between the CE algorithm and the Bilateral algorithm is that the CE algorithm updates its over-approximation using *negative* examples. Most conjunctive abstract domains are not closed under negation. Thus, given a negative example, there usually does not exist an abstract value that only excludes that particular negative example.

5.8 Chapter Notes

The idea for the KS^{+} algorithm occurred to me during Andy King's talk at a Dagstuhl seminar describing the results of (King and Søndergaard, 2010). Though I was familiar with the paper,

I had never internalized the working of KS algorithm properly. The generalization of the KS$^+$ algorithm to the Bilateral framework followed naturally from the requirement that each iteration of the Bilateral algorithm is required to make progress.

The concept of Abstract Consequence was concealed inside the KS algorithm, and had to be gleaned out. The concept of abstract consequence was used implicitly, and an acceptable abstract consequence was computed by reading a single row of a matrix in Howell form (line 5 of Algorithm 9).

The main surprise was that the Howell form, a normal form using for the $\mathcal{E}_{2^w}$ abstract domain, stressed in Elder et al. (2011) was not of central importance in generalizing the KS algorithm; instead the Abstract Consequence operation was the key.

Chapter 6

A Generalization of Stålmarck's Method

> In building a statue, a sculptor doesn't keep adding clay to his subject. Actually, he keeps chiselling away at the inessentials until the truth of its creation is revealed without obstructions.
> — Bruce Lee

As explained in Section 3.4, Stålmarck's method (Sheeran and Stålmarck, 2000) is an algorithm for checking satisfiability of a formula in propositional logic. In this chapter, I give a new account of Stålmarck's method by explaining each of the key components in terms of concepts from the field of abstract interpretation. In particular, I show that Stålmarck's method can be viewed as a general framework, which I call Stålmarck[$\mathcal{A}$], that is parameterized on an abstract domain $\mathcal{A}$ and operations on $\mathcal{A}$. This abstraction-based view allows Stålmarck's method to be lifted from propositional logic to *richer logics*, such as LRA. Furthermore, the generalized Stålmarck's method falls into the Satisfiability Modulo Abstraction (SMA) paradigm: the generalized Stålmarck's-method solver we created is designed and implemented using concepts from abstract interpretation.

In this chapter, I use the vantage point of abstract interpretation to describe the elements of the Dilemma Rule as follows:

Branch of a Proof: In Stålmarck's method, each proof-tree branch is associated with a so-called *formula relation* (Sheeran and Stålmarck, 2000). In abstract-interpretation terms, each branch is associated with an abstract-domain element.

Splitting: The step of splitting the current goal into sub-goals can be expressed in terms of *meet* ($\sqcap$).

Application of simple deductive rules: Stålmarck's method applies a set of simple deductive rules after each split. In abstract-interpretation terms, the rules perform a *semantic reduction* (Cousot and Cousot, 1979).

"Intersecting" results: The step of combining the results obtained from an earlier split are described as an "intersection" in Stålmarck's papers. In the abstract-interpretation-based framework, the combining step is the *join* ($\sqcup$) of two abstract-domain values.

This more general view of Stålmarck's method furnishes insight on when an invocation of the Dilemma Rule fails to make progress in a proof. In particular, both branches of a Dilemma may each succeed (locally) in advancing the proof, but the abstract domain used to represent proof states may not be precise enough to represent the common information when the join of the two branches is performed; consequently, the global state of the proof is not advanced.

As mentioned in Section 1.2, the other advantage of the abstraction-based view of Stålmarck's method is that it brings out a connection between Stålmarck's method and symbolic abstraction; in particular, Stålmarck[$\mathcal{A}$] computes $\widehat{\alpha}_{\mathcal{A}}(\varphi)$. This new algorithm for symbolic abstraction approaches its result from "above", and is denoted by $\widetilde{\alpha}^{\downarrow}$. Because the method approaches its result from "above", if the computation takes too much time, it can be stopped to yield a safe result—i.e., an over-approximation to the best abstract operation—at any stage, similar to the Bilateral framework (Chapter 5).

The $\widetilde{\alpha}^{\downarrow}$ framework, however, is based on much different principles from the RSY and Bilateral frameworks. The latter frameworks use an *inductive-learning approach* to learn from examples, while the $\widetilde{\alpha}^{\downarrow}$ framework uses a *deductive approach* by using inference rules to deduce the answer. Thus, they represent two different classes of frameworks, with different requirements for the abstract domain. In contrast to the RSY and Bilateral framework, which uses a decision procedure as a black box, the $\widetilde{\alpha}^{\downarrow}$ framework adopts (and adapts) some principles from Stålmarck's decision procedure.

The contributions of this chapter can be summarized as follows:

- I present a connection between symbolic abstraction and Stålmarck's method for checking satisfiability (Section 6.1).
- I present a generalization of Stålmarck's method that lifts the algorithm from propositional logic to richer logics (Section 6.2).
- I present a new parametric framework that, for some abstract domains, is capable of performing most-precise abstract operations in the limit, including $\widehat{\alpha}(\varphi)$ and $\widehat{\mathrm{Assume}}[\varphi](A)$, as well as creating a representation of $\widehat{\mathrm{Post}}[\tau]$. Because the method approaches most-precise values from "above", if the computation takes too much time it can be stopped to yield a sound result (Section 6.2).
- I present instantiations of our framework for two logic/abstract-domain pairs: QFBV/$\mathcal{E}_{2^w}$ and LRA/Polyhedra, and discuss completeness (Section 6.3).
- I present experimental results that illustrate the dual-use nature of our framework. One experiment uses it to compute abstract transformers, which are then used to generate invariants; another experiment uses it for checking satisfiability (Section 6.4).

Section 6.5 presents related work.

6.1 Overview

I now illustrate the key points of the generalized Stålmarck's method using three examples:

- Section 6.1.1 illustrates how using inequality relations instead of equivalence relations in Stålmarck's method gives us a more powerful decision procedure for propositional logic.
- Section 6.1.2 shows how the generalized Stålmarck's method applies to computing representations of abstract transformers.
- Section 6.1.3 describes the application of the generalized Stålmarck's method to checking unsatisfiability of formulas in the LRA logic.

The top-level, overall goal of Stålmarck's method can be understood in terms of the operation $\widetilde{\alpha}(\psi)$. However, Stålmarck's method is recursive (counting down on a parameter k), and the operation performed at each recursive level is the slightly more general operation $\widetilde{\text{Assume}}[\psi](A)$. Thus, we will discuss $\widetilde{\text{Assume}}$ in Sections 6.1.2 and 6.1.3.

6.1.1 An Improvement of Stålmarck's Method Replacing Equivalence Relations with Inequality Relations for Propositional Logic

Instead of computing an equivalence relation $\equiv$ on literals as done in Section 3.4, let us compute an inequality relation $\leq$ between literals. Figure 6.1 shows a few of the propagation rules that deduce inequalities. Because (i) an equivalence $a \equiv b$ can be represented using two inequality constraints, $a \leq b$ and $b \leq a$, (ii) an inequivalence $a \not\equiv b$ can be treated as an equivalence $a \equiv \neg b$, and (iii) $a \leq b$ cannot be represented with any number of equivalences, inequality relations are a strictly more expressive method than equivalence relations for abstracting a set of variable assignments. Moreover, Example 6.1 shows that, for some tautologies, replacing equivalence relations with inequality relations enables Stålmarck's method to be able to find a k-saturation proof with a strictly lower value of k.

$$\frac{a \Leftrightarrow (b \Rightarrow c)}{c \leq a \qquad \neg a \leq b}\ \textsc{Imp1} \qquad \frac{a \Leftrightarrow (b \Leftrightarrow c) \qquad a \leq \mathbf{0}}{b \leq \neg c \qquad \neg c \leq b}\ \textsc{Iff1} \qquad \frac{a \Leftrightarrow (b \Rightarrow c) \qquad \mathbf{1} \leq b \qquad c \leq \mathbf{0}}{a \leq 0}\ \textsc{Imp2}$$

Figure 6.1: Examples of propagation rules for inequality relations on literals.

$w_1 \leq \mathbf{0}$		...	by assumption
$w_2 \leq \neg w_3$,	$\neg w_3 \leq w_2$	...	Rule Iff1 on $w_1 \Leftrightarrow (w_2 \Leftrightarrow w_3)$
$q \leq w_2$,	$\neg w_2 \leq p$	...	Rule Imp1 on $w_2 \Leftrightarrow (p \Rightarrow q)$
$q \leq \neg w_3$		...	$q \leq w_2, w_2 \leq \neg w_3$
$w_3 \leq p$		...	$w_2 \leq \neg w_3$ implies $w_3 \leq \neg w_2, \neg w_2 \leq p$
$\neg p \leq w_3$,	$\neg w_3 \leq \neg q$	...	Rule Imp1 on $w_3 \Leftrightarrow (\neg q \Rightarrow \neg p)$
$q \leq \mathbf{0}$		...	$\neg w_3 \leq \neg q$ implies $q \leq w_3, q \leq \neg w_3$
$\mathbf{1} \leq p$		...	$w_3 \leq p, \neg p \leq w_3$ implies $\neg w_3 \leq p$
$w_2 \leq \mathbf{0}$,		...	Rule Imp2 on $w_2 \Leftrightarrow (p \Rightarrow q)$
$w_3 \leq \mathbf{0}$		...	Rule Imp2 on $w_3 \Leftrightarrow (\neg q \Rightarrow \neg p)$
$\mathbf{1} \leq \mathbf{0}$		...	$w_2 \leq \neg w_3, \neg w_3 \leq w_2, w_2 \leq \mathbf{0}, w_3 \leq \mathbf{0}$

Figure 6.2: 0-saturation proof that χ is valid, using inequality relations on literals.

Example 6.1. Consider the propositional-logic formula $\chi \equiv (p \Rightarrow q) \Leftrightarrow (\neg q \Rightarrow \neg p)$. The corresponding integrity constraints are $w_1 \Leftrightarrow (w_2 \Leftrightarrow w_3)$, $w_2 \Leftrightarrow (p \Rightarrow q)$, and $w_3 \Leftrightarrow (\neg q \Rightarrow \neg p)$. The root variable of χ is w_1. Using formula relations (i.e., equivalence relations over literals), Stålmarck's method finds a 1-saturation proof that χ is valid. In contrast, using inequality relations, a Stålmarck-like algorithm finds a 0-saturation proof. The proof starts by assuming that $w_1 \leq \mathbf{0}$. 0-saturation using the propagation rules of Figure 6.1 results in the contradiction $\mathbf{1} \leq \mathbf{0}$, as shown in Figure 6.2. ■

Because it can find a k-saturation proof with a strictly lower value of k than the standard Stålmarck's method, we say that the instantiation of Stålmarck's method with inequality relations is *more powerful than* the instantiation with equivalence relations. In general, Stålmarck's method can be made more and more powerful by using a more and more expressive abstraction: each time you plug in a successively more expressive abstraction, a proof may be possible with a lower value of k.

6.1.2 Computing Representations of Abstract Transformers

Example 6.2. Consider the following x86 assembly code

```
L1:  cmp eax, 2   L2:  jz L4   L3:  …  L4:  …
```

The instruction at `L1` sets the zero flag (`zf`) to true if the value of register `eax` equals 2. At instruction `L2`, if `zf` is true the program jumps to location `L4` by updating the value of the program counter (`pc`) to `L4`; otherwise, control falls through to program location `L3`. The transition formula that expresses the state transformation from the beginning of `L1` to the beginning of `L4` is thus

$$\varphi \equiv (\texttt{zf} \Leftrightarrow (\texttt{eax} = 2)) \wedge (\texttt{pc}' = \mathit{ITE}(\texttt{zf}, \texttt{L4}, \texttt{L3})) \wedge (\texttt{pc}' = \texttt{L4}) \wedge (\texttt{eax}' = \texttt{eax}),$$

where φ is a QFBV formula, and $\mathit{ITE}(b, t_1, t_2)$ is an *if-then-else* term, which evaluates to term t_1 if the Boolean condition b is true, and to term t_2 if b is false.

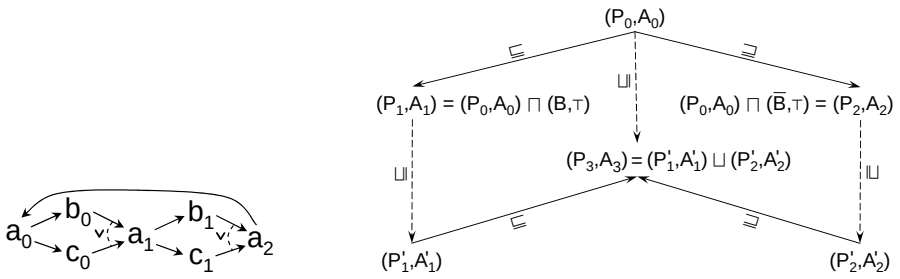
Let $\mathcal{E}_{2^{32}}$ be the abstract domain of affine-equalities over the 32-bit x86 registers. Let $A_0 = \top_{\mathcal{E}_{2^{32}}}$, the empty set of affine constraints over input-state and output-state variables. We now describe how our algorithm creates a representation of the $\mathcal{E}_{2^{32}}$ transformer for φ by computing $\widetilde{\mathrm{Assume}}[\varphi](A_0)$.

First, the ITE term in φ is rewritten as $(\texttt{zf} \Rightarrow (\texttt{pc}' = \texttt{L4})) \wedge (\neg\texttt{zf} \Rightarrow (\texttt{pc}' = \texttt{L3}))$. Thus, the transition formula becomes $\varphi = (\texttt{zf} \Leftrightarrow (\texttt{eax} = 2)) \wedge (\texttt{zf} \Rightarrow (\texttt{pc}' = \texttt{L4})) \wedge (\neg\texttt{zf} \Rightarrow (\texttt{pc}' = \texttt{L3})) \wedge (\texttt{pc}' = \texttt{L4}) \wedge (\texttt{eax}' = \texttt{eax})$.

Next, *propagation rules* are used to compute a semantic reduction with respect to φ, starting from A_0. The main feature of the propagation rules is that they are "local"; that is, they make use of only a small part of formula φ to compute the semantic reduction.

1. Because φ has to be true, we can conclude that each of the conjuncts of φ are also true; that is, $\texttt{zf} \Leftrightarrow (\texttt{eax} = 2)$, $\texttt{zf} \Rightarrow (\texttt{pc}' = \texttt{L4})$, $\neg\texttt{zf} \Rightarrow (\texttt{pc}' = \texttt{L3})$, $\texttt{pc}' = \texttt{L4}$, and $\texttt{eax}' = \texttt{eax}$ are all true.

2. Suppose that we have a function $\mu\widetilde{\alpha}_{\mathcal{E}_{2^{32}}}$ such that for a literal $l \in \mathcal{L}$, $A' = \mu\widetilde{\alpha}_{\mathcal{E}_{2^{32}}}(l)$ is a sound overapproximation of $\widehat{\alpha}(l)$. Because the literal $\texttt{pc}' = \texttt{L4}$ is true, we conclude that



(a) (b)

Figure 6.3: (a) Inconsistent inequalities in the (unsatisfiable) formula used in Example 6.3. (b) Application of the Dilemma Rule to abstract value (P_0, A_0). The dashed arrows from (P_i, A_i) to (P'_i, A'_i) indicate that (P'_i, A'_i) is a semantic reduction of (P_i, A_i).

$A' = \mu\widetilde{\alpha}_{\mathcal{E}_{232}}(\mathtt{pc}' = \mathtt{L4}) = \{\mathtt{pc}' - \mathtt{L4} = 0\}$ holds, and thus $A_1 = A_0 \sqcap A' = \{\mathtt{pc}' - \mathtt{L4} = 0\}$, which is a semantic reduction of A_0.

3. Similarly, because the literal $\mathtt{eax}' = \mathtt{eax}$ is true, we obtain $A_2 = A_1 \sqcap \mu\widetilde{\alpha}_{\mathcal{E}_{232}}(\mathtt{eax}' = \mathtt{eax}) = \{\mathtt{pc}' - \mathtt{L4} = 0, \mathtt{eax}' - \mathtt{eax} = 0\}$.

4. We know that $\neg\mathtt{zf} \Rightarrow (\mathtt{pc}' = \mathtt{L3})$. Furthermore, $\mu\widetilde{\alpha}_{\mathcal{E}_{232}}(\mathtt{pc}' = \mathtt{L3}) = \{\mathtt{pc}' - \mathtt{L3} = 0\}$. Now $\{\mathtt{pc}' - \mathtt{L3} = 0\} \sqcap A_2$ is $\bot$, which implies that $[\![\mathtt{pc}' = \mathtt{L3}]\!] \cap \gamma(\{\mathtt{pc}' - \mathtt{L4} = 0, \mathtt{eax}' - \mathtt{eax} = 0\}) = \emptyset$. Thus, we can conclude that $\neg\mathtt{zf}$ is false, and hence that $\mathtt{zf}$ is true. This value of $\mathtt{zf}$, along with the fact that $\mathtt{zf} \Leftrightarrow (\mathtt{eax} = 2)$ is true, enables us to determine that $A'' = \mu\widetilde{\alpha}_{\mathcal{E}_{232}}(\mathtt{eax} = 2) = \{\mathtt{eax} - 2 = 0\}$ must hold. Thus, our final semantic-reduction step produces $A_3 = A_2 \sqcap A'' = \{\mathtt{pc}' - \mathtt{L4} = 0, \mathtt{eax}' - \mathtt{eax} = 0, \mathtt{eax} - 2 = 0\}$.

Abstract value A_3 is a set of affine constraints over the registers at `L1` (input-state variables) and those at `L4` (output-state variables). ■

The above example illustrates how our technique propagates truth values to various subformulas of φ. The process of repeatedly applying propagation rules to compute $\widetilde{\text{Assume}}$ is called 0-assume. The next example illustrates the *Dilemma Rule*, a more powerful rule for computing semantic reductions.

6.1.3 Checking Unsatisfiability of LRA formulas

Example 6.3. Let $\mathcal{L}$ be LRA, and let $\mathcal{A}$ be the polyhedral abstract domain (Cousot and Halbwachs, 1978). Consider the formula

$$\psi \equiv (a_0 < b_0) \wedge (a_0 < c_0) \wedge (b_0 < a_1 \vee c_0 < a_1) \wedge (a_1 < b_1) \wedge (a_1 < c_1) \wedge (b_1 < a_2 \vee c_2 < a_2) \wedge (a_2 < a_0)$$

The structure of $\psi \in \mathcal{L}$ is illustrated in Figure 6.3(a). Suppose that we want to compute $\widetilde{\text{Assume}}[\psi](\top_{\mathcal{A}})$.

To make the communication between the truth values of subformulas and the abstract value explicit, we associate a fresh Boolean variable with each subformula of ψ to give a set of *integrity constraints* $\mathcal{I}$. In this case, $\mathcal{I}_\psi = \{u_1 \Leftrightarrow \bigwedge_{i=2}^{8} u_i, u_2 \Leftrightarrow (a_0 < b_0), u_3 \Leftrightarrow (a_0 < c_0), u_4 \Leftrightarrow (u_9 \vee u_{10}), u_5 \Leftrightarrow (a_1 < b_1), u_6 \Leftrightarrow (a_1 < c_1), u_7 \Leftrightarrow (u_{11} \vee u_{12}), u_8 \Leftrightarrow (a_2 < a_0), u_9 \Leftrightarrow (b_0 < a_1), u_{10} \Leftrightarrow (c_0 < a_1), u_{11} \Leftrightarrow (b_1 < a_2), u_{12} \Leftrightarrow (c_1 < a_2)\}$. The integrity constraints encode the structure of ψ via the set of Boolean variables $\mathcal{U} = \{u_1, u_2, \ldots, u_{12}\}$. When $\mathcal{I}$ is used as a formula, it denotes the conjunction of the individual integrity constraints.

We now introduce an abstraction over $\mathcal{U}$; in particular, we use the Cartesian domain $\mathcal{P} = (\mathcal{U} \to \{0, 1, *\})_\perp$ in which $*$ denotes "unknown", and each element in $\mathcal{P}$ represents a set of assignments in $\mathbb{P}(\mathcal{U} \to \{0, 1\})$. We denote an element of the Cartesian domain as a mapping, e.g., $[u_1 \mapsto 0, u_2 \mapsto 1, u_3 \mapsto *]$, or $[0, 1, *]$ if u_1, u_2, and u_3 are understood. $\top_{\mathcal{P}}$ is the element $\lambda u.*$. The "single-point" partial assignment in which variable v is set to b is denoted by $\top_{\mathcal{P}}[v \mapsto b]$.

The variable $u_1 \in \mathcal{U}$ represents the root of ψ; consequently, the single-point partial assignment $\top_{\mathcal{P}}[u_1 \mapsto 1]$ corresponds to the assertion that ψ is satisfiable. In fact, the models of ψ are closely related to the concrete values in $[\![\mathcal{I}]\!] \cap \gamma(\top_{\mathcal{P}}[u_1 \mapsto 1])$. For every concrete value in $[\![\mathcal{I}]\!] \cap \gamma(\top_{\mathcal{P}}[u_1 \mapsto 1])$, its projection onto $\{a_i, b_i, c_i \mid 0 \leq i \leq 1\} \cup \{a_2\}$ gives us a model of ψ; that is, $[\![\psi]\!] = ([\![\mathcal{I}]\!] \cap \gamma(\top_{\mathcal{P}}[u_1 \mapsto 1]))|_{(\{a_i, b_i, c_i | 0 \leq i \leq 1\} \cup \{a_2\})}$. By this means, the problem of computing $\widetilde{\text{Assume}}[\psi](\top_{\mathcal{A}})$ is reduced to that of computing $\widetilde{\text{Assume}}[\mathcal{I}]((\top_{\mathcal{P}}[u_1 \mapsto 1], \top_{\mathcal{A}}))$, where $(\top_{\mathcal{P}}[u_1 \mapsto 1], \top_{\mathcal{A}})$ is an element of the reduced product of $\mathcal{P}$ and $\mathcal{A}$.

Because u_1 is true in $\top_{\mathcal{P}}[u_1 \mapsto 1]$, the integrity constraint $u_1 \Leftrightarrow \bigwedge_{i=2}^{8} u_i$ implies that $u_2 \ldots u_8$ are also true, which refines $\top_{\mathcal{P}}[u_1 \mapsto 1]$ to $P_0 = [1, 1, 1, 1, 1, 1, 1, 1, *, *, *, *]$. Because u_2 is true and $u_2 \Leftrightarrow (a_0 < b_0) \in \mathcal{I}$, $\top_{\mathcal{A}}$ can be refined using $\mu\widetilde{\alpha}_{\mathcal{A}}(a_0 < b_0) = \{a_0 - b_0 < 0\}$. Doing the same for u_3, u_5, u_6, and u_8, refines $\top_{\mathcal{A}}$ to $A_0 = \{a_0 - b_0 < 0, a_0 - c_0 < 0, a_1 - b_1 < 0, a_1 - c_1 < 0, a_2 - a_0 < 0\}$. These steps refine $(\top_{\mathcal{P}}[u_1 \mapsto 1], \top_{\mathcal{A}})$ to (P_0, A_0) via 0-assume.

To increase precision, we need to use the Dilemma Rule, a branch-and-merge rule, in which the current abstract state is split into two (disjoint) abstract states, 0-assume is applied to both abstract values, and the resulting abstract values are merged by performing a join. The steps of the Dilemma Rule are shown schematically in Figure 6.3(b) and described below.

In our example, the value of u_9 is unknown in P_0. Let $B \in \mathcal{P}$ be $\top_{\mathcal{P}}[u_9 \mapsto 0]$; then $\overline{B}$, the abstract complement[1] of B, is $\top_{\mathcal{P}}[u_9 \mapsto 1]$. The current abstract value (P_0, A_0) is split into

$$(P_1, A_1) = (P_0, A_0) \sqcap (B, \top) \quad \text{and} \quad (P_2, A_2) = (P_0, A_0) \sqcap (\overline{B}, \top).$$

Now consider 0-assume on (P_1, A_1). Because u_9 is false, and u_4 is true, we can conclude that u_{10} has to be true, using the integrity constraint $u_4 \Leftrightarrow (u_9 \vee u_{10})$. Because u_{10} holds and $u_{10} \Leftrightarrow (c_0 < a_1) \in \mathcal{I}$, A_1 can be refined with the constraint $c_0 - a_1 < 0$. Because $a_0 - c_0 < 0 \in A_1$, $a_0 - a_1 < 0$ can be inferred. Similarly, when performing 0-assume on (P_2, A_2), $a_0 - a_1 < 0$ is inferred. Call the abstract values computed by 0-assume (P_1', A_1') and (P_2', A_2'), respectively. At this point, the join of (P_1', A_1') and (P_2', A_2') is taken. Because $a_0 - a_1 < 0$ is present in both branches, it is retained in the join. The resulting abstract value is $(P_3, A_3) = ([1, 1, 1, 1, 1, 1, 1, 1, *, *, *, *], \{a_0 - b_0 < 0, a_0 - c_0 < 0, a_1 - b_1 < 0, a_1 - c_1 < 0, a_2 - a_0 < 0, a_0 - a_1 < 0\}$. Note that although P_3 equals P_0, A_3 is lower in the lattice $\mathcal{A}$ than A_0 (i.e., $A_3 \sqsubset A_0$), and hence (P_3, A_3) is a semantic reduction of (P_0, A_0) with respect to ψ.

Now suppose that (P_3, A_3) is split using u_{11}. Using reasoning similar to that performed above, $a_1 - a_2 < 0$ is inferred on both branches, and hence so is $a_0 - a_2 < 0$. However, $a_0 - a_2 < 0$ contradicts $a_2 - a_0 < 0$; consequently, the abstract value reduces to $(\bot_{\mathcal{P}}, \bot_{\mathcal{A}})$ on both branches. Thus, $\widetilde{\text{Assume}}[\psi](\top_{\mathcal{A}}) = \bot_{\mathcal{A}}$, and hence ψ is unsatisfiable. In this way, $\widetilde{\text{Assume}}$ instantiated with the polyhedral domain can be used to decide the satisfiability of a LRA formula. ■

[1] $\overline{B}$ is an abstract complement of B if and only if $\gamma(B) \cap \gamma(\overline{B}) = \emptyset$, and $\gamma(B) \cup \gamma(\overline{B}) = \gamma(\top)$.

The process of repeatedly applying the Dilemma Rule is called 1-assume. That is, repeatedly some variable $u \in \mathcal{U}$ is selected whose truth value is unknown, the current abstract value is split using $B = \top_{\mathcal{P}}[u \mapsto 0]$ and $\overline{B} = \top_{\mathcal{P}}[u \mapsto 1]$, 0-assume is applied to each of these values, and the resulting abstract values are merged via join (Figure 6.3(b)). Different policies for selecting the next variable on which to split can affect how quickly an answer is found; however, any fair selection policy will return the same answer. The efficacy of the Dilemma Rule is partially due to case-splitting; however, the real power of the Dilemma Rule is due to the fact that it *preserves information learned in both branches when a case-split is "abandoned" at a join point*.

The generalization of the 1-assume algorithm is called k-assume: repeatedly some variable $u \in \mathcal{U}$ is selected whose truth value is unknown, the current abstract value is split using $B = \top_{\mathcal{P}}[u \mapsto 0]$ and $\overline{B} = \top_{\mathcal{P}}[u \mapsto 1]$; (k–1)-assume is applied to each of these values; and the resulting values are merged via join. However, there is a trade-off: higher values of k give greater precision, but are also computationally more expensive.

For certain abstract domains and logics, $\widetilde{\text{Assume}}[\psi](\top_{\mathcal{A}})$ is complete—i.e., with a high-enough value of k for k-assume, $\widetilde{\text{Assume}}[\psi](\top_{\mathcal{A}})$ always computes the most-precise $\mathcal{A}$ value possible for ψ. However, our experiments show that $\widetilde{\text{Assume}}[\psi](\top_{\mathcal{A}})$ has very good precision with $k = 1$ (see Section 6.4)—which jibes with the observation that, in practice, with Stålmarck's method for propositional validity (tautology) checking "a formula is either [provable with $k = 1$] or not a tautology at all!" (Harrison, 1996, p. 227).

6.2 Algorithm for $\widetilde{\text{Assume}}[\varphi](A)$

This section presents our algorithm for computing $\widetilde{\text{Assume}}[\varphi](A) \in \mathcal{A}$, for $\varphi \in \mathcal{L}$. The proofs of the various theorems can be skipped without losing continuity.

The assumptions of our framework are as follows:

1. There is a Galois connection $\mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$ between $\mathcal{A}$ and concrete domain $\mathcal{C}$.

2. There is an algorithm to perform the join of arbitrary elements of $\mathcal{A}$.

Algorithm 16: $\widetilde{\text{Assume}}[\varphi](A)$

```
1 ⟨I, u_φ⟩ ← integrity(φ)
2 P ← ⊤_P[u_φ ↦ 1]
3 (P̃, Ã) ← k-assume[I]((P, A))
4 return Ã
```

Algorithm 17: 0-assume$[\mathcal{I}]((P, A))$

```
1 repeat
2   (P′, A′) ← (P, A)
3   foreach J ∈ I do
4     if J has the form u ⇔ ℓ then
5       (P, A) ← LeafRule(J, (P, A))
6     else
7       (P, A) ← InternalRule(J, (P, A))
8 until ((P, A) = (P′, A′)) ∥ timeout
9 return (P, A)
```

Algorithm 18: k-assume$[\mathcal{I}]((P, A))$

```
1  repeat
2    (P′, A′) ← (P, A)
3    foreach u ∈ U such that P(u) = ∗ do
4      (P_0, A_0) ← (P, A)
5      (B, B̄) ← (⊤_P[u ↦ 0], ⊤_P[u ↦ 1])
6      (P_1, A_1) ← (P_0, A_0) ⊓ (B, ⊤)
7      (P_2, A_2) ← (P_0, A_0) ⊓ (B̄, ⊤)
8      (P′_1, A′_1) ← (k–1)-assume[I]((P_1, A_1))
9      (P′_2, A′_2) ← (k–1)-assume[I]((P_2, A_2))
10     (P, A) ← (P′_1, A′_1) ⊔ (P′_2, A′_2)
11 until ((P, A) = (P′, A′)) ∥ timeout
12 return (P, A)
```

3. Given a literal $l \in \mathcal{L}$, there is an algorithm $\mu\widetilde{\alpha}$ to compute a safe (overapproximating) "micro-$\widetilde{\alpha}$"—i.e., $A' = \mu\widetilde{\alpha}(l)$ such that $\gamma(A') \supseteq [\![l]\!]$.

4. There is an algorithm to perform the meet of an arbitrary element of $\mathcal{A}$ with an arbitrary element of $\{\mu\widetilde{\alpha}(l) \mid \ell \in \text{literal}(\mathcal{L})\}$.

Note that $\mathcal{A}$ is allowed to have infinite descending chains; because $\widetilde{\text{Assume}}$ works from above, it is allowed to stop at any time, and the value in hand is an over-approximation of the most precise answer.

Algorithm 16 presents the algorithm that computes $\widetilde{\text{Assume}}[\varphi](A)$ for $\varphi \in \mathcal{L}$ and $A \in \mathcal{A}$. Line 1 calls the function integrity, which converts φ into integrity constraints $\mathcal{I}$ by assigning a fresh Boolean variable to each subformula of φ, using the rules described in Figure 6.4. The variable u_φ corresponds to formula φ. We use $\mathcal{U}$ to denote the set of Boolean variables created when converting φ to $\mathcal{I}$. Algorithm 16 also uses a second abstract domain $\mathcal{P}$, each of whose elements represents a set of Boolean assignments in $\mathbb{P}(\mathcal{U} \to \{0, 1\})$. For simplicity, in this presentation $\mathcal{P}$ is the Cartesian domain $(\mathcal{U} \to \{0, 1, *\})_\perp$, but other more-expressive Boolean domains could be used.

$$\frac{\varphi := \ell \qquad \ell \in \text{literal}(\mathcal{L})}{u_\varphi \Leftrightarrow \ell \ \in \mathcal{I}} \textsc{Leaf} \qquad\qquad \frac{\varphi := \varphi_1 \text{op}\ \varphi_2}{u_\varphi \Leftrightarrow (u_{\varphi_1}\ \text{op}\ u_{\varphi_2}) \ \in \mathcal{I}} \textsc{Internal}$$

Figure 6.4: Rules used to convert a formula $\varphi \in \mathcal{L}$ into a set of integrity constraints $\mathcal{I}$. op represents any binary connective in $\mathcal{L}$, and literal($\mathcal{L}$) is the set of atomic formulas and their negations.

On line 2 of Algorithm 16, an element of $\mathcal{P}$ is created in which u_φ is assigned the value 1, which asserts that φ is true. Algorithm 16 is parameterized by the value of k (where $k \geq 0$). Let $\gamma_{\mathcal{I}}((P, A))$ denote $\gamma((P, A)) \cap [\![\mathcal{I}]\!]$. The call to k-assume on line 3 returns $(\widetilde{P}, \widetilde{A})$, which is a semantic reduction of (P, A) with respect to $\mathcal{I}$; that is, $\gamma_{\mathcal{I}}((\widetilde{P}, \widetilde{A})) = \gamma_{\mathcal{I}}((P, A))$ and $(\widetilde{P}, \widetilde{A}) \sqsubseteq (P, A)$. In general, the greater the value of k, the more precise is the result computed by Algorithm 16. The next theorem establishes that Algorithm 16 computes an over-approximation of $\text{Assume}[\varphi](A)$.

Theorem 6.4. *For all $\varphi \in \mathcal{L}$, $A \in \mathcal{A}$, if $\widetilde{A} = \widetilde{Assume}[\varphi](A)$, then $\gamma(\widetilde{A}) \supseteq [\![\varphi]\!] \cap \gamma(A)$, and $\widetilde{A} \sqsubseteq A$.*

Proof. Line 1 calls the function integrity, which converts φ into integrity constraints $\mathcal{I}$ by assigning a fresh Boolean variable to each subformula of φ, using the rules described in Figure 6.4. u_φ is assigned to formula φ. We use $\mathcal{U}$ to denote the set of fresh variables created when converting φ to $\mathcal{I}$, and we use $\mathcal{V}$ to denote the vocabulary of the abstract domain $\mathcal{A}$.

Thus, after line 1 we have that

$$[\![\varphi]\!] = ([\![\mathcal{I}]\!] \cap [\![u_\varphi]\!]) \downarrow_{\mathcal{V}}, \tag{6.1}$$

where $''M \downarrow \mathcal{V}''$ denotes the operation of discarding all constants, functions, and relations of a model M that are not in the vocabulary $\mathcal{V}$. On line 2 of Algorithm 16, an element P of the Cartesian domain $\mathcal{P}$ is created in which u_φ is assigned the value 1. Thus, we have that

$$[\![u_\varphi]\!] = \gamma((P, \top)) \tag{6.2}$$

Using Equations (6.1) and (6.2), we obtain

$$[\![\varphi]\!] = ([\![\mathcal{I}]\!] \cap \gamma((P, \top_{\mathcal{A}}))) \downarrow_{\mathcal{V}} \tag{6.3}$$

$$[\![\varphi]\!] \cap \gamma(A) = ([\![\mathcal{I}]\!] \cap \gamma((P, \top_{\mathcal{A}}))) \downarrow_{\mathcal{V}} \cap \gamma(A) \tag{6.4}$$

$$[\![\varphi]\!] \cap \gamma(A) = ([\![\mathcal{I}]\!] \cap \gamma((P, \top_{\mathcal{A}}))) \downarrow_{\mathcal{V}} \cap \gamma((\top_{\mathcal{P}}, A)) \downarrow_{\mathcal{V}} \tag{6.5}$$

$$[\![\varphi]\!] \cap \gamma(A) = ([\![\mathcal{I}]\!] \cap \gamma((P, \top_{\mathcal{A}})) \cap \gamma((\top_{\mathcal{P}}, A))) \downarrow_{\mathcal{V}} \tag{6.6}$$

$$[\![\varphi]\!] \cap \gamma(A) = ([\![\mathcal{I}]\!] \cap \gamma((P, A))) \downarrow_{\mathcal{V}} \tag{6.7}$$

After line 3, we have that

$$(\widetilde{P}, \widetilde{A}) = \text{k-assume}[\mathcal{I}]((P, A)) \tag{6.8}$$

Using Theorem 6.5 on Equation (6.8), we have that

$$\gamma_{\mathcal{I}}((\widetilde{P}, \widetilde{A})) = \gamma_{\mathcal{I}}((P, A)) \tag{6.9}$$

$$(\widetilde{P}, \widetilde{A}) \sqsubseteq (P, A) \tag{6.10}$$

Using Equation (6.10), we immediately have

$$\widetilde{A} \sqsubseteq A \tag{6.11}$$

Using Equation (6.9), we have that

$$\gamma((\widetilde{P}, \widetilde{A})) \cap [\![I]\!] = \gamma((P, A)) \cap [\![I]\!]$$

$$(\gamma((\widetilde{P}, \widetilde{A})) \cap [\![I]\!]) \downarrow_{\mathcal{V}} = (\gamma((P, A)) \cap [\![I]\!]) \downarrow_{\mathcal{V}}$$

Using Equation (6.7), we have that

$$(\gamma((\widetilde{P}, \widetilde{A})) \cap [\![I]\!]) \downarrow_{\mathcal{V}} = [\![\varphi]\!] \cap \gamma(A)$$

$$\gamma((\widetilde{P}, \widetilde{A})) \downarrow_{\mathcal{V}} \supseteq [\![\varphi]\!] \cap \gamma(A)$$

$$\gamma(\widetilde{A}) \supseteq [\![\varphi]\!] \cap \gamma(A)$$

□

Algorithm 18 presents the algorithm to compute k-assume, for $k \geq 1$. Given the integrity constraints $\mathcal{I}$, and the current abstract value (P, A), k-assume$[\mathcal{I}]((P, A))$ returns an abstract value that is a semantic reduction of (P, A) with respect to $\mathcal{I}$. The crux of the computation is the inner loop body, lines 4–10, which implements an analog of the Dilemma Rule from Stålmarck's method (Sheeran and Stålmarck, 2000).

The steps of the Dilemma Rule are shown schematically in Figure 6.3(b). At line 3 of Algorithm 18, a Boolean variable u whose value is unknown is chosen. $B = \top_{\mathcal{P}}[u \mapsto 0]$ and its complement $\overline{B} = \top_{\mathcal{P}}[u \mapsto 1]$ are used to split the current abstract value (P_0, A_0) into two abstract values $(P_1, A_1) = (P, A) \sqcap (B, \top)$ and $(P_2, A_2) = (P, A) \sqcap (\overline{B}, \top)$, as shown in lines 6 and 7.

The calls to $(\mathrm{k}{-}1)$-assume at lines 8 and 9 compute semantic reductions of (P_1, A_1) and (P_2, A_2) with respect to $\mathcal{I}$, which creates (P_1', A_1') and (P_2', A_2'), respectively. Finally, at line 10 (P_1', A_1') and (P_2', A_2') are merged by performing a join. (The result is labeled (P_3, A_3) in Figure 6.3(b).)

The steps of the Dilemma Rule (Figure 6.3(b)) are repeated until a fixpoint is reached, or some resource bound is exceeded. The next theorem establishes that $\mathrm{k\text{-}assume}[\mathcal{I}]((P, A))$ computes a semantic reduction of (P, A) with respect to $\mathcal{I}$.

Theorem 6.5. *For all $P \in \mathcal{P}$ and $A \in \mathcal{A}$, if $(P', A') = \mathrm{k\text{-}assume}[\mathcal{I}]((P, A))$, then $\gamma_{\mathcal{I}}((P', A')) = \gamma_{\mathcal{I}}((P, A))$ and $(P', A') \sqsubseteq (P, A)$.*

Proof. We prove this via induction on k. Theorem 6.6 proves the base case when $k = 0$.

To prove the inductive case, assume that Algorithm 18 is sound for $k - 1$, i.e., for all $P \in \mathcal{P}$ and $A \in \mathcal{A}$, if $(P', A') = (\text{k-1})\text{-assume}[\mathcal{I}]((P, A))$, then

$$\gamma_{\mathcal{I}}((P', A')) = \gamma_{\mathcal{I}}((P, A)) \text{ and } (P', A') \sqsubseteq (P, A) \tag{6.12}$$

Let (P^0, A^0) be the value of (P, A) passed as input to Algorithm 18, and (P^i, A^i) be the value of (P, A) computed at the end of the i^{th} iteration of the loop body consisting of lines 4–10 (that is, the value computed at line 10).

We show via induction that, for each i, $\gamma_{\mathcal{I}}((P^i, A^i)) = \gamma_{\mathcal{I}}((P^0, A^0))$ and $(P^i, A^i) \sqsubseteq (P^0, A^0)$.

We first prove the base case for $i = 1$; that is, $\gamma_{\mathcal{I}}((P^1, A^1)) = \gamma_{\mathcal{I}}((P^0, A^0))$ and $(P^1, A^1) \sqsubseteq (P^0, A^0)$.

At line 3 of Algorithm 18, a variable $u \in \mathcal{U}$ is chosen such that $P^0(u) = *$. After line 5, $B = \top_{\mathcal{P}}[u \mapsto 0]$ and $\overline{B} = \top_{\mathcal{P}}[u \mapsto 1]$, and, thus, we have that

$$\gamma((B, \top)) \cup \gamma((\overline{B}, \top) = \gamma((\top, \top)) \tag{6.13}$$

B and $\overline{B}$ are used to split the current abstract value (P^0, A^0) into two abstract values.

After line 6 of 18, we have that $(P_1, A_1) = (P^0, A^0) \sqcap (B, \top)$, which implies

$$\gamma((P_1, A_1)) \supseteq \gamma((P^0, A^0)) \cap \gamma((B, \top)) \tag{6.14}$$

Similarly, after line 7 of Algorithm 18, we have that

$$\gamma((P_2, A_2)) \supseteq \gamma((P^0, A^0)) \cap \gamma((\overline{B}, \top)) \tag{6.15}$$

From Equations (6.14) and (6.15), we have that

$$\begin{aligned}
\gamma((P_1, A_1)) \cup \gamma((P_2, A_2)) &\supseteq (\gamma((P^0, A^0)) \cap \gamma((B, \top))) \cup (\gamma((P^0, A^0)) \cap \gamma((\overline{B}, \top))) \\
&\supseteq \gamma((P^0, A^0)) \cap (\gamma((B, \top)) \cup \gamma((\overline{B}, \top)) \\
&\supseteq \gamma((P^0, A^0)) \cap \gamma((\top, \top)) \text{ (using Equation (6.13))} \\
&\supseteq \gamma((P^0, A^0))
\end{aligned}$$

Thus, we obtain

$$\gamma_{\mathcal{I}}((P_1, A_1)) \cup \gamma_{\mathcal{I}}((P_2, A_2)) \supseteq \gamma_{\mathcal{I}}((P^0, A^0)) \tag{6.16}$$

After lines 6 and 7 of Algorithm 18 we also have that

$$(P_1, A_1) \sqsubseteq (P^0, A^0) \tag{6.17}$$

$$(P_2, A_2) \sqsubseteq (P^0, A^0) \tag{6.18}$$

After line 8 of Algorithm 18, we have that

$$(P_1', A_1') = (\text{k}{-}1)\text{-assume}[\mathcal{I}]((P_1, A_1)) \tag{6.19}$$

By using the induction hypothesis (Equation (6.12)) in Equation (6.19), we obtain

$$(P_1', A_1') \sqsubseteq (P_1, A_1) \tag{6.20}$$

Using Equations (6.17) and (6.20), we obtain

$$(P_1', A_1') \sqsubseteq (P^0, A^0) \tag{6.21}$$

Similarly, after line 9 of Algorithm 18, we obtain

$$(P_2', A_2') \sqsubseteq (P^0, A^0) \tag{6.22}$$

Using Equations (6.21) and (6.22), we obtain

$$(P_1', A_1') \sqcup (P_2', A_2') \sqsubseteq (P^0, A^0) \tag{6.23}$$

Using Equation (6.23), we have that

$$\gamma((P_1', A_1') \sqcup (P_2', A_2')) \subseteq \gamma((P^0, A^0))$$

$$\gamma((P_1', A_1') \sqcup (P_2', A_2')) \cap [\![\mathcal{I}]\!] \subseteq \gamma((P^0, A^0)) \cap [\![\mathcal{I}]\!]$$

This gives us

$$\gamma_{\mathcal{I}}((P'_1, A'_1) \sqcup (P'_2, A'_2)) \subseteq \gamma_{\mathcal{I}}((P^0, A^0)) \tag{6.24}$$

After line 10 of Algorithm 18, we have that $(P^1, A^1) = (P'_1, A'_1) \sqcup (P'_2, A'_2)$. Using Equation (6.23), we obtain

$$(P^1, A^1) \sqsubseteq (P^0, A^0) \tag{6.25}$$

Using Equation (6.24), we obtain

$$\gamma_{\mathcal{I}}(P^1, A^1) \subseteq \gamma_{\mathcal{I}}((P^0, A^0)) \tag{6.26}$$

Furthermore, by the induction hypothesis (Equation (6.12)), after lines 8 and 9 of Algorithm 18, we also have

$$\gamma_{\mathcal{I}}((P'_1, A'_1)) = \gamma_{\mathcal{I}}((P_1, A_1)) \tag{6.27}$$

$$\gamma_{\mathcal{I}}((P'_2, A'_2)) = \gamma_{\mathcal{I}}((P_2, A_2)) \tag{6.28}$$

After line 10 of Algorithm 18, we have that

$$\gamma_{\mathcal{I}}((P^1, A^1)) = \gamma_{\mathcal{I}}((P'_1, A'_1) \sqcup (P'_2, A'_2))$$

$$\gamma_{\mathcal{I}}((P^1, A^1)) \supseteq \gamma_{\mathcal{I}}((P'_1, A'_1)) \cup \gamma_{\mathcal{I}}((P'_2, A'_2))$$

Using Equations (6.27) and (6.28), we have that

$$\gamma_{\mathcal{I}}((P^1, A^1)) \supseteq \gamma_{\mathcal{I}}((P_1, A_1)) \cup \gamma_{\mathcal{I}}((P_2, A_2))$$

Using Equation (6.16), we obtain

$$\gamma_{\mathcal{I}}((P^1, A^1)) \supseteq \gamma_{\mathcal{I}}((P^0, A^0)) \tag{6.29}$$

Using Equations (6.26) and (6.29), we have that

$$\gamma_{\mathcal{I}}((P^1, A^1)) = \gamma_{\mathcal{I}}((P^0, A^0)) \tag{6.30}$$

Thus, using Equations (6.25) and (6.30) we have proved the base case with $i = 1$. An almost identical proof can be used to prove the inductive case for the induction on i. Thus, for each i, $\gamma_{\mathcal{I}}((P^i, A^i)) = \gamma_{\mathcal{I}}((P^0, A^0))$ and $(P^i, A^i) \sqsubseteq (P^0, A^0)$.

Because the value returned by Algorithm 18 is the final value computed at line 10, we have proven the inductive case for the induction on k. Thus, by induction, we have shown that for all $P \in \mathcal{P}$ and $A \in \mathcal{A}$, if $(P', A') = \text{k-assume}[\mathcal{I}]((P, A))$, then $\gamma_{\mathcal{I}}((P', A')) = \gamma_{\mathcal{I}}((P, A))$ and $(P', A') \sqsubseteq (P, A)$. $\square$

$$\frac{J = (u_1 \Leftrightarrow (u_2 \vee u_3)) \in \mathcal{I} \qquad P(u_1) = 0}{(P \sqcap \top[u_2 \mapsto 0, u_3 \mapsto 0], A)}\ \textsc{Or1} \qquad \frac{J = (u_1 \Leftrightarrow (u_2 \wedge u_3)) \in \mathcal{I} \qquad P(u_1) = 1}{(P \sqcap \top[u_2 \mapsto 1, u_3 \mapsto 1], A)}\ \textsc{And1}$$

Figure 6.5: Boolean rules used by Algorithm 17 in the call $\mathrm{InternalRule}(J, (P, A))$.

$$\frac{J = (u \Leftrightarrow l) \in \mathcal{I} \qquad P(u) = 1}{(P, A \sqcap \mu\widetilde{\alpha}_{\mathcal{A}}(l))}\ \textsc{ToA-1} \qquad \frac{J = (u \Leftrightarrow l) \in \mathcal{I} \qquad P(u) = 0}{(P, A \sqcap \mu\widetilde{\alpha}_{\mathcal{A}}(\neg l))}\ \textsc{ToA-0}$$

$$\frac{J = (u \Leftrightarrow \ell) \in \mathcal{I} \qquad A \sqcap \mu\widetilde{\alpha}_{\mathcal{A}}(l) = \bot_{\mathcal{A}}}{(P \sqcap \top[u \mapsto 0], A)}\ \textsc{AtoP-0}$$

Figure 6.6: Rules used by Algorithm 17 in the call $\mathrm{LeafRule}(J, (P, A))$.

Algorithm 17 describes the algorithm to compute 0-assume: given the integrity constraints $\mathcal{I}$, and an abstract value (P, A), $0\text{-assume}[\mathcal{I}]((P, A))$ returns an abstract value (P', A') that is a semantic reduction of (P, A) with respect to $\mathcal{I}$. It is in this algorithm that information is passed between the component abstract values $P \in \mathcal{P}$ and $A \in \mathcal{A}$ via *propagation rules*, like the ones shown in Figures 6.5 and 6.6. In lines 4–7 of Algorithm 17, these rules are applied by using a single integrity constraint in $\mathcal{I}$ and the current abstract value (P, A).

Given $J \in \mathcal{I}$ and (P, A), the net effect of applying any of the propagation rules is to compute a semantic reduction of (P, A) with respect to $J \in \mathcal{I}$. The propagation rules used in Algorithm 17 can be classified into two categories:

1. Rules that apply on line 7 when J is of the form $p \Leftrightarrow (q \text{ op } r)$, shown in Figure 6.5. Such an integrity constraint is generated from each internal subformula of formula φ. These rules compute a non-trivial semantic reduction of P with respect to J by only using information from P. For instance, rule And1 says that if J is of the form $p \Leftrightarrow (q \wedge r)$, and p is 1 in P, then we can infer that both q and r must be 1. Thus, $P \sqcap \top[q \mapsto 1, r \mapsto 1]$ is a semantic reduction of P with respect to J. (See Example 6.2, step 1.)

2. Rules that apply on line 5 when J is of the form $u \Leftrightarrow \ell$, shown in Figure 6.6. Such an

integrity constraint is generated from each leaf of the original formula φ. This category of rules can be further subdivided into

a) Rules that propagate information from abstract value P to abstract value A; viz., rules ProA-0 and ProA-1. For instance, rule ProA-1 states that given $J = u \Leftrightarrow l$, and $P(u) = 1$, then $A \sqcap \mu\widetilde{\alpha}(l)$ is a semantic reduction of A with respect to J. (See Example 6.2, steps 2 and 3.)

b) Rule AroP-0, which propagates information from abstract value A to abstract value P. Rule AroP-0 states that if $J = (u \Leftrightarrow \ell)$ and $A \sqcap \mu\widetilde{\alpha}(l) = \bot_{\mathcal{A}}$, then we can infer that u is false. Thus, the value of $P \sqcap \top[u \mapsto 0]$ is a semantic reduction of P with respect to J. (See Example 6.2, step 4.)

Algorithm 17 repeatedly applies the propagation rules until a fixpoint is reached, or some resource bound is reached. The next theorem establishes that 0-assume computes a semantic reduction of (P, A) with respect to $\mathcal{I}$.

Theorem 6.6. *For all* $P \in \mathcal{P}, A \in \mathcal{A}$, *if* $(P', A') = 0\text{-assume}[\mathcal{I}]((P, A))$, *then* $\gamma_{\mathcal{I}}((P', A')) = \gamma_{\mathcal{I}}((P, A))$ *and* $(P', A') \sqsubseteq (P, A)$.

Proof. Let (P^0, A^0) be the value of (P, A) passed as input to Algorithm 17, and (P^i, A^i) be the value of (P, A) computed at the end of the i^{th} iteration of the loop body consisting of lines 4–7 (that is, the value computed at line 5 or line 7).

We show via induction that, for each i, $\gamma_{\mathcal{I}}((P^i, A^i)) = \gamma_{\mathcal{I}}((P^0, A^0))$ and $(P^i, A^i) \sqsubseteq (P^0, A^0)$.

We first prove the base case for $i = 1$; that is, $\gamma_{\mathcal{I}}((P^1, A^1)) = \gamma_{\mathcal{I}}((P^0, A^0))$, and $(P^1, A^1) \sqsubseteq (P^0, A^0)$. After line 3 of Algorithm 17, we have that

$$J \in \mathcal{I} \tag{6.31}$$

From Equation (6.31), we have that

$$[\![J]\!] \supseteq [\![\mathcal{I}]\!] \tag{6.32}$$

It is easy to show that, given $J \in \mathcal{I}$ and (P^0, A^0), the net effect of applying any of the propagation rules in Figures 6.5 and 6.6 is to compute a semantic reduction of (P^0, A^0) with

respect to $J \in \mathcal{I}$. Thus, after the i^{th} iteration of the loop body, we have that

$$\gamma((P^1, A^1)) \cap [\![J]\!] = \gamma((P^0, A^0)) \cap [\![J]\!] \tag{6.33}$$

$$(P^1, A^1) \sqsubseteq (P^0, A^0) \tag{6.34}$$

Using Equation (6.33),

$$(\gamma((P^1, A^1)) \cap [\![J]\!]) \cap [\![\mathcal{I}]\!] = (\gamma((P^0, A^0)) \cap [\![J]\!]) \cap [\![\mathcal{I}]\!] \tag{6.35}$$

Using Equations (6.32) and (6.35),

$$\gamma((P^1, A^1)) \cap [\![\mathcal{I}]\!] = \gamma((P^0, A^0)) \cap [\![\mathcal{I}]\!] \tag{6.36}$$

This gives us

$$\gamma_{\mathcal{I}}((P^1, A^1)) = \gamma_{\mathcal{I}}((P^0, A^0)) \tag{6.37}$$

Thus, Equations (6.34) and (6.37) prove the base case of the induction for $i = 1$. An almost identical proof can be used to prove the inductive case for the induction on i. Thus, for each i, $\gamma_{\mathcal{I}}((P^i, A^i)) = \gamma_{\mathcal{I}}((P^0, A^0))$ and $(P^i, A^i) \sqsubseteq (P^0, A^0)$.

Because the value returned by Algorithm 17 is the final value of (P, A) computed at line 5 or line 7, we have shown that for all $P \in \mathcal{P}, A \in \mathcal{A}$, if $(P', A') = 0\text{-assume}[\mathcal{I}]((P, A))$, then $\gamma_{\mathcal{I}}((P', A')) = \gamma_{\mathcal{I}}((P, A))$ and $(P', A') \sqsubseteq (P, A)$. □

6.3 Instantiations

In this section, we describe instantiations of our framework for two logical-language/abstract-domain pairs: (QFBV, $\mathcal{E}_{2^w}$) and (LRA, Polyhedra). For each instantiation, we describe the $\mu\widetilde{\alpha}$ operation. We say that an $\widetilde{\text{Assume}}$ algorithm is *complete* for a logic $\mathcal{L}$ and abstract domain $\mathcal{A}$ if it is guaranteed to compute the best value $\widehat{\text{Assume}}[\varphi](A)$ for $\varphi \in \mathcal{L}$ and $A \in \mathcal{A}$. We give conditions under which the two instantiations are complete.

6.3.1 Bitvector Affine-Equalities Domain (QFBV, $\mathcal{E}_{2^w}$)

Given a literal $l \in$ QFBV, we compute $\mu\widetilde{\alpha}_{\mathcal{E}_{2^w}}(l)$ by invoking $\widetilde{\alpha}^{\updownarrow}_{\mathcal{E}_{2^w}}(l)$. That is, we harness $\widetilde{\alpha}^{\updownarrow}_{\mathcal{E}_{2^w}}$ in service of $\widetilde{\text{Assume}}_{\mathcal{E}_{2^w}}$, but only for $\mu\widetilde{\alpha}_{\mathcal{E}_{2^w}}$, which means that $\widetilde{\alpha}^{\updownarrow}_{\mathcal{E}_{2^w}}$ is only applied to literals.

Algorithm 16 is not complete for (QFBV,$\mathcal{E}_{2^w}$). Let x be a bitvector of width 2, and let $\varphi = (x \neq 0 \wedge x \neq 1 \wedge x \neq 2)$. Thus, $\widehat{\text{Assume}}[\varphi](\top_{\mathcal{E}_{2^2}}) = \{x - 3 = 0\}$. The $\mathcal{E}_{2^2}$ domain is not expressive enough to represent disequalities. For instance, $\mu\widetilde{\alpha}(x \neq 0)$ equals $\top_{\mathcal{E}_{2^2}}$. Because Algorithm 16 considers only a single integrity constraint at a time, we obtain $\widetilde{\text{Assume}}[\varphi](\top_{\mathcal{E}_{2^2}}) = \mu\widetilde{\alpha}(x \neq 0) \sqcap \mu\widetilde{\alpha}(x \neq 1) \sqcap \mu\widetilde{\alpha}(x \neq 2) = \top_{\mathcal{E}_{2^2}}$.

The current approach can be made complete for (QFBV, $\mathcal{E}_{2^w}$) by making 0-assume consider multiple integrity constraints during propagation (in the limit, having to call $\mu\widetilde{\alpha}(\varphi)$). For the affine subset of QFBV, an alternative approach would be to perform a 2^w-way split on the $\mathcal{E}_{2^w}$ value each time a disequality is encountered, where w is the bit-width—in effect, rewriting $x \neq 0$ to $(x = 1 \vee x = 2 \vee x = 3)$ in the example above. Furthermore, if there is a $\mu\widetilde{\text{Assume}}$ operation, then the second approach can be extended to handle all of QFBV: $\mu\widetilde{\text{Assume}}[\ell](A)$ would be used to take the current $\mathcal{E}_{2^w}$ abstract value A and a literal ℓ, and return an over-approximation of $\widetilde{\text{Assume}}[\ell](A)$. All these approaches would be prohibitively expensive. Our current approach, though theoretically not complete, works very well in practice (see Section 6.4).

6.3.2 Polyhedral Domain (LRA, Polyhedra)

The second instantiation that we implemented is for the logic LRA and the polyhedral domain (Cousot and Halbwachs, 1978). Because an LRA disequality $t \neq 0$ can be normalized to $(t < 0 \vee t > 0)$, every literal l in a normalized LRA formula is merely a half-space in the polyhedral domain. Consequently, $\mu\widetilde{\alpha}_{\text{Polyhedra}}(l)$ is exact, and easy to compute. Furthermore, because of this precision, the $\widetilde{\text{Assume}}$ algorithm is complete for (LRA, Polyhedra). In particular, if $k = |\varphi|$, then k-assume is sufficient to guarantee that $\widetilde{\text{Assume}}[\varphi](A)$ returns $\widehat{\text{Assume}}[\varphi](A)$. For polyhedra, our implementation uses PPL (Bagnara et al., 2008).

The observation in the last paragraph applies in general: if $\mu\widetilde{\alpha}_{\mathcal{A}}(l)$ is exact for all literals $l \in \mathcal{L}$, then Algorithm 16 is complete for logic $\mathcal{L}$ and abstract domain $\mathcal{A}$.

name	instrs	procs	BBs	brs
write	232	10	134	26
finger	532	18	298	48
subst	1093	16	609	74
label	1167	16	573	103
chkdsk	1468	18	787	119
convert	1927	38	1013	161
route	1982	40	931	243
comp	2377	35	1261	224
logoff	2470	46	1145	306
setup	4751	67	1862	589

Table 6.1: The characteristics of the x86 binaries of Windows utilities used in the experiments. The columns show the number of instructions (instrs); the number of procedures (procs); the number of basic blocks (BBs); the number of branch instructions (brs).

6.4 Experimental Evaluation

6.4.1 Computing Representations of Abstract Transformers

In this section, I compare two methods for computing abstract transformers for the affine-equality domain $\mathcal{E}_{2^{32}}$:

- the $\widetilde{\alpha}^{\updownarrow}$-based framework (Chapter 5), instantiated for $\mathcal{E}_{2^{32}}$, and
- the $\widetilde{\alpha}^{\downarrow}$-based framework described in this chapter, instantiated for $\mathcal{E}_{2^{32}}$.

The experiments were designed to answer the following questions:

1. How does the speed of $\widetilde{\alpha}^{\downarrow}$ compare with that of $\widetilde{\alpha}^{\updownarrow}$, especially when each call to $\widehat{\alpha}$ is given a fixed time limit?
2. How does the precision of $\widetilde{\alpha}^{\downarrow}$ compare with that of $\widetilde{\alpha}^{\updownarrow}$, especially when each call to $\widehat{\alpha}$ is given a fixed time limit?

To address these questions, we performed affine-relation analysis (ARA) on x86 machine code, computing affine-equality relations over the x86 registers.

	$\widetilde{\alpha}^{\updownarrow}_{t=\infty}$	$\widetilde{\alpha}^{\updownarrow}_{t=1}$	$\widetilde{\alpha}^{\downarrow}_{t=\infty}$	$\widetilde{\alpha}^{\downarrow}_{t=1}$
write	52.10	19.33	10.01	**6.536**
finger	99.42	31.51	29.03	**12.35**
subst	118.1	51.62	27.45	**17.03**
label	89.84	57.13	28.00	**18.72**
chkdsk	249.3	88.56	106.0	**33.96**
route	438.7	124.1	95.56	**43.89**
convert	212.4	98.02	83.56	**34.82**
comp	588.6	149.3	105.9	**46.78**
logoff	481.4	176.8	168.9	**75.11**
setup	1318	358.7	476.5	**177.7**

Table 6.2: Time taken, in seconds, by the $\widetilde{\alpha}^{\updownarrow}$ and $\widetilde{\alpha}^{\downarrow}$ algorithms for WPDS construction. The subscript is used to denote the time limit used for a single invocation of a call to $\widetilde{\alpha}$: $t = \infty$ denotes that no time limit was given, and $t = 1$ denotes that a time limit of 1 second was given. For each benchmark, the algorithm that takes the least time is highlighted in bold.

The experimental setup in this section differs slightly from that used in Sections 4.3 and 5.5 in the following two ways:

- The $\widetilde{\alpha}^{\updownarrow}$ and $\widetilde{\alpha}^{\downarrow}$ algorithms are passed an over-approximation of the answer computed using operator-by-operator reinterpretation.[2] This scenario better reflects how $\widetilde{\alpha}$ frameworks are likely to be used in practice, and also highlights the fact that these two frameworks can make use of an initial over-approximation. A (non-trivial) initial over-approximation aids the Bilateral algorithm ($\widetilde{\alpha}^{\updownarrow}$) by guiding the set of abstract consequences it chooses.

- The set of benchmarks used in this experiment (Table 6.1) is a strict superset of the benchmarks used in Sections 4.3 and 5.5.

Apart from the above two items, the rest of the experimental setup is the same as that used in Sections 4.3 and 5.5.

Table 6.2 shows the time taken, in seconds, by the $\widetilde{\alpha}^{\updownarrow}$ and $\widetilde{\alpha}^{\downarrow}$ algorithms for WPDS construction. I also ran each of the algorithms with and without a total time limit on a single invocation of a call to $\widetilde{\alpha}$. The subscript is used to denote the time limit used for a single invocation of a call to $\widetilde{\alpha}$:

[2]Section 3.1.2 briefly describes the TSL system used for computing transformers via reinterpretation.

	$\widetilde{\alpha}^{\updownarrow}_{t=\infty}$ better than $\widetilde{\alpha}^{\downarrow}_{t=1}$	$\widetilde{\alpha}^{\downarrow}_{t=1}$ better than $\widetilde{\alpha}^{\updownarrow}_{t=\infty}$
write	0%	11.54%
finger	0%	12.50%
subst	0%	12.16%
label	0%	4.85%
chkdsk	9.24%	0%
route	6.17%	4.53%
convert	0%	0%
comp	0%	1.34%
logoff	11.76%	10.78%
setup	0%	4.70%

Table 6.3: Comparison of the degree of improvement gained by using $\widetilde{\alpha}^{\updownarrow}_{t=\infty}$ and $\widetilde{\alpha}^{\downarrow}_{t=1}$ (measured as the percentage of control points for which the inferred one-vocabulary affine relation inferred by one analysis was strictly more precise than that inferred by the other analysis).

$t = \infty$ denotes that no time limit was given, and $t = 1$ denotes that a time limit of 1 second was given. From Table 6.2, we see that $\widetilde{\alpha}^{\downarrow} t = 1$ is 7.5 times faster than $\widetilde{\alpha}^{\updownarrow}_{t=\infty}$, and 2.7 times faster than $\widetilde{\alpha}^{\updownarrow}_{t=1}$ (computed as the geometric mean). This table answers question 1 stated above.

To answer question 2, we compare the precision of the WPDS analysis when using $\widetilde{\alpha}^{\updownarrow}$ with the precision obtained using $\widetilde{\alpha}^{\downarrow}$. In particular, we compare the affine-relation invariants computed by the $\widetilde{\alpha}^{\downarrow}$-based and $\widetilde{\alpha}^{\updownarrow}$-based analyses for each control point. Table 6.3 compares the precision of $\widetilde{\alpha}^{\updownarrow}_{t=\infty}$ and $\widetilde{\alpha}^{\downarrow}_{t=1}$. We see that $\widetilde{\alpha}^{\downarrow}$ computes precise invariants even when given a much smaller time limit of 1 second. Though not shown in Table 6.3, the precision of $\widetilde{\alpha}^{\updownarrow}_{t=1}$ was slightly worse than $\widetilde{\alpha}^{\updownarrow}_{t=\infty}$, while the precision of $\widetilde{\alpha}^{\downarrow}_{t=\infty}$ was the same as that of $\widetilde{\alpha}^{\downarrow}_{t=1}$. Increasing the depth of the Dilemma Rule from $k = 1$ to $k = 2$ did not improve precision of $\widetilde{\alpha}^{\downarrow}$.

6.4.2 Checking Unsatisfiability of LRA formulas

The formula used in Example 6.3 is just one instance of a family of unsatisfiable LRA formulas (McMillan et al., 2009). Let $\chi_d = (a_d < a_0) \wedge \bigwedge_{i=0}^{d-1}((a_i < b_i) \wedge (a_i < c_i) \wedge ((b_i < a_{i+1}) \vee (c_i < a_{i+1})))$. The formula ψ in Example 6.3 is χ_2; that is, the number of "diamonds" is 2 (see Figure 6.3(a)). These "diamond formulas" appear in the naive encoding of sequences of if-then-elses

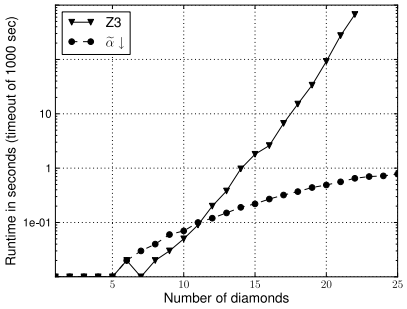


Figure 6.7: Semilog plot of Z3 vs. $\widetilde{\alpha}^{\downarrow}$ on χ_d formulas.

in programs, for instance, when trying to compute the worst-case execution time (Henry et al., 2014). We used the (LRA, Polyhedra) instantiation of our framework to check whether $\widetilde{\alpha}(\chi_d) = \bot$ for $d = 1 \ldots 25$ using 1-assume. We ran this experiment on a single processor of a 16-core 2.4 GHz Intel Zeon computer running 64-bit RHEL Server release 5.7. The semilog plot in Figure 6.7 compares the running time of $\widetilde{\alpha}^{\downarrow}$ with that of Z3, version 3.2 (de Moura and Bjørner, 2008). The time taken by Z3 increases exponentially with d, exceeding the timeout threshold of 1000 seconds for $d = 23$. These measurements corroborate the results of a similar experiment conducted by McMillan et al. (2009), where the reader can also find an in-depth explanation of Z3's behavior on this family of formulas.

On the other hand, the running time of $\widetilde{\alpha}^{\downarrow}$ increases linearly with d taking 0.78 seconds for $d = 25$. The cross-over point is $d = 12$. In Example 6.3, we saw how two successive applications of the Dilemma Rule suffice to prove that ψ is unsatisfiable. That explanation generalizes to χ_d: d applications of the Dilemma Rule are sufficient to prove unsatisfiability of χ_d. The order in which Boolean variables with unknown truth values are selected for use in the Dilemma Rule has no bearing on this linear behavior, as long as no variable is starved from being chosen (i.e., a fair-choice schedule is used). Each application of the Dilemma Rule is able to infer that $a_i < a_{i+1}$ for some i.

We do not claim that $\widetilde{\alpha}^{\downarrow}$ is always better than mature SMT solvers such as Z3. We do believe that it represents another interesting point in the design space of SMT solvers, similar in nature

to the GDPLL algorithm (McMillan et al., 2009) and the k-lookahead technique used in the DPLL($\sqcup$) algorithm (Bjørner and de Moura, 2008).

6.5 Related Work

In the context of checking functional equivalence of sequential circuits, van Eijk (1998) and Bjesse and Claessen (2000) present symbolic-abstraction algorithms for the abstract domain of Boolean equalities. Van Eijk performs symbolic abstraction using BDDs. Bjesse and Claessen convert van Eijk's algorithm to use Stålmarck's method (Sheeran and Stålmarck, 2000) instead of BDDs. Furthermore, Bjesse and Claessen (2000) extend Stålmarck's method from using equivalences between variables to implications between variables.

Recent work has also explored connections between abstract interpretation and decision procedures (D'Silva et al., 2012, 2013; Haller, 2013; D'Silva et al., 2014). In particular, D'Silva et al. (2013) generalize the algorithm for Conflict Driven Clause Learning used in SAT solvers to solve the lattice-theoretic problem of determining if an additive transformer on a Boolean lattice is always bottom. In contrast, our algorithms address a broader class of problems. Our algorithms apply to non-Boolean lattices. Moreover, provided there are no timeouts, our algorithms are capable of discovering if the most-precise answer is $\bot$. However, they are also useful when the most-precise answer is not $\bot$; in particular, our algorithms can be used to compute best transformers. Furthermore, as described in Chapter 7, our algorithms can be used to solve the BII problem (assuming no timeouts), and even when there are timeouts, they generate inductive program invariants. Our work and that of D'Silva et al. were performed independently and contemporaneously.

6.6 Chapter Notes

I first came across Stålmarck's method in Harrison (2009). The fact that there was a merge of proof branches involved was what stuck with me. Plenty of thinking and discussions with Tom

resulted in the simple insight that the intersection of constraints at the merge of the Dilemma Rule corresponds to the join of abstract values.

I spent a considerable amount of time trying to engineer a pure Stålmarck-based SAT/SMT solver that would be competitive with state-of-the-art solvers. Alas the tremendous progress in SAT solvers, such as variable-selection heuristics, and conflict-driven clause learning (CDCL), and my inadequate engineering skills forced me to abandon this approach; see Thakur and Reps (2012) for some preliminary experiments using my earlier prototype, and Chapter 10 for a more practical design that incorporates the best of the Stålmarck-style approach and the DPLL/CDCL approach.

One takeaway from my earlier prototype was the need for implementations of abstract domains that scaled as the number of variables increased; in my experience, the abstract operations of join and meet became prohibitively expensive when the number of variables was greater than 50.

Chapter 7

Computing Best Inductive Invariants

In Chapters 4, 5, and 6, symbolic abstraction was used to compute best abstract transformers for a sequence of statements. In this chapter, I show how symbolic abstraction can be used to compute *best inductive invariants* for an entire program.

Let $\mathcal{C}$ be the concrete domain that describes the collecting semantics of the program. For a concrete transformer τ, let $\mathrm{Post}[\tau] : \mathcal{C} \to \mathcal{C}$ denote the operator that applies the concrete transformer. A set of invariants $\{I_k\}$ are said be *inductive* with respect to a set of transformers $\{\tau_{ij}\}$ if, for all i and j, $\mathrm{Post}[\tau_{ij}](\llbracket I_i \rrbracket) \subseteq \llbracket I_j \rrbracket$, where $\llbracket I_k \rrbracket \in \mathcal{C}$ denotes the meaning of I_k.

The choice of a particular abstract domain $\mathcal{A}$ fixes a limit on the precision of the invariants identified by an analysis. If $\{I_k\}$ is a set of inductive invariants, and $I_j \in \mathcal{A}$ for each single member I_j of $\{I_k\}$, then the set of invariants $\{I_k\}$ are said to be inductive $\mathcal{A}$-*invariants*. (For brevity, I also refer to a single member I_j of $\{I_k\}$ as an inductive $\mathcal{A}$-invariant.) Furthermore, a *most-precise* inductive $\mathcal{A}$-invariant exists: $\mathrm{Post}[\tau]$ is monotonic in $\mathcal{C}$; given two $\mathcal{A}$-invariants, we can take their meet; thus, provided $\mathcal{A}$ is a meet semi-lattice, the most-precise inductive $\mathcal{A}$-invariant exists. (On the other hand, computing the *most precise $\mathcal{A}$-invariant* at a program point, defined as the abstraction of the collecting semantics at that point, is generally infeasible.)

In summary, the best inductive invariant (BII) problem can be stated as follows:

> Given program P, and an abstract domain $\mathcal{A}$, find the most-precise inductive $\mathcal{A}$-invariant for P.

BII is clearly an important problem because it represents the limit of obtainable precision for a given abstract domain. The key insights behind our approach are:

- The BII problem reduces to the problem of applying $\widehat{\mathrm{Post}}$.
- The problem of applying $\widehat{\mathrm{Post}}$ reduces to the problem of symbolic abstraction.

Because the symbolic-abstraction approach solves the BII problem (in the absence of timeouts), it can obtain more precise results than more conventional approaches to implementing abstract interpreters. In particular, the symbolic-abstraction approach can identify invariants and procedure summaries that are more precise than the ones obtained by more conventional approaches.

Not only does the work provide insight on fundamental limits in abstract interpretation, the algorithms that I present are also practical. Santini is an invariant-generation tool based on the principles of symbolic abstraction. Santini uses the predicate-abstraction domain that can infer arbitrary Boolean combinations of a given set of predicates. The implementation of the abstract domain was simple: just 1200 lines of C# code. We then compared the performance of Santini with Houdini (Flanagan and Leino, 2001), which is a well-established tool that infers only conjunctive invariants from a given set of predicates. We ran the Corral model checker (Lal et al., 2012) using invariants supplied by Houdini and Santini. For 19 examples for which Corral gave an indefinite result using Houdini-supplied invariants, invariants discovered using Santini allowed Corral to give a definite result in 9 cases (47%); see (Thakur et al., 2013, Section 5). This experiment shows that symbolic abstraction provides a powerful tool that can be used to implement automatically a correct and precise invariant generator that uses an expressive abstract domain. The other instantiation of our BII framework is based on WALi (Kidd et al., 2007), a tool for solving program-analysis problems using an abstract domain, which we have used to perform machine-code analysis (Thakur et al., 2013, Section 5).

7.1 Basic Insights

The following observation states how the most-precise inductive $\mathcal{A}$-invariant can be computed:

Observation 7.1. Let program P consist of (i) nodes $N = \{n_i\}$ with enter node n_1, (ii) edges $E_P = \{n_i \to n_j\}$, and (iii) a concrete-state transformer $\tau_{i,j}$ associated with each edge $n_i \to n_j$. Let $\mathcal{A}$ be an abstract domain. The **best inductive invariant** (BII) for P that is expressible in $\mathcal{A}$ is the least fixed-point of the equation system

$$V_1 = a_1 \qquad V_j = \bigsqcup_{n_i \to n_j \in E_P} \widehat{\mathrm{Post}}[\tau_{i,j}](V_i), \tag{7.1}$$

where a_1 is the best value in $\mathcal{A}$ that over-approximates the set of allowed input states at the enter node n_1. ("Best" means best with respect to $\mathcal{A}$.)

As a corollary of Observation 7.1, we have

Observation 7.2. When the least solution to Equation (7.1) can be obtained algorithmically, e.g., by Kleene iteration in an abstract domain with no infinite ascending chains, the BII problem reduces to the problem of applying $\widehat{\mathrm{Post}}$.

The precision obtained from a solution to BII depends on the set of "observation points" (or, equivalently the equation system being solved). For instance, in Example 3.6, the strawman solution does compute the *best* inductive $\mathcal{A}$-invariant if the intermediate point between τ_1 and τ_2 is observable. Example 3.6 shows that, from the standpoint of precision, the fewer observation points, the better. As with many methods in automatic program analysis and verification, our method normally requires that each loop be cut by at least one observation point. Thus, the set of loop headers would be a natural choice for the set of observation points. The take-away from this discussion is that it can be desirable to have a procedure that is capable of applying $\widehat{\mathrm{Post}}$ for an arbitrary loop-free sequence of instructions.

Equation (7.1) has a conventional form, but is non-standard because it uses the application of the most-precise abstract transformer $\widehat{\mathrm{Post}}[\tau_{i,j}]$, whereas most work on abstract interpretation

uses less-precise abstract transformers. For instance, some systems only support the state transformations of a restricted modeling language—e.g., affine programs (Cousot and Halbwachs, 1978; Müller-Olm and Seidl, 2004) or Boolean programs (Ball and Rajamani, 2000). Transformations outside the modeling language are over-approximated very coarsely. For example, for an assignment statement "x := e" in the original program, if e is an expression that uses any non-modeled operator, the statement is modeled as "x := ?" or, equivalently, "havoc(x)". (That is, after "x := e" executes, x can hold any value.) This translation from a program to the program-modeling language already involves some loss of precision.

In contrast, the application of $\widehat{\text{Post}}[\tau_{i,j}]$ in Equation (7.1) always incorporates the full concrete semantics of $\tau_{i,j}$ (i.e., without an a priori abstraction step).

7.2 Best Inductive Invariants and Intraprocedural Anaysis

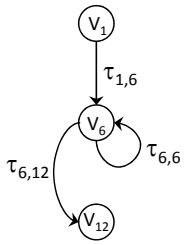
Figure 7.1(a) shows an example program that illustrates finding the best inductive invariant when the abstract domain is the affine-equalities domain $\mathcal{E}_{2^{32}}$. We concentrate on lines 1, 6, and 12. Figure 7.1(b) depicts the dependences in the equation system over node-variables $\{V_1, V_6, V_{12}\}$. Figure 7.1(c) gives formulas for the transition relations among $\{V_1, V_6, V_{12}\}$. The remainder of this section illustrates how to solve the BII problem for the following equation system, which corresponds to Figures 7.1(b) and 7.1(c):

$$
\begin{aligned}
V_1 &= \top \\
V_6 &= \widehat{\text{Post}}[\tau_{1,6}](V_1) \sqcup \widehat{\text{Post}}[\tau_{6,6}](V_6) \\
V_{12} &= \widehat{\text{Post}}[\tau_{6,12}](V_6)
\end{aligned}
$$

(a)

```
(1)  // Initialize
(2)  a = read_input();
(3)  b = a;
(4)  x = 0;
(5)  y = 0;
(6)  while (*) {   // Loop invariant: a==b && x==y
(7)      a = a+2;
(8)      b = (x==y) ? b+2 : read_input();
(9)      x = x+1;
(10)     y = (a==b) ? y+1 : read_input();
(11) }
(12) . . .  // Exit invariant: a==b && x==y
```

(b)



(c)

$$\tau_{1,6} \stackrel{\text{def}}{=} b' = a' \wedge x' = 0 \wedge y' = 0$$

$$\tau_{6,6} \stackrel{\text{def}}{=} \left(\begin{array}{l} a' = a + 2 \\ \wedge\ (x = y) \Rightarrow (b' = b + 2) \\ \wedge\ x' = x + 1 \\ \wedge\ (a' = b') \Rightarrow (y' = y + 1) \end{array} \right)$$

$$\tau_{6,12} \stackrel{\text{def}}{=} a' = a \wedge b' = b \wedge x' = x \wedge y' = y$$

Figure 7.1: (a) Example program. (b) Dependences among node-variables in the program's equation system (over node-variables $\{V_1, V_6, V_{12}\}$). (c) The transition relations among $\{V_1, V_6, V_{12}\}$ (expressed as formulas).

It is convenient to rewrite these equations as

$$\begin{aligned} V_1 &= \top \\ V_6 &= V_6 \sqcup \widehat{\mathrm{Post}}[\tau_{1,6}](V_1) \sqcup \widehat{\mathrm{Post}}[\tau_{6,6}](V_6) \\ V_{12} &= V_{12} \sqcup \widehat{\mathrm{Post}}[\tau_{6,12}](V_6) \end{aligned} \tag{7.2}$$

Solving the BII Problem from Below. In the most basic approach to solving the BII problem, we assume that we have an essentially standard fixed-point solver that performs chaotic iteration. The method will create successively better under-approximations to the solution, until it finds a fixed point, which will also be the best inductive invariant. We illustrate the algorithm on Equation (7.2).[1]

[1] We write abstract values in Courier typeface (e.g., $[\mathrm{a} = \mathrm{b}, \mathrm{x} = 0, \mathrm{y} = 0]$ or $[\mathrm{a}' = \mathrm{b}', \mathrm{x}' = 0, \mathrm{y}' = 0]$ are pre-state and post-state abstract values, respectively); concrete state-pairs in Roman typeface (e.g., $\left[a \mapsto 42, b \mapsto 27, x \mapsto 5, y \mapsto 19, a' \mapsto 17, b' \mapsto 17, x' \mapsto 0, y' \mapsto 0\right]$); and approximations to BII solutions as mappings from node-variables to abstract values (e.g., $\langle \mathrm{V}_1 \mapsto \top, \mathrm{V}_6 \mapsto [\mathrm{a} = \mathrm{b}, \mathrm{x} = 0, \mathrm{y} = 0], \mathrm{V}_{12} \mapsto [\mathrm{a} = 28, \mathrm{b} = 28, \mathrm{x} = 35, \mathrm{y} = 35] \rangle$).

What is special, compared to standard equation solvers, is that each application of the right-hand side of an equation in Equation (7.2)—defined by the corresponding formula in Figure 7.1(c)—is given the *best-transformer interpretation* by means of a function $\widehat{\mathrm{Post}}$ for applying the best abstract transformer. That is, $\widehat{\mathrm{Post}}$ satisfies $\widehat{\mathrm{Post}}[\tau](a) = (\alpha \circ \mathrm{Post}[\tau] \circ \gamma)(a)$. A specific instance of $\widehat{\mathrm{Post}}$ is the function $\widehat{\mathrm{Post}}^{\uparrow}$, given as Algorithm 8.

Figure 7.2 shows a possible chaotic-iteration sequence when a BII solver is invoked to find the best inductive affine-equality invariant for Equation (7.2), namely,

$$\langle \mathrm{V}_1 \mapsto \top, \mathrm{V}_6 \mapsto [\mathrm{a} = \mathrm{b}, \mathrm{x} = \mathrm{y}], \mathrm{V}_{12} \mapsto [\mathrm{a} = \mathrm{b}, \mathrm{x} = \mathrm{y}] \rangle.$$

Note that this abstract value corresponds exactly to the loop-invariant and exit-invariant shown in the comments on lines 6 and 12 of Figure 7.1(a). Moreover, the same abstract value would be arrived at no matter what sequence of choices is made during chaotic iteration to find the least fixed-point of Equation (7.2).

One such sequence is depicted in Figure 7.2, where three chaotic-iteration steps are performed before the least fixed point is found. The three steps propagate information from V_1 to V_6; from V_6 to V_6; and from V_6 to V_{12}, respectively. (At this point, to discover that chaotic iteration has quiesced, the solver would have to do some additional work, which we have not shown because it does not provide any additional insight on how BII problems are solved.)

The Value of a Bilateral Algorithm. $\widehat{\mathrm{Post}}^{\uparrow}$ is not resilient to timeouts. A query to the SMT solver—or the cumulative time for $\widehat{\mathrm{Post}}^{\uparrow}$—might take too long, in which case the only answer that is safe for $\widehat{\mathrm{Post}}^{\uparrow}$ to return is $\top$ (line 5 of Algorithm 8). To remedy this situation, we use a *bilateral* algorithm for $\widehat{\mathrm{Post}}$. Algorithm 15 (Chapter 5) shows our bilateral algorithm for $\widehat{\mathrm{Post}}$, called $\widehat{\mathrm{Post}}^{\updownarrow}$.

Figure 7.3 shows a possible trace of Iteration 2 from Figure 7.2 when the call to $\widehat{\mathrm{Post}}^{\uparrow}$ (Algorithm 8) is replaced by a call to $\widehat{\mathrm{Post}}^{\updownarrow}$ (Algorithm 15). Note how a collection of individual, non-trivial, upper-bounding constraints are acquired on the second, third, and fifth calls to `AbstractConsequence`: $[\mathrm{x}' = 1]$, $[\mathrm{y}' = 1]$, and $[\mathrm{a}' = \mathrm{b}']$, respectively. By these means, *upper′* works

Initialization: $\text{ans} := \langle \mathrm{V}_1 \mapsto \top, \mathrm{V}_6 \mapsto \bot, \mathrm{V}_{12} \mapsto \bot \rangle$

Iteration 1:
$$\begin{array}{rcl} V_6 & := & V_6 \sqcup \widehat{\mathrm{Post}}^{\uparrow}[\tau_{1,6}](V_1) \\ & = & \bot \sqcup \widehat{\mathrm{Post}}^{\uparrow}[\tau_{1,6}](\top) \end{array}$$

$$\begin{array}{rcl} \mathit{lower}' & := & \bot \\ \langle S, S' \rangle & := & \texttt{Model}(\texttt{true} \wedge \tau_{1,6} \wedge \neg\widehat{\gamma}(\bot)) \\ & = & \begin{bmatrix} a \mapsto 42, b \mapsto 27, x \mapsto 5, y \mapsto 99, \\ a' \mapsto 17, b' \mapsto 17, x' \mapsto 0, y' \mapsto 0 \end{bmatrix} \quad \begin{array}{l} \text{// A satisfying concrete} \\ \text{// state-pair} \end{array} \\ \mathit{lower}' & := & \bot \sqcup [\mathrm{a}' = 17, \mathrm{b}' = 17, \mathrm{x}' = 0, \mathrm{y}' = 0] \\ \langle S, S' \rangle & := & \texttt{Model}(\texttt{true} \wedge \tau_{1,6} \wedge \neg\widehat{\gamma}([\mathrm{a}' = 17, \mathrm{b}' = 17, \mathrm{x}' = 0, \mathrm{y}' = 0])) \\ & = & \begin{bmatrix} a \mapsto 73, b \mapsto 2, x \mapsto 15, y \mapsto 19, \\ a' \mapsto 28, b' \mapsto 28, x' \mapsto 0, y' \mapsto 0 \end{bmatrix} \quad \begin{array}{l} \text{// A satisfying concrete} \\ \text{// state-pair} \end{array} \\ \mathit{lower}' & := & [\mathrm{a}' = 17, \mathrm{b}' = 17, \mathrm{x}' = 0, \mathrm{y}' = 0] \sqcup [\mathrm{a}' = 28, \mathrm{b}' = 28, \mathrm{x}' = 0, \mathrm{y}' = 0] \\ v' & := & [\mathrm{a}' = \mathrm{b}', \mathrm{x}' = 0, \mathrm{y}' = 0] \end{array}$$

$$\begin{array}{rcl} V_6 & := & \bot \sqcup [\mathrm{a} = \mathrm{b}, \mathrm{x} = 0, \mathrm{y} = 0] = [\mathrm{a} = \mathrm{b}, \mathrm{x} = 0, \mathrm{y} = 0] \\ \text{ans} & := & \langle \mathrm{V}_1 \mapsto \top, \mathrm{V}_6 \mapsto [\mathrm{a} = \mathrm{b}, \mathrm{x} = 0, \mathrm{y} = 0], \mathrm{V}_{12} \mapsto \bot \rangle \end{array}$$

Iteration 2:
$$\begin{array}{rcl} V_6 & := & V_6 \sqcup \widehat{\mathrm{Post}}^{\uparrow}[\tau_{6,6}](V_6) \\ & = & [\mathrm{a} = \mathrm{b}, \mathrm{x} = 0, \mathrm{y} = 0] \sqcup \widehat{\mathrm{Post}}^{\uparrow}[\tau_{6,6}]([\mathrm{a} = \mathrm{b}, \mathrm{x} = 0, \mathrm{y} = 0]) \end{array}$$

$$\begin{array}{rcl} \mathit{lower}' & := & \bot \\ \langle S, S' \rangle & := & \texttt{Model}(\widehat{\gamma}([\mathrm{a} = \mathrm{b}, \mathrm{x} = 0, \mathrm{y} = 0]) \wedge \tau_{6,6} \wedge \neg\widehat{\gamma}(\bot)) \\ & = & \begin{bmatrix} a \mapsto 56, b \mapsto 56, x \mapsto 0, y \mapsto 0, \\ a' \mapsto 58, b' \mapsto 58, x' \mapsto 1, y' \mapsto 1 \end{bmatrix} \quad \begin{array}{l} \text{// A satisfying concrete} \\ \text{// state-pair} \end{array} \\ \mathit{lower}' & := & \bot \sqcup [\mathrm{a}' = 58, \mathrm{b}' = 58, \mathrm{x}' = 1, \mathrm{y}' = 1] \\ \langle S, S' \rangle & := & \texttt{Model}(\widehat{\gamma}([\mathrm{a} = \mathrm{b}, \mathrm{x} = 0, \mathrm{y} = 0]) \wedge \tau_{6,6} \wedge \neg\widehat{\gamma}([\mathrm{a}' = 58, \mathrm{b}' = 58, \mathrm{x}' = 1, \mathrm{y}' = 1])) \\ & = & \begin{bmatrix} a \mapsto 16, b \mapsto 16, x \mapsto 0, y \mapsto 0, \\ a' \mapsto 18, b' \mapsto 18, x' \mapsto 1, y' \mapsto 1 \end{bmatrix} \quad \begin{array}{l} \text{// A satisfying concrete} \\ \text{// state-pair} \end{array} \\ \mathit{lower}' & := & [\mathrm{a}' = 58, \mathrm{b}' = 58, \mathrm{x}' = 1, \mathrm{y}' = 1] \sqcup [\mathrm{a}' = 18, \mathrm{b}' = 18, \mathrm{x}' = 1, \mathrm{y}' = 1] \\ v' & := & [\mathrm{a}' = \mathrm{b}', \mathrm{x}' = 1, \mathrm{y}' = 1] \end{array}$$

$$\begin{array}{rcl} V_6 & := & [\mathrm{a} = \mathrm{b}, \mathrm{x} = 0, \mathrm{y} = 0] \sqcup [\mathrm{a} = \mathrm{b}, \mathrm{x} = 1, \mathrm{y} = 1] = [\mathrm{a} = \mathrm{b}, \mathrm{x} = \mathrm{y}] \\ \text{ans} & := & \langle \mathrm{V}_1 \mapsto \top, \mathrm{V}_6 \mapsto [\mathrm{a} = \mathrm{b}, \mathrm{x} = \mathrm{y}], \mathrm{V}_{12} \mapsto \bot \rangle \end{array}$$

Iteration 3:
$$\begin{array}{rcl} V_{12} & := & V_{12} \sqcup \widehat{\mathrm{Post}}^{\uparrow}[\tau_{6,12}](V_6) \\ & = & \bot \sqcup \widehat{\mathrm{Post}}^{\uparrow}[\tau_{6,12}]([\mathrm{a} = \mathrm{b}, \mathrm{x} = \mathrm{y}]) \end{array}$$

$$\begin{array}{rcl} \mathit{lower}' & := & \bot \\ \langle S, S' \rangle & := & \texttt{Model}(\widehat{\gamma}([\mathrm{a} = \mathrm{b}, \mathrm{x} = \mathrm{y}]) \wedge \tau_{6,6} \wedge \neg\widehat{\gamma}(\bot)) \\ & = & \begin{bmatrix} a \mapsto 17, b \mapsto 17, x \mapsto 99, y \mapsto 99, \\ a' \mapsto 17, b' \mapsto 17, x' \mapsto 99, y' \mapsto 99 \end{bmatrix} \quad \begin{array}{l} \text{// A satisfying concrete} \\ \text{// state-pair} \end{array} \\ \mathit{lower}' & := & \bot \sqcup [\mathrm{a}' = 17, \mathrm{b}' = 17, \mathrm{x}' = 99, \mathrm{y}' = 99] \\ \langle S, S' \rangle & := & \texttt{Model}(\widehat{\gamma}([\mathrm{a} = \mathrm{b}, \mathrm{x} = \mathrm{y}]) \wedge \tau_{6,6} \wedge \neg\widehat{\gamma}([\mathrm{a}' = 17, \mathrm{b}' = 17, \mathrm{x}' = 99, \mathrm{y}' = 99])) \\ & = & \begin{bmatrix} a \mapsto 28, b \mapsto 28, x \mapsto 35, y \mapsto 35, \\ a' \mapsto 28, b' \mapsto 28, x' \mapsto 35, y' \mapsto 35 \end{bmatrix} \quad \begin{array}{l} \text{// A satisfying concrete} \\ \text{// state-pair} \end{array} \\ \mathit{lower}' & := & [\mathrm{a}' = 17, \mathrm{b}' = 17, \mathrm{x}' = 99, \mathrm{y}' = 99] \sqcup [\mathrm{a}' = 28, \mathrm{b}' = 28, \mathrm{x}' = 35, \mathrm{y}' = 35] \\ v' & := & [\mathrm{a}' = \mathrm{b}', \mathrm{x}' = \mathrm{y}'] \end{array}$$

$$\begin{array}{rcl} V_{12} & := & \bot \sqcup [\mathrm{a} = \mathrm{b}, \mathrm{x} = \mathrm{y}] = [\mathrm{a} = \mathrm{b}, \mathrm{x} = \mathrm{y}] \\ \text{ans} & := & \langle \mathrm{V}_1 \mapsto \top, \mathrm{V}_6 \mapsto [\mathrm{a} = \mathrm{b}, \mathrm{x} = \mathrm{y}], \mathrm{V}_{12} \mapsto [\mathrm{a} = \mathrm{b}, \mathrm{x} = \mathrm{y}] \rangle \end{array}$$

Fixed Point!

Figure 7.2: A possible chaotic-iteration sequence when a BII solver is invoked to find the best inductive affine-equality invariant for Equation (7.2). The parts of the trace enclosed in boxes show the actions that take place in calls to Algorithm 8 ($\widehat{\mathrm{Post}}^{\uparrow}$). (By convention, primes are dropped from the abstract value returned from a call on $\widehat{\mathrm{Post}}^{\uparrow}$.)

$$
\begin{array}{rcl}
V_6 & := & V_6 \sqcup \widehat{\mathrm{Post}}^{\updownarrow}[\tau_{6,6}](V_6) \\
 & = & [\mathrm{a} = \mathrm{b}, \mathrm{x} = 0, \mathrm{y} = 0] \sqcup \widehat{\mathrm{Post}}^{\updownarrow}[\tau_{6,6}]([\mathrm{a} = \mathrm{b}, \mathrm{x} = 0, \mathrm{y} = 0]) \\
\hline
\mathit{upper}' & := & \top \\
\mathit{lower}' & := & \bot \\
p' & := & \texttt{AbstractConsequence}(\bot, \top) \\
 & = & \bot \\
\langle S, S' \rangle & := & \texttt{Model}(\widehat{\gamma}([\mathrm{a} = \mathrm{b}, \mathrm{x} = 0, \mathrm{y} = 0]) \wedge \tau_{6,6} \wedge \neg\widehat{\gamma}(\bot)) \\
 & = & \begin{bmatrix} a \mapsto 56, b \mapsto 56, x \mapsto 0, y \mapsto 0, \\ a' \mapsto 58, b' \mapsto 58, x' \mapsto 1, y' \mapsto 1 \end{bmatrix} \quad \begin{array}{l} \text{// A satisfying concrete} \\ \text{// state-pair} \end{array} \\
\mathit{lower}' & := & \bot \sqcup [\mathrm{a}' = 58, \mathrm{b}' = 58, \mathrm{x}' = 1, \mathrm{y}' = 1] \\
p' & := & \texttt{AbstractConsequence}([\mathrm{a}' = 58, \mathrm{b}' = 58, \mathrm{x}' = 1, \mathrm{y}' = 1], \top) \\
 & = & [\mathrm{x}' = 1] \\
\langle S, S' \rangle & := & \texttt{Model}(\widehat{\gamma}([\mathrm{a} = \mathrm{b}, \mathrm{x} = 0, \mathrm{y} = 0]) \wedge \tau_{6,6} \wedge \neg\widehat{\gamma}([\mathrm{x}' = 1])) \\
 & = & \mathsf{None} \\
\mathit{upper}' & := & \top \sqcap [\mathrm{x}' = 1] = [\mathrm{x}' = 1] \\
p' & := & \texttt{AbstractConsequence}([\mathrm{a}' = 58, \mathrm{b}' = 58, \mathrm{x}' = 1, \mathrm{y}' = 1], [\mathrm{x}' = 1]) \\
 & = & [\mathrm{y}' = 1] \\
\langle S, S' \rangle & := & \texttt{Model}(\widehat{\gamma}([\mathrm{a} = \mathrm{b}, \mathrm{x} = 0, \mathrm{y} = 0]) \wedge \tau_{6,6} \wedge \neg\widehat{\gamma}([\mathrm{y}' = 1])) \\
 & = & \mathsf{None} \\
\mathit{upper}' & := & [\mathrm{x}' = 1] \sqcap [\mathrm{y}' = 1] = [\mathrm{x}' = 1, \mathrm{y}' = 1] \\
p' & := & \texttt{AbstractConsequence}([\mathrm{a}' = 58, \mathrm{b}' = 58, \mathrm{x}' = 1, \mathrm{y}' = 1], [\mathrm{x}' = 1, \mathrm{y}' = 1]) \\
 & = & [\mathrm{a}' = 58] \\
\langle S, S' \rangle & := & \texttt{Model}(\widehat{\gamma}([\mathrm{a} = \mathrm{b}, \mathrm{x} = 0, \mathrm{y} = 0]) \wedge \tau_{6,6} \wedge \neg\widehat{\gamma}([\mathrm{a}' = 58])) \\
 & = & \begin{bmatrix} a \mapsto 19, b \mapsto 19, x \mapsto 0, y \mapsto 0, \\ a' \mapsto 21, b' \mapsto 21, x' \mapsto 1, y' \mapsto 1 \end{bmatrix} \quad \begin{array}{l} \text{// A satisfying concrete} \\ \text{// state-pair} \end{array} \\
\mathit{lower}' & := & [\mathrm{a}' = 58, \mathrm{b}' = 58, \mathrm{x}' = 1, \mathrm{y}' = 1] \sqcup [\mathrm{a}' = 21, \mathrm{b}' = 21, \mathrm{x}' = 1, \mathrm{y}' = 1] \\
 & = & [\mathrm{a}' = \mathrm{b}', \mathrm{x}' = 1, \mathrm{y}' = 1] \\
p' & := & \texttt{AbstractConsequence}([\mathrm{a}' = \mathrm{b}', \mathrm{x}' = 1, \mathrm{y}' = 1], [\mathrm{x}' = 1, \mathrm{y}' = 1]) \\
 & = & [\mathrm{a}' = \mathrm{b}'] \\
\langle S, S' \rangle & := & \texttt{Model}(\widehat{\gamma}([\mathrm{a} = \mathrm{b}, \mathrm{x} = 0, \mathrm{y} = 0]) \wedge \tau_{6,6} \wedge \neg\widehat{\gamma}([\mathrm{a}' = \mathrm{b}'])) \\
 & = & \mathsf{None} \\
\mathit{upper}' & := & [\mathrm{x}' = 1, \mathrm{y}' = 1] \sqcap [\mathrm{a}' = \mathrm{b}'] \\
 & = & [\mathrm{a}' = \mathrm{b}', \mathrm{x}' = 1, \mathrm{y}' = 1] \\
\mathit{lower}' \neq \mathit{upper}' & = & \mathsf{false} \\
v' & := & [\mathrm{a}' = \mathrm{b}', \mathrm{x}' = 1, \mathrm{y}' = 1] \\
\hline
V_6 & := & [\mathrm{a} = \mathrm{b}, \mathrm{x} = 0, \mathrm{y} = 0] \sqcup [\mathrm{a} = \mathrm{b}, \mathrm{x} = 1, \mathrm{y} = 1] = [\mathrm{a} = \mathrm{b}, \mathrm{x} = \mathrm{y}] \\
\mathrm{ans} & := & \langle \mathrm{V}_1 \mapsto \top, \mathrm{V}_6 \mapsto [\mathrm{a} = \mathrm{b}, \mathrm{x} = \mathrm{y}], \mathrm{V}_{12} \mapsto \bot \rangle
\end{array}
$$

Figure 7.3: A possible trace of Iteration 2 from Figure 7.2 when the call to $\widehat{\mathrm{Post}}^{\uparrow}$ (Algorithm 8) is replaced by a call to $\widehat{\mathrm{Post}}^{\updownarrow}$ (Algorithm 15).

its way down the chain $\top \sqsupset [\mathrm{x}' = 1] \sqsupset [\mathrm{x}' = 1, \mathrm{y}' = 1] \sqsupset [\mathrm{a}' = \mathrm{b}', \mathrm{x}' = 1, \mathrm{y}' = 1]$. After each call to `AbstractConsequence`, the abstract-consequence constraint is tested to see if it really is an upper bound on the answer. For instance, the fourth call to `AbstractConsequence` returns $[\mathrm{a}' = 58]$.

The assertion that $[\mathrm{a}' = 58]$ is an upper-bounding constraint is refuted by the concrete state-pair

$$\langle S, S' \rangle = \begin{bmatrix} a \mapsto 19, b \mapsto 19, x \mapsto 0, y \mapsto 0, \\ a' \mapsto 21, b' \mapsto 21, x' \mapsto 1, y' \mapsto 1 \end{bmatrix},$$

which is used to improve the value of *lower*′.

The important point is that if Iteration 2 is taking too much time, $\widehat{\mathrm{Post}}^{\updownarrow}$ can be stopped and *upper*′ returned as the answer. In contrast, if $\widehat{\mathrm{Post}}^{\uparrow}$ is stopped because it is taking too much time, the only safe answer that can be returned is $\top$. The "can-be-stopped-anytime" property of $\widehat{\mathrm{Post}}^{\updownarrow}$ can make a significant difference in the final answer. For instance, suppose that $\widehat{\mathrm{Post}}^{\uparrow}$ and $\widehat{\mathrm{Post}}^{\updownarrow}$ both stop early during Iteration 2 (Figures 7.2 and 7.3, respectively), and that $\widehat{\mathrm{Post}}^{\updownarrow}$ returns $[\mathrm{x} = 1, \mathrm{y} = 1]$, whereas $\widehat{\mathrm{Post}}^{\uparrow}$ returns $\top$. Assuming no further timeouts take place during the evaluation of Equation (7.2), the respective final answers would be

$$\widehat{\mathrm{Post}}^{\uparrow} : \langle \mathrm{V}_1 \mapsto \top, \mathrm{V}_6 \mapsto \top, \mathrm{V}_{12} \mapsto \top \rangle$$
$$\widehat{\mathrm{Post}}^{\updownarrow} : \langle \mathrm{V}_1 \mapsto \top, \mathrm{V}_6 \mapsto [\mathrm{x} = \mathrm{y}], \mathrm{V}_{12} \mapsto [\mathrm{x} = \mathrm{y}] \rangle$$

Because of the timeout, the answer computed by $\widehat{\mathrm{Post}}^{\updownarrow}$ is not the *best* inductive affine-equality invariant; however, the answer establishes that the equality constraint $[\mathrm{x} = \mathrm{y}]$ holds at both lines 6 and 12 of Figure 7.1(a).

Attaining the Best Inductive $\mathcal{A}$-Invariant

Lemma 7.3. *The least fixed-point of Equation* (7.1) *(the best $\mathcal{A}$-transformer equations of a transition system) is the best inductive invariant expressible in $\mathcal{A}$.*

Corollary 7.4. *Applying an equation solver to the best $\mathcal{A}$-transformer equations, using either $\widehat{\mathrm{Post}}^{\uparrow}$ or $\widehat{\mathrm{Post}}^{\updownarrow}$ to evaluate equation right-hand sides, finds the best inductive $\mathcal{A}$-invariant if there are no timeouts during the evaluation of any right-hand side.*

7.3 Best Inductive Invariants and Interprocedural Analysis

This section presents a method for solving the BII problem for multi-procedure programs. Our framework is similar to the so-called "functional approach" to interprocedural analysis of Sharir and Pnueli (1981) (denoted by *SP*), which works with an abstract domain that abstracts transition relations. For instance, our approach

- also uses an abstract domain that abstracts transition relations, and
- creates a *summary transformer* for each reachable procedure P, which over-approximates the transition relation of P.

However, to make the symbolic-abstraction approach suitable for interprocedural analysis, the algorithm uses a generalization of $\widehat{\text{Post}}$, called $\widehat{\text{Compose}}[\tau](a)$, where $\tau \in \mathcal{L}[\overrightarrow{v}; \overrightarrow{v}']$ and $a \in \mathcal{A}[\overrightarrow{v}; \overrightarrow{v}']$ both represent transition relations over the program variables $\overrightarrow{v}$. The goal of $\widehat{\text{Compose}}[\tau](a)$ is to create an $\mathcal{A}[\overrightarrow{v}; \overrightarrow{v}']$ value that is the best over-approximation of a's action followed by τ's action. Furthermore, instead of Equation (7.1), the least solution to Equations (7.3)–(7.7) below is found, where each application of the right-hand side of an equation is given the *best-transformer interpretation*—in this case, by means of $\widehat{\text{Compose}}$.

A *program* is defined by a set of procedures P_i, $0 \leq i \leq K$, and represented by an interprocedural control-flow graph $G = (N, F)$. G consists of a collection of intraprocedural control-flow graphs $G_1, G_2, \ldots, G_K$, one of which, $G_{\textit{main}}$, represents the program's main procedure. The node set N_i of $G_i = (N_i, F_i)$ is partitioned into five disjoint subsets: $N_i = E_i \uplus X_i \uplus C_i \uplus R_i \uplus L_i$. G_i contains exactly one *enter* node (i.e., $E_i = \{e_i\}$) and exactly one *exit* node (i.e., $X_i = \{x_i\}$). A procedure call in G_i is represented by two nodes, a *call* node $c \in C_i$ and a *return-site* node $r \in R_i$, and has two edges: (i) a *call-to-enter* edge from c to the enter node of the called procedure, and (ii) an *exit-to-return-site* edge from the exit node of the called procedure to r. The functions *call* and *ret* record matching call and return-site nodes: $\textit{call}(r) = c$ and $\textit{ret}(c) = r$. It is assumed that an enter node has no incoming edges except call-to-enter edges.

$$\phi(e_{\mathit{main}}) = Id|_{a_1} \qquad a_1 \in \mathcal{A} \text{ describes the set of initial stores at } e_{\mathit{main}} \tag{7.3}$$

$$\phi(e_p) = Id|_a \qquad e_p \in E,\ p \neq \mathit{main}, \text{ and } a = \bigsqcup_{c \in C,\, c \text{ calls } p} V_c \tag{7.4}$$

$$\phi(n) = \bigsqcup_{m \to n \in F} \widehat{\text{Compose}}[\tau_{m,n}](\phi(m)) \quad \text{for } n \in N,\ n \notin (R \cup E) \tag{7.5}$$

$$\phi(n) = \widehat{\text{Compose}}[\widehat{\gamma}(\phi(x_q))](\phi(\mathit{call}(n))) \quad \text{for } n \in R, \text{ and } \mathit{call}(n) \text{ calls } q \tag{7.6}$$

$$V_n = \text{range}(\phi(n)) \tag{7.7}$$

The equations involve two sets of "variables": $\phi(n)$ and V_n, where $n \in N$. $\phi(n)$ is a partial function that represents a summary of the transformation from $e_{\mathit{proc}(n)}$ to n. $Id|_a$ denotes the identity transformer restricted to inputs in $a \in \mathcal{A}$. The domain of $\phi(n)$ over-approximates the set of reachable states at $e_{\mathit{proc}(n)}$ from which it is possible to reach n; the range of $\phi(n)$ over-approximates the set of reachable states at n. V_n's value equals the range of $\phi(n)$.

$\widehat{\text{Compose}}^{\updownarrow}$ is essentially identical to Algorithm 15, except that in line 5, $\widehat{\text{Compose}}^{\updownarrow}$ performs a query using a three-state formula,

$$\langle S, S', S'' \rangle \leftarrow \texttt{Model}(\widehat{\gamma}_{[\vec{v},\vec{v}']}(a) \wedge \tau_{[\vec{v}',\vec{v}'']} \wedge \neg \widehat{\gamma}_{[\vec{v},\vec{v}'']}(p')),$$

and in line 11, $\widehat{\text{Compose}}^{\updownarrow}$ applies a two-state version of β to S and S'', dropping S' completely: $\mathit{lower}' \leftarrow \mathit{lower}' \sqcup \beta(S, S'')$. ($\widehat{\text{Compose}}^{\uparrow}$ is defined similarly.)

An important difference between our algorithm and the *SP* algorithm is that in our algorithm, the initial abstract value for the enter node e_p for procedure p is specialized to the reachable inputs of p (see Equation (7.4)). In the *SP* algorithm, $\phi(e_p)$ is always set to *Id*. Figure 7.4 illustrates the effect on the inferred abstract post-condition at x_p of specializing the abstract pre-condition at enter node e_p. The abstract domain used in Figure 7.4 is the domain of affine equalities over machine integers $\mathcal{E}_{2^{32}}$. With that domain, it is possible to express that a 32-bit variable x holds an even number: $2^{31}x = 0$. Consequently, the initial abstract value for enter node e_{halve} is the

```
(1)  main() {                          (8)  void halve() {
(2)      x = read_input();             (9)    x = x ≫ 1;
(3)      while (x != 1) {              (10) }
(4)          if (even(x)) halve();     (11)
(5)          else increase();          (12) void increase() {
(6)      }                             (13)   x = 3*x + 1;
(7)  }                                 (14) }
```

		Abstract value at enter node e_p	
		Id	$Id\|_a$, where $a = \bigsqcup_{c \in C,\, c \text{ calls } p} V_c$
halve	pre-condition (e_{halve})	$x' = x$	$2^{31}x = 0 \wedge x' = x$
	post-condition (x_{halve})	$\top$	$2^{31}x = 0 \wedge x - 2x' = 0$
increase	pre-condition (e_{increase})	$x' = x$	$2^{31}x = 1 \wedge x' = x$
	post-condition (x_{increase})	$x' = 3x + 1$	$2^{31}x = 1 \wedge 2^{31}x' = 0 \wedge x' = 3x + 1$

Figure 7.4: The effect of specializing the abstract pre-condition at enter node e_p, and the resulting strengthening of the inferred abstract post-condition. (The abstract domain is the domain of affine equalities $\mathcal{E}_{2^{32}}$.)

identity relation, constrained so that x is even. Similarly, the initial abstract value for enter node e_{increase} is the identity relation, constrained so that x is odd.

Note that the abstract value at the exit point x_p of a procedure p serves as a procedure summary—i.e., an abstraction of p's transition relation. Figure 7.4 shows that by giving halve and increase more precise abstract values at the respective enter nodes, more precise procedure summaries at the respective exit points are obtained. In particular, for halve, the constraint $x - 2x' = 0$ provides a good characterization of the effect of a right-shift operation, but only when x is known to be even (cf. the entries for halve's post-condition in columns 3 and 4 of Figure 7.4).

7.4 Related Work

Houdini (Flanagan and Leino, 2001) is the first algorithm that I am aware of that solves a version of the BII problem. The paper on Houdini does not describe the work in terms of abstract interpretation. Santini (Thakur et al., 2013, Section 5) was directly inspired by Houdini, as an effort to broaden Houdini's range of applicability.

Yorsh et al. (2006) introduced a particularly interesting technique. Their algorithm for the BII problem can be viewed as basically solving Equation (7.1) using $\widehat{\mathrm{Post}}^{\uparrow}$. However, they observed that it is not necessary to rely on calls to an SMT solver for *all* of the concrete states used by $\widehat{\mathrm{Post}}^{\uparrow}$; instead, they used concrete execution of the program as a way to generate concrete states very cheaply. If for some program point q of interest they have state-set S_q, they obtain an under-approximation for the abstract value V_q by performing $V_q = \sqcup\{\beta(\sigma) \mid \sigma \in S_q\}$. This idea is similar in spirit to the computation of candidate invariants from execution information by Daikon (Ernst et al., 2007). Because $\widehat{\mathrm{Post}}^{\updownarrow}$ works simultaneously from below and from above, the Yorsh et al. heuristic can be used to improve the speed of convergence of *lower'* in line 11 of Algorithm 15.

If we think of $\tau = \langle \ldots, \tau_{i,j}, \ldots \rangle$ as a monolithic transformer, an alternative way of stating the objective of intraprocedural BII is as follows:

- Given a concrete transformer τ and an abstract value $a \in \mathcal{A}$ that over-approximates the set of initial states, apply the best abstract transformer for τ^* to a (i.e., apply $\widehat{\mathrm{Post}}[\tau^*](a)$).

This problem was the subject of a recent technical report by Garoche et al. (2012).

7.5 Chapter Notes

Extending the results of "best abstract transformers" to "best inductive invariant" appeared daunting at first. Some amount of head scratching resulted in the simple observation that the BII problem reduces to the problem of applying $\widehat{\mathrm{Post}}$.

The Santini tool, which incorporates the ideas described in this chapter, was developed by Akash Lal of Microsoft Research. Because I had little to do with the implementation of this tool and the running of the experiments, I have omitted these experimental results from this thesis. The curious reader is encouraged to read about Santini and its application in the Corral verification tool in Thakur et al. (2013).

Chapter 8

Bit-Vector Inequality Domain

In this chapter, I describe how symbolic abstraction enables us to define a new abstract domain, called the *Bit-Vector Inequality* ($\mathcal{BVI}$) domain, that addresses the following challenges: (1) identifying affine-inequality invariants while handling overflow in arithmetic operations over bit-vector data-types, and (2) holding onto invariants about values in memory during machine-code analysis.

Need for bit-vector invariants. The polyhedral domain (Cousot and Halbwachs, 1978) is capable of expressing relational affine inequalities over rational (or real) variables. However, the native machine-integer data-types used in programs (e.g., `int`, `unsigned int`, `long`, etc.) perform bit-vector arithmetic, and arithmetic operations wrap around on overflow. Thus, the underlying point space used in the polyhedral domain does not faithfully model bit-vector arithmetic, and consequently the conclusions drawn from an analysis based on the polyhedral domain are unsound, unless special steps are taken (Simon and King, 2007).

Example 8.1. The following C-program fragment incorrectly computes the average of two `int`-valued variables (Bloch, 2014):

```
unsigned int low, high, mid;
assume(0 <= low <= high);
mid = (low + high)/2;
assert(0 <= low <= mid <= high);
```

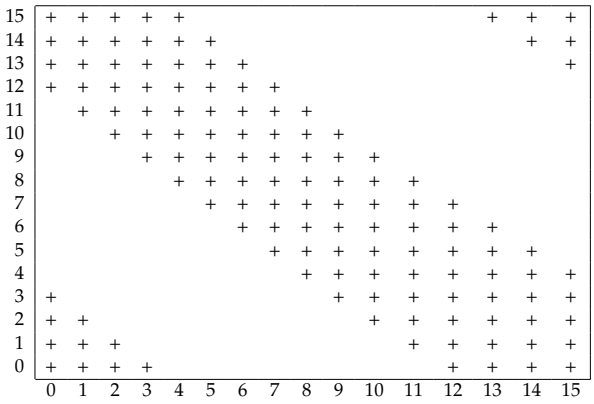
A static analysis based on polyhedra would draw the unsound conclusion that the assertion always holds. In particular, assuming 32-bit `ints`, when the sum of low and high is greater than $2^{32} - 1$, the sum overflows to a negative value, and the resulting value of mid is negative. Consequently, there exist runs in which the assertion fails. These runs are overlooked when the polyhedral domain is used for static analysis because the domain fails to take into account the bit-vector semantics of program variables. ■

The problem that we wish to solve is *not* one of merely *detecting* overflow—e.g., to restrain an analyzer from having to explore what happens after an overflow occurs. On the contrary, our goal is to be able to track soundly the effects of arithmetic operations, including wrap-around effects of operations that overflow. This ability is useful, for instance, when analyzing code generated by production code generators, such as dSPACE TargetLink (dSPACE, 2014), which use the "compute-through-overflow" technique (Garner, 1978). Furthermore, clever idioms for bit-twiddling operations, such as the ones explained in Warren (2003), sometimes rely on overflow.

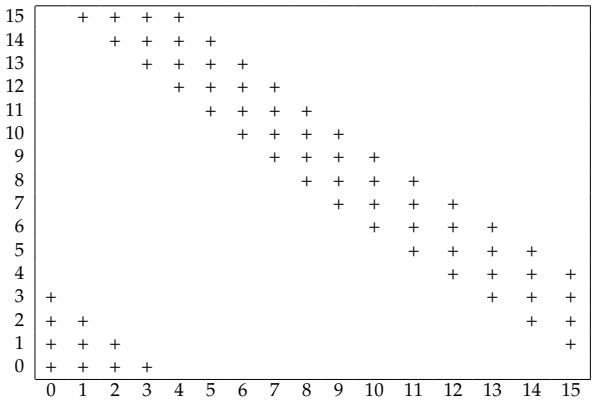
Challenges in dealing with bit-vectors. The ideas used in designing an inequality domain for reals do not carry over to one designed for bit-vectors. First, in bit-vector arithmetic, additive constants cannot be cancelled on both sides of an inequality, as illustrated in the following example.

Example 8.2. Let x and y be 4-bit unsigned integers. Figures 8.1(a) and 8.1(b) depict the solutions in bit-vector arithmetic of the inequalities $x + y + 4 \leq 7$ and $x + y \leq 3$, respectively. Although $x+y+4 \leq 7$ and $x+y \leq 3$ are syntactically quite similar, their solution spaces are quite different. In particular, because of wrap-around of values computed on the left-hand sides using bit-vector arithmetic, one cannot just subtract 4 from both sides to convert the inequality $x + y + 4 \leq 7$ into $x + y \leq 3$. ■

Second, in bit-vector arithmetic, positive constant factors cannot be cancelled on both sides of an inequality; for example, if x and y are 4-bit bit-vectors, then $(4, 4)$ is in the solution set of $2x + 2y \leq 4$, but not of $x + y \leq 2$.



(a) $x + y + 4 \leq 7$



(b) $x + y \leq 3$

Figure 8.1: Each $+$ represents a solution of the indicated inequality in 4-bit unsigned bit-vector arithmetic.

Recent work has developed several abstract domains of relational affine *equalities* over variables that hold machine integers (Müller-Olm and Seidl, 2005; King and Søndergaard, 2010; Elder et al., 2011).[1] These domains *do* account for wrap-around on overflow. With respect to their analysis capabilities, the drawback of these domains is that they are unable to identify *inequality* invariants.

While some simple domains do exist that are capable of representing certain kinds of inequalities over bit-vectors (e.g., intervals with a congruence constraint, sometimes called "strided-intervals" (Reps et al., 2006; Sen and Srikant, 2007; Balakrishnan and Reps, 2010)), such domains

[1]The bit-vector affine-equalities domain $\mathcal{E}_{2^w}$ has been used in this thesis in Sections 4.3, 5.5, and 6.4.1.

are non-relational domains. That is, they are not capable of expressing relations among several variables.

Challenges when dealing with memory. When analyzing machine-code, memory is usually modeled as a flat array. When analyzing Intel x86 machine code, for instance, memory is modeled as a map from 32-bit bit-vectors to 8-bit bit-vectors. Consequently, an analysis has to deal with complications arising from the little-endian addressing mode and aliasing, as illustrated in the next example.

Example 8.3. Consider the following machine-code snippet:

```
mov eax, [ebp]
mov [ebp+2], ebx
```

The first instruction loads the four bytes pointed to by register `ebp` into the 32-bit register `eax`. Suppose that the value in register `ebp` is A. After the first instruction, the bytes of `eax` contain, in least-significant to most-significant order, the value at memory location A, the value at location $A + 1$, the value at location $A + 2$, and the value at location $A + 3$. The second instruction stores the value in register ebx into the memory pointed to by ebp+2. Due to this instruction, the values at memory locations $A + 2$ through $A + 5$ are overwritten, after which the value in register eax no longer equals (the little-endian interpretation of) the bytes in memory pointed to by ebp. ■

The contribution of this chapter can be summarized as follows:

- I describe the design and implementation of the $\mathcal{BVI}$ abstract domain, which is capable of expressing affine-inequality invariants among bit-vector data-types, as well as handling invariants about values in memory during machine-code analysis.

Section 8.2 presents related work. The presentation of an experimental evaluation of the $\mathcal{BVI}$ domain is postponed until Chapter 9, where it can be found in Section 9.4

8.1 The $\mathcal{BVI}$ Abstract Domain

In this section, I present the intuition and formalism for the design and implementation of the $\mathcal{BVI}$ abstract domain.

Abstract-Domain Vocabulary. For a given family of abstract domains $\mathcal{A}$, $\mathcal{A}[V]$ denotes the specific instance of $\mathcal{A}$ that is defined over vocabulary V. Usually, the standard vocabulary used to define an abstract domain is the set of program variables. For instance, in Section 5.5 we used the abstract domain $\mathcal{E}_{2^{32}}[R]$, where R was the set of all 32-bit x86 registers; that is, $\mathcal{E}_{2^{32}}[R]$ is the bit-vector affine-equalities domain $\mathcal{E}_{2^{32}}$ defined over the vocabulary of 32-bit x86 registers. Consequently, $\mathcal{E}_{2^{32}}[R]$ is capable of expressing invariants among members of R.

Key Insights. The key insights behind the design of the $\mathcal{BVI}$ are:

- We extend the standard vocabulary of an abstract domain with *view expressions*, which are w-bit terms expressed in some logic $\mathcal{L}$. Thus, view expressions are capable of holding onto richer constraints about the program state than the unenriched abstract domain alone.
- We construct the $\mathcal{BVI}$ domain as a reduced product between the bit-vector affine-equality domain $\mathcal{E}_{2^w}$ and the bit-vector interval domain $\mathcal{I}_{2^w}$.
- We use symbolic abstraction to implement precise versions of the abstract-domain operations for $\mathcal{BVI}$.

Cousot et al. (2011b) use the term *observables* for view expressions. They also define the *observational reduced product*, which, in essence, is a reduced product of abstract domains those vocabularies have been extended with observables.

I will illustrate each of the concepts listed in the key insights above using examples.

Example 8.4. This example illustrates how *view expressions* enable the abstract domain $\mathcal{I}_{2^w}$ to express inequality constraints, and constraints over memory.

Let $P \equiv \{x, y, z\}$ be the standard vocabulary consisting of 32-bit program variables x, y, and z. Let $t_1 \equiv 2x + 3y$ be a 32-bit term *view expression*. Let $T \equiv P \cup \{t_1\}$ be the standard vocabulary P extended with the view expression t_1.

An element of $\mathcal{I}_{2^{32}}[P]$ is able to only express a non-relational constraint between x, y, and z. On the other hand, the abstract value $I_1 \equiv [t_1 \mapsto [0,5]] \in \mathcal{I}_{2^{32}}[T]$ expresses the relational bit-vector inequality constraint $0 \leq 2x + 3y \wedge 2x + 3y \leq 5$.

To express invariants on values in memory (for machine-code analysis), we can add a view expression that holds onto the values in memory. I use $[e]$ to denote the 32-bit value at the address e accessed using little-endian addressing. Let $m_1 \equiv [x + 2]$, $m_2 \equiv ([y] + 2z)$, and $M \equiv P \cup \{t_1, m_1, m_2\}$. The abstract domain $\mathcal{I}_{2^{32}}[M]$ is capable of expressing bit-vector affine-inequality constraints, as well as constraints on the values in memory. For instance, the element $I_2 \equiv [t_1 \mapsto [0,5], m_1 \mapsto [1,10], m_2 \mapsto [42,56]] \in \mathcal{I}_{2^{32}}[M]$ expresses the following relational inequality constraint over program variables and the contents of memory:

$$\begin{aligned} & 0 \leq 2x + 3y \wedge 2x + 3y \leq 5 \\ \wedge\ & 1 \leq [x+2] \wedge [x+2] \leq 10 \\ \wedge\ & 42 \leq [y] + 2z \wedge [y] + 2z \leq 56 \end{aligned}$$

■

Example 8.5. In this example, I illustrate how computing a reduced product of $\mathcal{E}_{2^w}$ and $\mathcal{I}_{2^w}$ improves precision.

Consider the abstract value $I_1 \equiv [t_1 \mapsto [0,5]] \in \mathcal{I}_{2^{32}}[T]$ defined in Example 8.4. An abstract value in $\mathcal{E}_{2^{32}}[T]$ is capable to expressing any bit-vector affine-equality relation among elements of T. Constructing the reduced product $\mathcal{E}_{2^{32}}[T] \star \mathcal{I}_{2^{32}}[T]$ expands the set of inequalities that are expressible. For instance, let $E_1 \equiv \{2x = z\} \in \mathcal{E}_{2^{32}}[T]$. The pair $\langle E_1, I_1 \rangle$ expresses the following constraint: $2x = z \wedge 0 \leq 2x + 3y \wedge 2x + 3y \leq 5$. Furthermore, $\langle E_1, I_1 \rangle$ implies the inequality $0 \leq z + 3y \wedge z + 3y \leq 5$, which cannot be expressed in the individual domains $\mathcal{E}_{2^{32}}[T]$ and $\mathcal{I}_{2^{32}}[T]$. ■

The previous two examples paint a rather rosy picture—in particular, it appears as if all one needs to do is add view expressions to be able to handle bit-vector inequality constraints. As the next example illustrates, adding view expressions is only half the story; we require machinery that is capable of interpreting the semantics of the view expressions.

Example 8.6. Consider the abstract domain $\mathcal{E}_{2^{32}}[T]$, where the vocabulary T is defined in Example 8.4. Let $E_1 \equiv \{t_1 = 8\} \in \mathcal{E}_{2^{32}}[T]$, and $E_2 \equiv \{x = 2, y = 3\} \in \mathcal{E}_{2^{32}}[T]$. We now want to compute $E_1 \sqcap E_2$. If we compute the meet of E_1 and E_2 using the meet algorithm described in Elder et al. (2011), we get $\{x = 2, y = 3, t_1 = 8\}$. However, though sound, this meet operation ignores the semantics of the view expression t_1. In particular, $\gamma(\{x = 2, y = 3, t_1 = 8\}) = [\![x = 2 \wedge y = 3 \wedge 2x + 3y = 8]\!] = \emptyset$. Consequently, the semantic reduction (Definition 3.7) of the meet of E_1 and E_2 should be $\bot$; that is, $\rho(E_1 \sqcap E_2) = \bot$.

This simple example illustrates the fact that existing approaches for performing abstract operations, such as meet and join, are not capable of interpreting the complex view expressions.

Furthermore, if a view expression holds onto values in memory, then the abstract operations need to reason about the theory of arrays (Bradley et al., 2006) to be precise. ■

The solution to implementing precise abstract operations, such as meet and join, is to use symbolic abstraction. For instance, in Example 8.6 we can compute the semantic reduction of $\{x = 2, y = 3, t_1 = 8\} \in \mathcal{E}_{2^{32}}[T]$, which is $\bot$, using symbolic abstraction (Section 3.2.3). Furthermore, symbolic abstraction is also used to compute the reduced product of $\mathcal{E}_{2^w}$ and $\mathcal{I}_{2^w}$. Thus, the "heavy-lifting" of interpreting the meaning of the view expressions is done by the algorithm for performing symbolic abstraction (Chapter 5).

Building on the intuition from the previous examples, we can define the $\mathcal{BVI}$ as follows:

Definition 8.7. Let P be the set of w-bit program variables. A *view expression* t is a w-bit term expressed over P in logic $\mathcal{L}$.

Given a vocabulary T consisting of a set of view expressions and program variables, the abstract domain $\mathcal{BVI}_{2^w}$ is defined as the reduced product $\mathcal{E}_{2^w}[T] \star \mathcal{I}_{2^w}[T]$.

Let $A_1 \equiv \langle E_1, I_1 \rangle, A_2 \equiv \langle E_2, I_2 \rangle \in \mathcal{BVI}_{2^w}$. The approximate abstract operations for $\mathcal{BVI}_2 w$, which do not interpret the meaning of the view expressions, are defined as:

$$A_1 \widetilde{\sqcap} A_2 \stackrel{\text{def}}{=} \langle E_1 \sqcap_{\mathcal{E}_{2^w}} E_2, I_1 \sqcap_{\mathcal{I}_{2^w}} I_2 \rangle$$

$$A_1 \widetilde{\sqcup} A_2 \stackrel{\text{def}}{=} \langle E_1 \sqcup_{\mathcal{E}_{2^w}} E_2, I_1 \sqcup_{\mathcal{I}_{2^w}} I_2 \rangle$$

$$A_1 \widetilde{\sqsubseteq} A_2 \stackrel{\text{def}}{=} E_1 \sqsubseteq_{\mathcal{E}_{2^w}} E_2 \text{ and } I_1 \sqsubseteq_{\mathcal{I}_{2^w}} I_2$$

The more precise abstract operations, which do interpret the meaning of the view expressions, are defined as:

$$\widehat{\gamma}(A_1) \stackrel{\text{def}}{=} \widehat{\gamma}_{\mathcal{E}_{2^w}}(E_1) \wedge \widehat{\gamma}_{\mathcal{I}_{2^w}}(I_1)$$

$$A_1 \widehat{\sqcap} A_2 \stackrel{\text{def}}{=} \widehat{\alpha}(\widehat{\gamma}(A_1) \wedge \widehat{\gamma}(A_2))$$

$$A_1 \widehat{\sqcup} A_2 \stackrel{\text{def}}{=} \widehat{\alpha}(\widehat{\gamma}(A_1) \vee \widehat{\gamma}(A_2))$$

$$A_1 \widehat{\sqsubseteq} A_2 \stackrel{\text{def}}{=} \widehat{\gamma}(A_1) \text{ implies } \widehat{\gamma}(A_2)$$

■

8.1.1 Implementation Details

Implementing Symbolic Abstraction. We use an instantiation of the Bilateral framework (Chapter 5) to compute the symbolic abstraction of the $\mathcal{BVI}$ abstract domain. Note that the Bilateral algorithm uses the approximate abstract operations $\widetilde{\sqcap}$, $\widetilde{\sqcup}$, and $\widetilde{\sqsubseteq}$ in order to compute the symbolic abstraction.

Lazy Symbolic Abstraction. In the implementation of the $\widehat{\sqcup}$ and $\widehat{\sqcap}$ operations of the $\mathcal{BVI}$ domain, the computation of the symbolic abstraction can be delayed until it becomes necessary. In essence, internal to the implementation of the $\mathcal{BVI}$ domain, the $\widehat{\sqcap}$ and $\widehat{\sqcup}$ operations are treated as logical-and and logical-or operations, respectively. When an answer is required to be an abstract value, a call to $\widehat{\alpha}$ is used to convert the formula into the abstract value. I call this technique of delaying the call to symbolic abstraction *Lazy Symbolic Abstraction*. For instance, suppose we have to compute $(A_1 \widehat{\sqcap} A_2) \widehat{\sqcap} A_3$, where $A_1, A_2, A_3 \in \mathcal{BVI}$. Using Definition 8.7, we have that

$$(A_1 \widehat{\sqcap} A_2) \widehat{\sqcap} A_3 \equiv \widehat{\alpha}(\widehat{\gamma}(\widehat{\alpha}(\widehat{\gamma}(A_1) \wedge \widehat{\gamma}(A_2))) \wedge \widehat{\gamma}(A_3)) \tag{8.1}$$

Instead, we could compute it as follows:

$$(A_1 \widehat{\sqcap} A_2) \widehat{\sqcap} A_3 \equiv \widehat{\alpha}(\widehat{\gamma}(A_1) \wedge \widehat{\gamma}(A_2) \wedge \widehat{\gamma}(A_3)) \tag{8.2}$$

Equation (8.2) avoids the call to $\widehat{\alpha}$ in the computation of $A_1 \widehat{\sqcap} A_2$, and remains in the logical

domain. The call to $\widehat{\alpha}$ is delayed until the end of the computation of the meets. There are two advantages of delaying the call to $\widehat{\alpha}$ as shown in Equation (8.2):

- Precision: Equation (8.2) avoids going back to the (less precise) abstract domain, and stays in the logical domain.
- Speed: Equation (8.2) has a single call to $\widehat{\alpha}$.

This same approach of delaying the call to $\widehat{\alpha}$ applies to the implementation of $\widehat{\sqcup}$. In practice, we perform an $\widehat{\alpha}$ operation that converts the internal formula representation into an abstract value when either of the following conditions are met:

- The number of conjunctions and disjunctions in the formula exceed a fixed threshold k.
- The join operation is performed at a loop header, or, more generally, at a *widening point* (Bourdoncle, 1993).

Heuristics for picking view expressions. When performing machine-code analysis, the choice of view expressions determines the precision of the abstract domain. In practice, the view expressions include memory accesses performed by the program. We also determine the *branch predicates* occurring in the program, and add these as view expressions. The branch predicate is computing by first performing symbolic execution of the basic block that ends in a branch instruction, and then inspecting the expression for the program counter in the symbolic state. Given a (Boolean) branch predicate b, we add the view expression $ITE(b, 1_{2^{32}}, 0_{2^{32}})$, where ITE is the if-then-else term. This enables the $\mathcal{BVI}$ abstract domain to behave similarly to a Cartesian predicate-abstraction domain (Flanagan and Qadeer, 2002), in that the $\mathcal{BVI}$ is capable of expressing conjunctions of predicates. However, the use the affine-equalities domain allows the $\mathcal{BVI}$ domain to also express a logical-xor of predicates, hence, making the domain more expressive than a Cartesian predicate-abstraction domain. This concept is illustrated in the next example.

Example 8.8. Consider the view expressions $t_1 \equiv ITE(b_1, 1_{2^{32}}, 0_{2^{32}})$, and $t_2 \equiv ITE(b_2, 1_{2^{32}}, 0_{2^{32}})$, and let $T \equiv \{t_1, t_2\}$.

$$a_1 \equiv \{t_1 = 1, t_2 = 0\} \in \mathcal{BVI}_{2^{32}}[T] \qquad \widehat{\gamma}(a_1) = b_1 \wedge \neg b_2$$

$$a_2 \equiv \{t_1 + t_2 = 2\} \in \mathcal{BVI}_{2^{32}}[T] \qquad \widehat{\gamma}(a_2) = b_1 \wedge b_2$$

$$a_1 \equiv \{t_1 + t_2 = 1\} \in \mathcal{BVI}_{2^{32}}[T] \qquad \widehat{\gamma}(a_3) = b_1 \oplus b_2$$

■

Picking the right set of view expressions is an interesting and challenging research question in itself. Exploring techniques based on, for example, counter-example guided abstraction refinement (CEGAR) is left as future work. Section 9.4 describes an experimental evaluation of the $\mathcal{BVI}$ abstract domain.

8.2 Related Work

Other work on identifying bit-vector-inequality invariants includes Brauer and King (2010, 2011) and Masdupuy (1992). Masdupuy (1992) proposed a relational abstract domain of interval congruences on rationals. One limitation of his machinery is that the domain represents diagonal grids of parallelepipeds, where the dimension of each parallelepiped equals the number of variables tracked (say n). In our work, we can have any number of view-variables, which means that the point-spaces represented can be constrained by more than n constraints.

Brauer and King employ bit-blasting to synthesize abstract transformers for the interval and octagon (Miné, 2001) domains. One of their papers uses universal-quantifier elimination on Boolean formulas (Brauer and King, 2010); the other avoids quantifier elimination (Brauer and King, 2011). Compared with their work, we avoid the use of bit-blasting and work directly with representations of sets of w-bit bit-vectors.

Chang and Leino (2005) developed a technique for extending the properties representable by a given abstract domain from schemas over variables to schemas over terms. To orchestrate the communication of information between domains, they designed the congruence-closure abstract domain, which introduces variables to stand for subexpressions that are alien to a base domain; to

the base domain, these expressions are just variables. Their scheme for propagating information between domains is mediated by the e-graph of the congruence-closure domain. In contrast, our method can make use of symbolic abstraction to propagate information between domains. Cousot et al. (2011b) have recently studied the iterated pairwise exchange of observations between components as a way to compute an over-approximation of a reduced product.

Chen et al. (2008) devised a way to use the polyhedral domain (Cousot and Halbwachs, 1978) to analyze programs that use floating-point computations. They use a linearization technique developed by Miné (2004), which soundly abstracts the floating-point computations performed in a program being analyzed into computations on reals. These values are then over-approximated using a variant of the polyhedral domain in which two of the underlying primitives are replaced by floating-point versions that use interval arithmetic with outward rounding to compute answers in floating-point arithmetic that over-approximate the exact real counterparts. Both of the replaced primitives can produce a floating-point overflow or the value NaN (Not a Number). In these cases, a sound answer is created by discarding a constraint. By this means, it is possible to use the floating-point polyhedral domain to analyze programs that use floating-point computations.

8.3 Chapter Notes

Tushar Sharma helped with part of the implementation and experimental evaluation of the $\mathcal{BVI}$ domain; I was responsible for idea behind of the $\mathcal{BVI}$ domain, and most of the design and implementation of the domain.

Chapter 9

Symbolic Abstraction for Machine-Code Verification

This chapter describes a model-checking algorithm for stripped machine-code, called McVeto (**M**achine-**C**ode **VE**rification **TO**ol). The unique challenges and opportunities in verification of stripped machine-code, as well as a comparison of McVeto with previous machine-code analysis tools, can be found in Chapter 2. In particular, McVeto is able to detect and explore "deviant behavior" in machine code. An example of such deviant behavior is when the program over-writes the return address stored on the stack frame, as illustrated in Section 2.3. In McVeto, deviant behavior is reported as an *acceptable-execution (AE) violation*. Moreover, McVeto is capable of verifying (or detecting flaws in) self-modifying code (SMC). To the best of my knowledge, McVeto is the first model checker to handle SMC.

The specific problem addressed by McVeto can be stated as follows: Given a stripped machine-code program P, and a (bad) target state, find either

- an input to P that results in an AE violation,
- an input to P that causes the target state to be reached, or
- a proof that P performs only AEs, and the bad state cannot be reached during any AE.

The (bad) target state is usually specified as a particular value of the program counter (PC), which we denote by *target*.

MCVETO works by starting with an initial coarse abstraction of the program's state space. The initial abstraction has only two abstract states, defined by the predicates "PC $=$ *target*" and "PC $\neq$ *target*" . The abstraction is gradually refined during subsequent analysis steps. Thus, the MCVETO algorithm can be viewed as computing the symbolic abstraction of a machine-code program with respect to a particular graph-based abstract domain.

The contributions of this chapter can be summarized as follows:

1. I describe how MCVETO adapts *directed proof generation* (DPG) (Gulavani et al., 2006) for model checking stripped machine code (Section 9.1).

2. I describe how MCVETO uses *trace-based generalization* to build and refine an abstraction of the program's state space entirely on-the-fly (Section 9.2.2). Trace-based generalization enables MCVETO to handle instruction aliasing and SMC.

3. I describe how MCVETO uses *speculative trace refinement* to identify *candidate invariants* that can speed up the convergence of DPG (Section 9.2.3).

4. I introduce a new approach to performing DPG on multi-procedure programs (Section 9.2.4).

5. I describe a language-independent algorithm to identify the aliasing condition relevant to a property in a given state (Section 9.2.5).

Item 2 addresses execution details that are typically ignored (unsoundly) by source-code analyzers. Items 2, 3, 4, and 5 are applicable to both source-code and machine-code analysis.

Section 9.3 describes the implementation details of MCVETO; Section 9.4 presents an experimental evaluation of MCVETO; Section 9.5 presents related work.

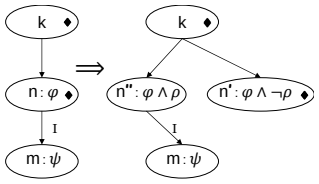


Figure 9.1: The general refinement step across frontier (n, I, m). The presence of a witness is indicated by a "♦" inside of a node.

9.1 Background on Directed Proof Generation (DPG)

Given a program P and a particular control location *target* in P, DPG returns either an input for which execution leads to *target* or a proof that *target* is unreachable (or DPG does not terminate). Two approximations of P's state space are maintained:

- A set T of concrete traces, obtained by running P with specific inputs. T *under*approximates P's state space.
- A graph G, called the *abstract graph*, obtained from P via abstraction (and abstraction refinement). G *over*approximates P's state space.

Nodes in G are labeled with formulas; edges are labeled with program instructions or program conditions. One node is the *start node* (where execution begins); another node is the *target node* (the goal to reach). Information to relate the under- and overapproximations is also maintained: a concrete state σ in a trace in T is called a *witness* for a node n in G if σ satisfies the formula that labels n.

If G has no path from *start* to *target*, then DPG has proved that *target* is unreachable, and G serves as the proof. Otherwise, DPG locates a *frontier*: a triple (n, I, m), where (n, m) is an edge on a path from *start* to *target* such that n has a witness w but m does not, and I is the instruction on (n, m). DPG either performs concrete execution (attempting to reach *target*) or refines G by splitting nodes and removing certain edges (which may prove that *target* is unreachable). Which action to perform is determined using the basic step from *directed test generation* (Godefroid et al., 2005), which uses symbolic execution to try to find an input that allows execution to

cross frontier (n, I, m). Symbolic execution is performed over symbolic states, which have two components: a *path constraint*, which represents a constraint on the input state, and a *symbolic map*, which represents the current state in terms of input-state quantities. DPG performs symbolic execution along the path taken during the concrete execution that produced witness w for n; it then symbolically executes I, and conjoins to the path constraint the formula obtained by evaluating m's predicate ψ with respect to the symbolic map. It calls an SMT solver to determine if the path constraint obtained in this way is satisfiable. If so, the result is a satisfying assignment that is used to add a new execution trace to T. If not, DPG refines G by splitting node n into n' and n'', as shown in Figure 9.1.

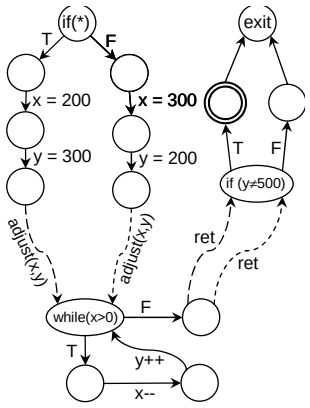
Refinement changes G to represent some *non-connectivity* information: in particular, n' is not connected to m in the refined graph (see Figure 9.1). Let ψ be the formula that labels m, c be the concrete witness of n, and S_n be the symbolic state obtained from the symbolic execution up to n. DPG chooses a formula ρ, called the *refinement predicate*, and splits node n into n' and n'' to distinguish the cases when n is reached with a concrete state that satisfies ρ (n'') and when it is reached with a state that satisfies $\neg\rho$ (n'). The predicate ρ is chosen such that (i) no state that satisfies $\neg\rho$ can lead to a state that satisfies ψ after the execution of I, and (ii) the symbolic state S_n satisfies $\neg\rho$. Condition (i) ensures that the edge from n' to m can be removed. Condition (ii) prohibits extending the current path along I (forcing the DPG search to explore different paths). It also ensures that c is a witness for n' and not for n'' (because c satisfies S_n)—and thus the frontier during the next iteration must be different.

9.2 McVeto

This section explains the methods used to achieve contributions 2–5. While McVeto was designed to provide sound DPG for machine code, a number of its novel features are also useful for source-code DPG. Thus, to make the chapter more accessible, our running example is the C++ program in Figure 9.2. It makes a non-deterministic choice between two blocks that each call procedure `adjust`, which loops—decrementing `x` and incrementing `y`. Note that the affine

```
int main() {
 int x, y;
 if(*) {
  x = 200;
  y = 300;
  adjust(x, y);
 }
 else {
  x = 300;
  y = 200;
  adjust(x, y);
 }
 if (y != 500)
  UNREACHABLE();
 return 0;
}
```

```
void adjust(int& x, int& y) {
 while(x > 0) {
  x--;
  y++;
 }
}
```



(a) (b)

Figure 9.2: (a) A program with a non-deterministic branch; (b) the program's ICFG.

relation $x + y = 500$ holds at the two calls on `adjust`, the loop-head in `adjust`, and the branch on `y!=500`.

9.2.1 Representing the Abstract Graph

The infinite abstract graph used in McVeto is finitely represented as a nested word automaton (NWA) (Alur and Madhusudan, 2009) and queried by symbolic operations.

Definition 9.1 (Alur and Madhusudan (2009))**.** A **nested word** $(w, \rightsquigarrow)$ over alphabet Σ is an ordinary word $w \in \Sigma^*$, together with a **nesting relation** $\rightsquigarrow$ of length $|w|$. $\rightsquigarrow$ is a collection of edges (over the positions in w) that do not cross. A nesting relation of length $l \geq 0$ is a subset of $\{-\infty, 1, 2, \ldots, l\} \times \{1, 2, \ldots, l, +\infty\}$ such that

- Nesting edges only go forwards: if $i \rightsquigarrow j$ then $i < j$.
- No two edges share a position: for $1 \leq i \leq l$, $|\{j \mid i \rightsquigarrow j\}| \leq 1$ and $|\{j \mid j \rightsquigarrow i\}| \leq 1$.
- Edges do not cross: if $i \rightsquigarrow j$ and $i' \rightsquigarrow j'$, then one cannot have $i < i' \leq j < j'$.

When $i \rightsquigarrow j$ holds, for $1 \leq i \leq l$, i is called a **call** position; if $i \rightsquigarrow +\infty$, then i is a **pending call**; otherwise i is a **matched call**, and the unique position j such that $i \rightsquigarrow j$ is called its **return**

successor. Similarly, when $i \rightsquigarrow j$ holds, for $1 \leq j \leq l$, j is a **return** position; if $-\infty \rightsquigarrow j$, then j is a **pending return**, otherwise j is a **matched return**, and the unique position i such that $i \rightsquigarrow j$ is called its **call predecessor**. A position $1 \leq i \leq l$ that is neither a call nor a return is an **internal** position.

MatchedNW denotes the set of nested words that have no pending calls or returns. **NWPrefix** denotes the set of nested words that have no pending returns.

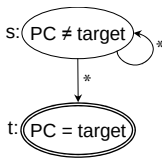
A **nested word automaton** (NWA) A is a 5-tuple $(Q, \Sigma, q_0, \delta, F)$, where Q is a finite set of states, Σ is a finite alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, and δ is a transition relation. The transition relation δ consists of three components, $(\delta_c, \delta_i, \delta_r)$, where

- $\delta_i \subseteq Q \times \Sigma \times Q$ is the transition relation for internal positions.
- $\delta_c \subseteq Q \times \Sigma \times Q$ is the transition relation for call positions.
- $\delta_r \subseteq Q \times Q \times \Sigma \times Q$ is the transition relation for return positions.

Starting from q_0, an NWA A reads a nested word $nw = (w, \rightsquigarrow)$ from left to right, and performs transitions (possibly non-deterministically) according to the input symbol and $\rightsquigarrow$. If A is in state q when reading input symbol σ at position i in w, and i is an internal or call position, A makes a transition to q' using $(q, \sigma, q') \in \delta_i$ or $(q, \sigma, q') \in \delta_c$, respectively. If i is a return position, let k be the call predecessor of i, and q_c be the state A was in just before the transition it made on the k^{th} symbol; A uses $(q, q_c, \sigma, q') \in \delta_r$ to make a transition to q'. If, after reading nw, A is in a state $q \in F$, then A **accepts** nw. ■

As discussed in Section 9.2.2 the key property of NWAs for abstraction refinement is that, even though they represent matched call/return structure, they are closed under intersection (Alur and Madhusudan, 2009). That is, given NWAs A_1 and A_2, one can construct an NWA A_3 such that $L(A_3) = L(A_1) \cap L(A_2)$.

In our NWAs, the alphabet consists of all possible machine-code instructions. In addition, we annotate each state with a predicate. Operations on NWAs extend cleanly to accommodate the semantics of predicates—e.g., the $\cap$ operation labels a product state $\langle q_1, q_2 \rangle$ with the conjunction



(a)

$$\delta_c = \left\{ \begin{matrix} (s, *, s), \\ (s, *, t) \end{matrix} \right\}$$

$$\delta_r = \left\{ \begin{matrix} (s, s, *, s), \\ (s, s, *, t) \end{matrix} \right\}$$

(b)

Figure 9.3: (a) Internal-transitions in the initial NWA-based abstract graph G_0 created by McVeto; (b) call- and return-transitions in G_0. * is a wild-card symbol that matches all instructions.

of the predicates on states q_1 and q_2. In McVeto's abstract graph, we treat the value of the PC as data; consequently, predicates can refer to the value of the PC (see Figure 9.3).

9.2.2 Abstraction Refinement Via Trace Generalization

In a source-code model checker, the initial overapproximation of a program's state space is often the program's ICFG. Unfortunately, for machine code it is difficult to create an accurate ICFG *a priori* because of the use of indirect jumps, jump tables, and indirect function calls—as well as more esoteric features, such as instruction aliasing and SMC. For this reason, McVeto begins with the degenerate NWA-based abstract graph G_0 shown in Figure 9.3, which overapproximates the program's state space; i.e., G_o accepts an overapproximation of the set of minimal[1] traces that reach *target*. The abstract graph is refined during the state-space exploration carried out by McVeto.

To avoid having to disassemble collections of nested branches, loops, procedures, or the whole program all at once, McVeto performs *trace-based disassembly*: as concrete traces are generated during DPG, instructions are disassembled one at a time by decoding the current bytes of memory starting at the value of the PC. Each indirect jump or indirect call encountered can be resolved to a specific address. Trace-based disassembly is one of the techniques that allows McVeto to handle self-modifying code.

McVeto uses each concrete trace $\pi \in T$ to refine abstract graph G. As mentioned in Section 9.1, the set T of concrete traces *under*approximates the program's state space, whereas G

[1] A trace τ that reaches *target* is *minimal* if τ does not have a proper prefix that reaches *target*.

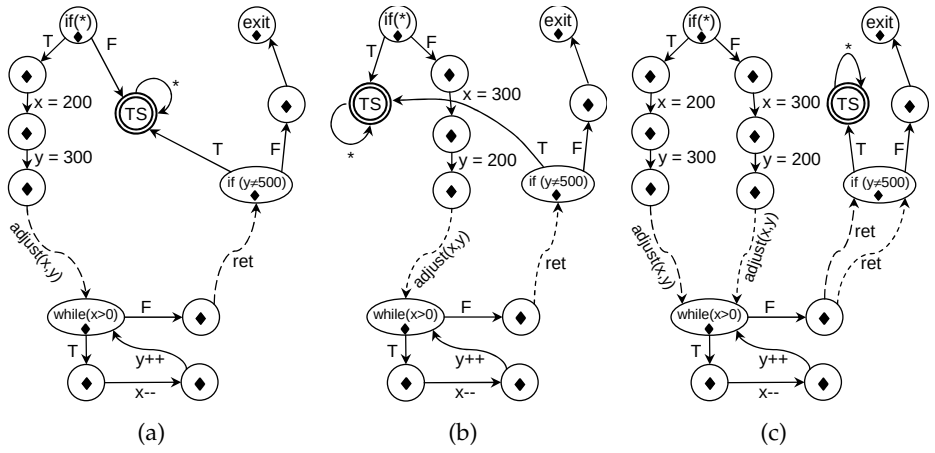


Figure 9.4: (a) and (b) show two generalized traces, each of which reaches the end of the program. (c) shows the intersection of the two generalized traces. ("♦" indicates that a node has a witness.)

represents an *over*approximation of the state space. McVeto repeatedly solves instances of the following *trace-generalization* problem:

Given a trace π, which is an *under*approximation of the program, convert π into an NWA-based abstract graph G_π that is an *over*approximation of the program.

A trace π that does not reach *target* is represented by (i) a nested-word prefix $(w, \rightsquigarrow)$ over instructions (Definition 9.1), together with (ii) an array of PC values, PC$[1..|w|+1]$, where PC$[|w|+1]$ has the special value HALT if the trace terminated execution. **Internal-steps**, **call-steps**, and **return-steps** are triples of the form $\langle \text{PC}[i], w[i], \text{PC}[i+1] \rangle$, $1 \leq i < |w|$, depending on whether i is an internal-position, call-position, or return-position, respectively. We create G_π by "folding" π—grouping together all nodes with the same PC value, and augmenting it in a way that overapproximates the portion of the program not explored by π (denoted by $\pi/[\text{PC}]$); see Figures 9.4(a), 9.4(b), and 9.5. In particular, G_π contains one accepting state, called *TS* (for "target surrogate"). *TS* is an accepting state because it represents *target*, as well as all non-*target* locations not visited by π.

We now have two overapproximations, the original abstract graph G and folded trace G_π. Abstract graphs are based on NWAs, and hence closed under intersection. Thus, by performing $G := G \cap G_\pi$, information about the portion of the program explored by π is incorporated into G, producing a third, improved overapproximation; see Figure 9.4(c). Equivalently, intersection elim-

Definition 9.2. Given π, we construct $G_\pi \stackrel{\text{def}}{=} \pi/[\text{PC}]$ as follows:

1. All positions $1 \leq k < |w| + 1$ for which PC$[k]$ has a given address a are collapsed to a single NWA state q_a. All such states are rejecting states (the target was not reached).
2. For each internal-step $\langle a, I, b\rangle$, G_π has an internal-transition (q_a, I, q_b).
3. For each call-step $\langle a_c, \texttt{call}, a_e\rangle$, G_π has a call-transition $(q_{a_c}, \texttt{call}, q_{a_e})$. ("`call`" stands for whatever instruction instance was used in the call-step.)
4. For each return-step $\langle a_x, \texttt{ret}, a_r\rangle$ for which the PC at the call predecessor holds address a_c, G_π has a return-transition $(q_{a_x}, q_{a_c}, \texttt{ret}, q_{a_r})$. ("`ret`" stands for whatever instruction instance was used in the return-step.)
5. G_π contains one accepting state, called *TS* (for "target surrogate"). *TS* is an accepting state because it represents *target*, as well as all the non-target locations that π did not explore.
6. G_π contains three "self-loops": $(TS, *, TS) \in \delta_i$, $(TS, *, TS) \in \delta_c$, and $(TS, TS, *, TS) \in \delta_r$. (We use "*" in the latter two transitions because there are many forms of `call` and `ret` instructions.)
7. For each unmatched instance of a call-step $\langle a_c, \texttt{call}, a_e\rangle$, G_π has a return-transition $(TS, q_{a_c}, *, TS)$. (We use * because any kind of `ret` instruction could appear in the matching return-step.)
8. Let B_b denote a (direct or indirect) branch that takes branch-direction b.
   - If π has an internal-step $\langle a, B_b, c\rangle$ but not an internal-step $\langle a, B_{\neg b}, d\rangle$, G_π has an internal-transition $(q_a, B_{\neg b}, TS)$.
   - For each internal-step $\langle a, B_T, c\rangle$, where B is an indirect branch, G_π has an internal-transition (q_a, B_T, TS).
9. For each call-step $\langle a_c, \texttt{call}, a_e\rangle$ where `call` is an indirect call, G_π has a call-transition $(q_{a_c}, \texttt{call}, TS)$.
10. If PC$[|w|+1] \neq$ HALT, G_π has an internal-transition $(q_{\text{PC}[|w|]}, I, TS)$, where "$I$" stands for whatever instruction instance was used in step $|w|$ of π. (We assume that an uncompleted trace never stops just before a `call` or `ret`.)
11. If PC$[|w| + 1] =$ HALT, G_π has an internal-transition $(q_{\text{PC}[|w|]}, I, \text{Exit})$, where "$I$" stands for whatever instruction instance was used in step $|w|$ of π and Exit is a distinguished non-accepting state.

Figure 9.5: Definition of the trace-folding operation $\pi/[\text{PC}]$.

Algorithm 19: Basic MCVETO algorithm (including trace-based disassembly)

```
1  π := nested-word prefix for an execution run on a random initial state
2  T := {π}; G_π := π/[PC]; G := (NWA from Figure 9.3) ∩ G_π
3  while true do
4    if target has a witness in T then
5      return "reachable"
6    Find a path τ in G from start to target
7    if no path exists then
8      return "not reachable"
9    Find a frontier (n, I, m) in G, where concrete state σ witnesses n
10   Perform symbolic execution of the instructions of the concrete trace that reaches σ, and
     then of instruction I; conjoin to the path constraint the formula obtained by evaluating m's
     predicate ψ with respect to the symbolic map; let S be the path constraint so obtained
11   if S is feasible, with satisfying assignment A then
12     π := nested-word prefix for an execution run on A
13     T := T ∪ {π}; G_π := π/[PC]; G := G ∩ G_π
14   else
15     Refine G along frontier (n, I, m) (see Figure 9.1)
```

inates the family of infeasible traces represented by the complement of G_π; however, because we already have G_π in hand, no automaton-complementation operation is required—cf. (Heizmann et al., 2010).

The issue of how one forms an NWPrefix from an instruction sequence—i.e., identifying the nesting structure—is handled by a policy in the trace-recovery tool for classifying each position as an internal-, call-, or return-position. Currently, for reasons discussed in Section 9.2.6, we use the following policy: the position of any form of `call` instruction is a call-position; the position of any form of `ret` instruction is a return-position. In essence, MCVETO uses `call` and `ret` instructions to restrict the instruction sequences considered. If these match the program's actual instruction sequences, we obtain the benefits of the NWA-based approach—especially the reuse of information among refinements of a given procedure. The basic MCVETO algorithm is stated as Algorithm 19.

Trace Generalization for Self-Modifying Code

To perform trace generalization for self-modifying code, state names are now of the form $q_{a,I}$, where a is an address and I is an instruction. Item 1 of the trace folding operation (Figure 9.5) is changed to

- All positions $1 \leq k < |w| + 1$ for which (i) PC$[k]$ has a given address a, and (ii) $w[k]$ has a given instruction I are collapsed to a single NWA state $q_{a,I}$. All such states are rejecting states (the target was not reached).

Internal-, call-, and return-steps are now quadruples, $\langle \text{PC}[i], w[i], \text{PC}[i+1], w[i+1] \rangle$ depending on whether i, for $1 \leq i < |w|$, is an internal-, call-, or return-position, respectively. Other items are changed accordingly to account for instructions in state names. In addition, items 8–10 are replaced by

- For each position i, $1 \leq i < |w| + 1$, G_π contains
  - an internal-transition $(q_{\text{PC}[i],w[i]}, *, TS)$
  - a call-transition $(q_{\text{PC}[i],w[i]}, *, TS)$
  - a return-transition $(q_{\text{PC}[i],w[i]}, q_{\text{PC}[j],w[j]}, *, TS)$, where $1 \leq j < i$ and $w[j]$ is the unmatched `call` with largest index in $w[1..i-1]$.

9.2.3 Speculative Trace Refinement

Motivated by the observation that DPG is able to avoid exhaustive loop unrolling if it discovers the right loop invariant, we developed mechanisms to discover candidate invariants from a folded trace, which are then incorporated into the abstract graph via NWA intersection. Although they are only *candidate* invariants, they are introduced into the abstract graph in the hope that they are invariants for the full program. The basic idea is to apply dataflow analysis to a graph obtained from the folded trace G_π. The recovery of invariants from G_π is similar in spirit to the computation of invariants from traces in Daikon (Ernst et al., 2007), but in McVeto they are computed *ex post*

facto by dataflow analysis on the folded trace. While any kind of dataflow analysis could be used in this fashion, McVeto currently uses the $\mathcal{BVI}$ abstract domain (Chapter 8). Specifically, the candidate invariant ψ at program point p is

$$\psi \equiv \widehat{\gamma}\left(\bigsqcup\{\beta(S) \mid S \text{ is a concrete state at } p\}\right)$$

The candidate invariants are used to create predicates for the nodes of G_π. Because an analysis may not account for the full effects of indirect memory references on the inferred variables, to incorporate a discovered candidate invariant φ for node n into G_π safely, we split n on φ and $\neg\varphi$. Again we have two overapproximations: G_π, from the folded trace, augmented with the candidate invariants, and the original abstract graph G. To incorporate the candidate invariants into G, we perform $G := G \cap G_\pi$; the $\cap$ operation labels a product state $\langle q_1, q_2\rangle$ with the conjunction of the predicates on states q_1 of G and q_2 of G_π.

Figure 9.6 shows how the candidate affine relation $\varphi \equiv x + y = 500$ would be introduced at the loop-head of `adjust` in the generalized traces from Figures 9.4(a) and 9.4(b). (Relation φ does, in fact, hold for the portions of the state space explored by Figures 9.4(a) and 9.4(b).) With this enhancement, subsequent steps of DPG will be able to show that the dotted loop-heads (labeled with $\neg\varphi$) can never be reached from *start*. In addition, the predicate φ on the solid loop-heads enables DPG to avoid exhaustive loop unrolling to show that the true branch of `y!=500` can never be taken.

9.2.4 Symbolic Methods for Interprocedural DPG

In other DPG systems (Gulavani et al., 2006; Beckman et al., 2008; Godefroid et al., 2010), *inter*procedural DPG is performed by invoking *intra*procedural DPG as a subroutine. In contrast, McVeto analyzes a representation of the entire program (refined on-the-fly), which allows it to reuse all information from previous refinement steps. For instance, in the program shown in Figure 9.7(a), procedure `lotsaBaz` makes several calls to `baz`. By invoking analysis once for each call site on `baz`, a tool such as Dash has to re-learn that `y` is set to 0. In contrast, McVeto

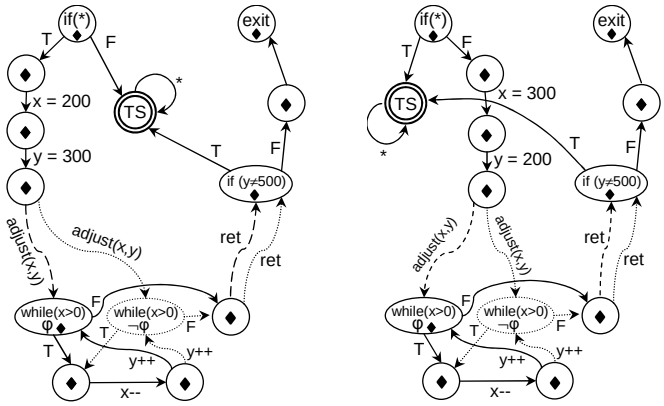


Figure 9.6: Figures 9.4(a) and 9.4(b) with the loop-head in `adjust` split with respect to the candidate invariant $\varphi \stackrel{\text{def}}{=} x + y = 500$.

```
int y;
void baz(){
  y=0;
  y++;
  y--;
}
```

```
void lotsaBaz(int a){
  y=0;
  if(a>0) baz();
  if(a>1) baz();
  if(a>2) baz();
  if(a>3) baz();
  if(y!=0)
    ERR: return;
}
```

(a)

```
int bar1() {
  int i,r = 0;
  for(i=0;i<100;i++){
    complicated(); r++;
  }
  return r;
}

int bar2(){ return 10; }
```

```
void foo(int x){
  int y;
  if(x == 0) y = bar2();
  else y = bar1();
  if(y == 10)
    ERR: return;
}
```

(b)

Figure 9.7: Programs that illustrate the benefit of using a conceptually infinite abstract graph.

only needs to learn this once and gets automatic reuse at all call sites. Note that such reuse is achieved in a different way in SMASH (Godefroid et al., 2010), which makes use of explicit procedure summaries. However, because the split between local and global variables is not known when analyzing machine code, it is not clear to us how MCVETO could generate such explicit summaries.

Furthermore, SMASH is still restricted to invoking intraprocedural analysis as a subroutine, whereas MCVETO is not limited to considering frontiers in just a single procedure: at each stage, it is free to choose a frontier in *any* procedure. To see why such freedom can be important, consider the source-code example in Figure 9.7(b) (where *target* is `ERR`). DASH might proceed as follows. The initial test uses $[\mathtt{x} \mapsto 42]$, which goes through `bar1`, but does not reach *target*. After

a few iterations, the frontier is the call to `bar1`, at which point DASH is invoked on `bar1` to prove that the return value is not 10. The subproof takes a long time because of the complicated loop in `bar1`. In essence, DASH gets stuck in `bar1` without recourse to an easier way to reach *target*. MCVETO can make the same choices, and would start to prove the same property for the return value of `bar1`. However, refinements inside of `bar1` cause the abstract graph to grow, and at some point, if the policy is to pick a frontier *closest* to *target*, the frontier switches to one in `main` that is closer to *target*—in particular, the true branch of the if-condition `x==0`. MCVETO will be able to extend that frontier by running a test with $[\texttt{x} \mapsto 0]$, which will go through `bar2` and reach *target*. The challenge that we face to support such flexibility is how to select the frontier while accounting for paths that reflect the nesting structure of calls and returns. As discussed below, by doing computations via automata, transducers, and pushdown systems, MCVETO can find the set of *all* frontiers, as well as identify the k *closest* frontiers.

Symbolic Methods to Find All Frontiers and Closest Frontiers

Definition 9.3. A *configuration* u of an NWA A is a sequence of states; that is, $u \in Q^*$. The transition relation δ of A defines a *transition relation* $\overset{\delta}{\Rightarrow}$ on configurations of A as follows:

- if $(q, \sigma, q') \in \delta_i$, then $qu \overset{\delta}{\Rightarrow} q'u$ for all $u \in Q^*$
- if $(q_c, \sigma, q_e) \in \delta_c$, then $q_c u \overset{\delta}{\Rightarrow} q_e q_c u$ for all $u \in Q^*$
- if $(q_x, q_c, \sigma, q_r) \in \delta_r$, then $q_x q_c u \overset{\delta}{\Rightarrow} q_r u$ for all $u \in Q^*$ ■

We often work with abstractions of NWAs in which alphabet symbols are dropped: $\Rightarrow \overset{\text{def}}{=} \bigcup_{\sigma \in \Sigma} \overset{\delta}{\Rightarrow}$. Let $\Rightarrow^*$ denote the reflexive transitive closure of $\Rightarrow$. For a set of configurations C, $\mathit{pre}^*(C) \overset{\text{def}}{=} \{c' \mid \exists c \in C : c' \Rightarrow^* c\}$ and $\mathit{post}^*(C) \overset{\text{def}}{=} \{c' \mid \exists c \in C : c \Rightarrow^* c'\}$—i.e., backward and forward reachability, respectively, with respect to transition relation $\Rightarrow$. When C is a regular language of configurations, automata for the configuration languages $\mathit{pre}^*(C)$ and $\mathit{post}^*(C)$ can be computed in polynomial time (Driscoll et al., 2012). Note that the check on line 6 of Algorithm 19 can be performed by testing whether $\mathit{start} \in \mathit{pre}^*(\mathit{target})$.

Given the set of all $\langle \text{PC}, \textit{stack} \rangle$ configurations for concrete states that occur in some execution trace, let X be the corresponding set of configurations in Q^* for the NWA of the abstract graph. It is straightforward to build an automaton that recognizes X because we can recover $\langle \text{PC}, \textit{stack} \rangle$ information during a traversal of an NWPrefix. (Henceforth, the automaton is referred to as X, as well.) We also create an automaton InChopButNotExecuted for the part of the chop between *start* and *target* that has not been reached during a concrete execution:

$$\text{InChopButNotExecuted} = \textit{post}^*(\textit{start}) \cap \textit{pre}^*(\textit{target}) \cap \neg X.$$

Given an automaton A for a language $L(A)$, let ID$[A]$ be the transducer for the projection of the identity relation on $L(A)$: $\{(a, a) \mid a \in L(A)\}$. It is straightforward to create a transducer for the *post* relation on NWA configurations: e.g., $(q, \sigma, q') \in \delta_i$ contributes the fragment

$$\rightarrow \bullet \overset{q/q'}{\rightarrow} \circledcirc\!\!\circlearrowleft q/q,\ q \in \Sigma$$

to the transducer. (Note that the transducer encodes *post*, not *post**.) Now we put these together to find all frontiers:

$$\text{Frontiers} = \text{ID}[X] \circ \textit{post} \circ \text{ID}[\text{InChopButNotExecuted}],$$

where $\circ$ denotes transducer composition. In English, what this does is the following: Frontiers identifies—as a relation between configuration pairs of the form $(x, \textit{icbne})$—all edges in the infinite transition relation of the abstract graph in which

1. x is reached during concrete execution (ID$[X]$)
2. one can go from x to *icbne* in one step (*post*)
3. *icbne* is on a path from *start* to *target* in the infinite transition relation of the abstract graph, but was not reached during a concrete execution (ID[InChopButNotExecuted])

The composition with the two projection functions, "ID[X]$\circ$..." and "...$\circ$ID[InChopButNotExecuted]", serves to specialize *post* to just the edges in the infinite transition relation of the abstract graph that run between a configuration in X and a configuration in InChopButNotExecuted.

We can obtain "closest frontiers" by using *weighted* NWAs (Driscoll et al., 2012) and adding shortest-distance weights to either *pre**(*target*) (to obtain frontiers that are closest to *target*) or to *post**(*start*) (to obtain frontiers that are closest to *start*), and then carrying the weights through the transducer-composition operations.

9.2.5 A Language-Independent Approach to Aliasing Relevant to a Property

This section describes how MCVETO identifies—in a language-independent way suitable for use with machine code—the aliasing condition relevant to a property in a given state. Lim et al. (2009) showed how to generate a Pre primitive for machine code; however, repeated application of Pre causes refinement predicates to explode. We now present a language-independent algorithm for obtaining an aliasing condition α that is suitable for use in machine-code analysis. From α, one immediately obtains Pre_α. There are two challenges to defining an appropriate notion of aliasing condition for use with machine code: (i) `int`-valued and address-valued quantities are indistinguishable at runtime, and (ii) arithmetic on addresses is used extensively.

Suppose that the frontier is (n, I, m), ψ is the formula on m, and S_n is the symbolic state obtained via symbolic execution of a concrete trace that reaches n. For source code, Beckman et al. (2008) identify aliasing condition α by looking at the relationship, in S_n, between the addresses written to by I and the ones used in ψ. However, their algorithm for computing α is language-*dependent*: their algorithm has the semantics of C implicitly encoded in its search for "the addresses written to by I". In contrast, as explained below, we developed an alternative, language-*independent* approach, both to identifying α and computing Pre_α. Further details on this particular problem can be found in Thakur et al. (2010).

```
void bar() {
  ERR: // address here is 0x10
}
void foo() {
  int b = MakeChoice() & 1;
  int r = b*0x68 + (1-b)*0x10;
  *(&r+2) = r;
  return;
}
int main() {
  foo();
  // address here is 0x68
}
```

Figure 9.8: `ERR` is reachable, but only along a path in which a `ret` instruction serves to perform a call.

9.2.6 Soundness Guarantee

The soundness argument for McVeto is more subtle than it otherwise might appear because of examples like the one shown in Figure 9.8. The statement `*(&r+2) = r;` overwrites `foo`'s return address, and `MakeChoice` returns a random 32-bit number. At the end of `foo`, half the runs return normally to `main`. For the other half, the `ret` instruction at the end of `foo` serves to call `bar`. The problem is that for a run that returns normally to `main` after trace generalization and intersection with G_0, there is no frontier. Consequently, half of the runs of McVeto, on average, would erroneously report that location `ERR` is unreachable.

McVeto uses the following policy P for classifying execution steps: (a) the position of any form of `call` instruction is a call-position; (b) the position of any form of `ret` instruction is a return-position. Our goals are (i) to define a property Q that is compatible with P in the sense that McVeto can check for violations of Q *while checking only NWPrefix paths*, and (ii) to establish a soundness guarantee: either McVeto reports that Q is violated (along with an input that demonstrates it), or it reports that *target* is reachable (again with an input that demonstrates it), or it correctly reports that Q is invariant and *target* is unreachable. To define Q, we augment the instruction-set semantics with an auxiliary stack. Initially, the auxiliary stack is empty; at each

`call`, a copy of the return address pushed on the processor stack is also pushed on the auxiliary stack; at each `ret`, the auxiliary stack is popped.

Definition 9.4. An **acceptable execution** (AE) under the instrumented semantics is one in which at each `ret` instruction (i) the auxiliary stack is non-empty, and (ii) the address popped from the processor stack matches the address popped from the auxiliary stack. ■

In the instrumented semantics, a flag V is set whenever the program performs an execution step that violates either condition (i) or (ii) of Definition 9.4. Instead of the initial NWA shown in Figure 9.3, we use a similar two-state NWA that has states q_1: PC $\neq$ *target* $\wedge$ $\neg$V and q_2: PC $=$ *target* $\vee$ V, where q_1 is non-accepting and q_2 is accepting. In addition, we add one more rule to the trace-generalization construction for G_π from Figure 9.5:

12. For each return-step $\langle a_x, \texttt{ret}, a_r \rangle$, G_π has an internal-transition $(q_{a_x}, \texttt{ret}, TS)$.

As shown below, these modifications cause the DPG algorithm to also search for traces that are AE violations.

Theorem 9.5 (Soundness of McVeto)**.**

1. *If McVeto reports "AE violation" (together with an input S), execution of S performs an execution that is not an AE.*
2. *If McVeto reports "bug found" (together with an input S), execution of S performs an AE to target.*
3. *If McVeto reports "OK", then (a) the program performs only AEs, and (b) target cannot be reached during any AE.*

Proof. If a program has a concrete execution trace that is not AE, there must exist a shortest non-AE prefix, which has the form "NWPrefix `ret`" where either (i) the auxiliary stack is empty, or (ii) the return address used by `ret` from the processor stack fails to match the return address on the auxiliary stack. At each stage, the abstract graph used by McVeto accepts an overapproximation

of the program's shortest non-AE execution-trace prefixes. This is true of the initial graph G_0 because internal transitions have wild-card symbols. Moreover, each folded trace $G_\pi = \pi/[\text{PC}]$ accepts traces of the form "NWPrefix `ret`" due to the addition of internal transitions to *TS* for each `ret` instruction (item 12 above). NWA intersection of two sound overapproximations leads to a refined sound overapproximation. Therefore, when MCVETO has shown that no accepting state is reachable, it has also proved that the program has no AE violations. □

For an example such as Figure 9.8, MCVETO reports "AE violation".

In cases when MCVETO reports "AE violation", it can indicate a stack-smashing attack. If one wishes to find out more information when there is an AE violation, one can run a purely intraprocedural version of MCVETO that does not give special treatment to `call` and `ret` instructions. This approach is potentially more expensive than running the interprocedural version of MCVETO, but it can find out additional information about executions that are not AE.

9.3 Implementation

The MCVETO implementation incorporates all of the techniques described in Section 9.2. The implementation uses only language-independent techniques; consequently, MCVETO can be easily retargeted to different languages. The main components of MCVETO are language-independent in two different dimensions:

1. The MCVETO DPG driver is structured so that one only needs to provide implementations of primitives for concrete and symbolic execution of a language's constructs, plus a handful of other primitives (e.g., Pre_α). Consequently, this component can be used for both source-level languages and machine-code languages.

2. For machine-code languages, I used two tools that *generate* the required implementations of the primitives for concrete and symbolic execution from descriptions of the syntax and concrete operational semantics of an instruction set. The abstract syntax and concrete semantics are specified using TSL (Lim and Reps, 2008). Translation of binary-encoded instructions

to abstract syntax trees is specified using a tool called ISAL (**I**nstruction **S**et **A**rchitecture **L**anguage).[2] The relationship between ISAL and TSL is similar to the relationship between Flex and Bison—i.e., a Flex-generated lexer passes tokens to a Bison-generated parser. In our case, the TSL-defined abstract syntax serves as the formalism for communicating values—namely, instructions' abstract syntax trees—between the two tools.

To perform symbolic queries on the conceptually-infinite abstract graph Section 9.2.4), the implementation uses OpenFst (Allauzen et al., 2007) (for transducers) and OpenNWA (Driscoll et al., 2012) (for NWAs).

9.4 Experimental Evaluation

The experiments in this section were designed to answer the following questions:

1. How effective is the $\mathcal{BVI}$ abstract domain (Chapter 8) for verifying machine code?
2. How effective is MCVETO in verifying machine code?
3. How does speculative trace refinement (Section 9.2.3) impact the performance of MCVETO?

We used benchmarks taken from Gulavani et al. (2006). The examples are small, but challenging. Our experiments (see Table 9.1) were run on a single core of a single-processor quad-core 3.0 GHz Xeon computer running Windows 7, configured so that a user process has 4 GB of memory. The benchmarks were compiled using Visual Studio 2010, and the executables were analyzed (without using symbol-table information). Out of the 16 benchmarks used, the target was reachable in 4, and the target PC was unreachable in 12; in Table 9.1, $[\rightsquigarrow]$ means the target PC is reachable, $[\not\rightsquigarrow]$ means the target PC is unreachable.

Analysis based on $\mathcal{BVI}$ abstract domain was able to prove that the target was unreachable for 8 out of the 12 unreachable benchmarks. An analysis based only on the affine-equality $\mathcal{E}_{2^{32}}$

[2] ISAL also handles other kinds of concrete syntactic issues, including (a) *encoding* (abstract syntax trees to binary-encoded instructions), (b) *parsing assembly* (assembly code to abstract syntax trees), and (c) *assembly pretty-printing* (abstract syntax trees to assembly code).

	Test Name	Expected	$\mathcal{BVI}$	McVeto	McVeto$^+$
1.	badBuggy	[$\rightsquigarrow$]	62.63 [?]	0.98 [$\rightsquigarrow$]	1.11 [$\rightsquigarrow$]
2.	berkeley	[$\rightsquigarrow$]	102.60 [?]	26.81 [$\rightsquigarrow$]	**11.95** [$\rightsquigarrow$]
3.	berkeley-nat	[$\rightsquigarrow$]	103.32 [?]	35.28 [$\rightsquigarrow$]	**21.65** [$\rightsquigarrow$]
4.	fig7	[$\rightsquigarrow$]	6.29 [?]	0.56 [$\rightsquigarrow$]	0.58 [$\rightsquigarrow$]
5.	cars	[$\not\rightsquigarrow$]	63.62 [?]	12.61 [$\not\rightsquigarrow$]	14.85 [$\not\rightsquigarrow$]
6.	cars.2	[$\not\rightsquigarrow$]	107.82 [?]	12.8 [$\not\rightsquigarrow$]	15.52 [$\not\rightsquigarrow$]
7.	fig6	[$\not\rightsquigarrow$]	6.42 [$\not\rightsquigarrow$]	0.87 [$\not\rightsquigarrow$]	2.86 [$\not\rightsquigarrow$]
8.	fig8	[$\not\rightsquigarrow$]	18.16 [?]	12.91 [$\not\rightsquigarrow$]	12.82 [$\not\rightsquigarrow$]
9.	fig9	[$\not\rightsquigarrow$]	1.50 [$\not\rightsquigarrow$]	1.22 [$\not\rightsquigarrow$]	2.78 [$\not\rightsquigarrow$]
10.	prog1	[$\not\rightsquigarrow$]	5.21 [$\not\rightsquigarrow$]	0.88 [$\not\rightsquigarrow$]	0.87 [$\not\rightsquigarrow$]
11.	prog2	[$\not\rightsquigarrow$]	11.30 [$\not\rightsquigarrow$]	0.96 [$\not\rightsquigarrow$]	1.78 [$\not\rightsquigarrow$]
12.	prog3	[$\not\rightsquigarrow$]	4.59 [$\not\rightsquigarrow$]	0.42 [$\not\rightsquigarrow$]	0.63 [$\not\rightsquigarrow$]
13.	prog4	[$\not\rightsquigarrow$]	125.48 [?]	[—]	[—]
14.	prog5	[$\not\rightsquigarrow$]	5.69 [$\not\rightsquigarrow$]	1.2 [$\not\rightsquigarrow$]	1.44 [$\not\rightsquigarrow$]
15.	test1	[$\not\rightsquigarrow$]	25.07 [$\not\rightsquigarrow$]	**2.52** [$\not\rightsquigarrow$]	[—]
16.	test2	[$\not\rightsquigarrow$]	58.567 [$\not\rightsquigarrow$]	**3.12** [$\not\rightsquigarrow$]	[—]

Table 9.1: Machine-code verification experiments. The first column displays the test name. The Expected column lists the expected answer: [$\rightsquigarrow$] means the target PC is reachable, [$\not\rightsquigarrow$] means the target PC is unreachable. The '$\mathcal{BVI}$' column displays the time taken (in seconds) and the answer obtained when performing analysis based on the $\mathcal{BVI}$ abstract domain; [?] means the analysis could not determine whether or not the target PC was reachable. No total timeout was specified for the $\mathcal{BVI}$ analysis. The 'McVeto' column displays the time taken (in seconds) by McVeto when not using any speculative trace refinement. The McVeto$^+$ column displays the time taken (in seconds) by McVeto when using speculative trace refinement. [—] means McVeto or McVeto$^+$ timed out after 100 seconds.

over registers was unable to prove unreachability for any of the benchmarks. This result answers question 1. Note that for the 4 benchmarks in which the target is reachable, the $\mathcal{BVI}$ analysis reports that there could be a path to the target, which is the only possible sound answer for such an analysis.

The 'McVeto' column in Table 9.1 displays the time taken (in seconds) by McVeto for verifying the benchmarks when not using speculative trace refinement (Section 9.2.3). The total timeout for McVeto was set to 100 seconds. For all but two of the benchmarks, McVeto was able to correctly ascertain the reachability or unreachability of the target. This result answers question 2. Note that no total timeout was given for the $\mathcal{BVI}$ analysis. Furthermore, the time reported for McVeto includes the time taken to learn the control-flow graph of the program using

trace generalization (Section 9.2.2); on the other hand, the $\mathcal{BVI}$ analysis uses CodeSurfer/x86 (Balakrishnan et al., 2005) to construct the control-flow graph.

In Table 9.1, McVeto$^+$ represents the McVeto algorithm when using speculative trace refinement. McVeto$^+$ performs better than McVeto for two of the reachable benchmarks, while it performs worse for two unreachable benchmarks, as highlighted in Table 9.1. This experiment answers question 3.

9.5 Related Work

Machine-Code Analyzers Targeted at Finding Vulnerabilities. A substantial amount of work exists on techniques to detect security vulnerabilities by analyzing source code for a variety of languages (Wagner et al., 2000; Livshits and Lam, 2005; Xie and Aiken, 2006). Less work exists on vulnerability detection for machine code. Kruegel et al. (2005) developed a system for automating mimicry attacks; it uses symbolic execution to discover attacks that can give up and regain execution control by modifying the contents of the data, heap, or stack so that the application is forced to return control to injected attack code at some point after the execution of a system call. Cova et al. (2006) used that platform to detect security vulnerabilities in x86 executables via symbolic execution.

Prior work exists on directed *test* generation for machine code (Godefroid et al., 2008; Brumley et al., 2008). Directed test generation combines concrete execution and symbolic execution to find inputs that increase test coverage. An SMT solver is used to obtain inputs that force previously unexplored branch directions to be taken. In contrast, McVeto implements directed *proof* generation. Unlike directed-test-generation tools, McVeto is goal-directed, and works by trying to refute the claim "no path exists that connects program entry to a given goal state".

Machine-Code Model Checkers. The Synergy model checker can be used to verify safety properties of an x86 executable compiled from a "single-procedure C program with only [`int`-valued] variables" (Gulavani et al., 2006) (i.e., no pointers). It uses debugging information to obtain information about variables and types, and uses Vulcan (Srivastava et al., 2001) to obtain

a CFG. It uses integer arithmetic—not bit-vector arithmetic—in its solver. In contrast, McVeto addresses the challenges of checking properties of stripped executables.

Air ("Assembly Iterative Refinement") (Chaki and Ivers, 2009) is a model checker for PowerPC. Air decompiles an assembly program to C, and then checks if the resulting C program satisfies the desired property by applying Copper (Chaki et al., 2004), a predicate-abstraction-based model checker for C source code. They state that the choice of Copper is not essential, and that any other C model checker, such as Slam (Ball and Rajamani, 2001) or Blast (Henzinger et al., 2002) would be satisfactory. However, the C programs that result from their translation step use pointer arithmetic and pointer dereferencing, whereas many C model checkers, including Slam and Blast, make unsound assumptions about pointer arithmetic.

[Mc]Square (Schlich, 2008) is a model checker for microcontroller assembly code. It uses explicit-state model-checking techniques (combined with a degree of abstraction) to check CTL properties.

Self-Modifying Code. The work on McVeto addresses a problem that has been almost entirely ignored by the PL research community. There is a paper on SMC by Gerth (1991), and a recent paper by Cai et al. (2007). However, both of the papers concern proof systems for reasoning about SMC. In contrast, McVeto can verify (or detect flaws in) SMC automatically.

As far as I know, McVeto is the first model checker to address verifying (or detecting flaws in) SMC.

Trace Generalization. The trace-generalization technique of Section 9.2.2 has both similarities to and differences from the *path programs* of Beyer et al. (2007) and the *trace-refinement* technique of Heizmann et al. (2010). All three techniques refine an overapproximation to eliminate *families* of infeasible concrete traces. However, trace generalization obtains the desired outcome in a substantially different way. Beyer et al. analyze refuted abstract traces to obtain new predicates to refine the predicate abstraction in use. The subsequent refinement step requires possibly expensive calls on an SMT solver to compute new abstract transformers. Heizmann et al. adopt a language-theoretic viewpoint: once a refutation automaton is constructed—which involves

calling an SMT solver and an interpolant generator—refinement is performed by automaton complementation followed by automaton intersection. In contrast, our generalized traces are created by generalizing a *feasible concrete trace* to create directly a representation that overapproximates the set of minimal traces that reach *target*. Consequently, refinement by trace generalization involves *no calls on an SMT solver*, and *avoids the potentially expensive step of automaton complementation*.

9.6 Chapter Notes

The design and implementation of MCVETO was based on that of the MCDASH tool (Lal et al., 2009). In particular, both tools use directed proof generation (Section 9.1) and a language-independent algorithm to identify aliasing relevant to a property (Section 9.2.5). However, the techniques for trace generalization (Section 9.2.2), speculative trace refinement (Section 9.2.3), interprocedural DPG (Section 9.2.4), and finding acceptable-execution violations (Section 9.2.6) are not found in MCDASH.

Chapter 10

A Distributed SAT Solver

> Groups do not need to be dominated by exceptionally intelligent people in order to be smart. Even if most of the people within a group are not especially well-informed or rational, it can still reach a collectively wise decision. This is a good thing, since human beings are not perfectly designed decision makers. [...] Yet despite all these limitations, when our imperfect judgments are aggregated in the right way, our collective intelligence if often excellent.
>
> — James Surowiecki, *The Wisdom of Crowds*

In this chapter, I describe a new distributed SAT solver, called `DiSSolve`, which uses a new proof rule that combines concepts from Stålmarck's method with those found in modern SAT solvers.

A modern *sequential* SAT solver usually implements same variant of the Davis, Putnam, Logemann, and Loveland (DPLL) procedure (Davis et al., 1962). A DPLL-based SAT solver performs a backtrack search over the set of assignments to literals. When a modern solver reaches a conflict during the search, it generates a *conflict clause* (or learned clause) (Marques Silva and Sakallah, 1996; Moskewicz et al., 2001). This learned clause enables the solver to prune the search space. This technique for "learning from failure" is called Conflict-Driven Clause Learning (CDCL). More information regarding DPLL/CDCL solvers can be found in Darwiche and Pipatsrisawat (2009); Marques-Silva et al. (2009).

A modern *parallel* SAT solver falls into the following two categories:

- The divide-and-conquer approach incrementally divides the search space into subspaces, which are then allocated to a DPLL/CDCL solver (Chrabakh and Wolski, 2003; Chu et al., 2008). Care has to be taken to ensure load balancing when performing the division of the search space.
- The Parallel Portfolio approach exploits the complementarity of different sequential DPLL/CDCL strategies to let them compete and cooperate on the same formula (Hamadi et al., 2009b). Though load balancing is not an issue in this approach, care has to be taken to craft the individual strategies.

The `DiSSolve` algorithm (Section 10.1.1) presented in this chapter can be seen as combining concepts from Stålmarck's method and modern DPLL/CDCL solvers:

1. `DiSSolve` partitions the search space using k variables in the same fashion that the Dilemma Rule partitions the search space in Stålmarck's method.
2. Each of the 2^k branches can be solved concurrently with the help of a sequential DPLL/CDCL solver, similar to what is done in the divide-and-conquer approach discussed above.
3. The DPLL/CDCL solver assigned to a branch is allotted a finite amount of time, after which the DPLL/CDCL solver returns a set of learned clauses. Such a branch-and-merge approach does not have to as careful about load balancing, unlike the divide-and-conquer approach.
4. `DiSSolve` performs a *union* of the information from all the branches, instead of an *intersection* as done in Stålmarck's method. Performing a union of clauses is more effective at pruning the search space compared to computing an intersection: with intersection only *common* information learned by every process can be used to prune the search space, while with union, *all* of the information learned by *each* process can be used to prune the search space. In abstract-interpretation terms, `DiSSolve` combines the information from

the branches using a *meet* ($\sqcap$), while the Dilemma Rule in Stålmarck's method combines the information using a *join* ($\sqcup$).

The `DiSSolve` implementation (Section 10.1.2) also makes use of techniques already implemented in modern DPLL/CDCL solvers, such as variable branching heuristics.

The contributions of this chapter can be summarized as follows:

- I describe, `DiSSolve`, a new SAT solver that combines the Dilemma rule in Stålmarck's method with clause-learning techniques from DPLL/CDCL solvers (Section 10.1).
- I evaluate the performance of `DiSSolve` when deployed on a multi-core machine and on the cloud (Section 10.2).
- I present a natural extension of the `DiSSolve` algorithm from SAT to SMT, and describe the `DiSSolve` algorithm as an SMA solver (Section 10.3).

Section 10.4 presents related work.

10.1 The `DiSSolve` Algorithm

This section explains the algorithm used in `DiSSolve`. Section 10.1.1 explains the basic algorithm and skips over some of the details of the algorithm. These algorithm details are filled in Section 10.1.2. Thus, Section 10.1.1 can be viewed as describing the algorithm *mechanism*, and Section 10.1.2 describing the algorithm *policy*.

10.1.1 Basic Algorithm

Algorithm 20 shows the basic `DiSSolve` algorithm for deciding the satisfiability of a propositional-logic formula φ. C is a set of clauses, which is initialized to the empty set (line 1). Recall that a *clause* c is a disjunction of literals. I use ϵ to represent an empty clause, whose meaning is to be interpreted as **false**. The meaning of a set of clauses $C = \{c_1, c_2, \ldots, c_n\}$ is: $[\![C]\!] \stackrel{\text{def}}{=} \bigwedge_{1 \leq i \leq n} c_i$. The meaning of the empty set of clauses $\emptyset$ is **true**.

Algorithm 20: `DiSSolve`(φ)

```
1  C ← ∅
2  for j ∈ {0, 1, 2, ...} do
3      V_j ← SplittingVariables(j)
4      t_j ← TimeBudget(j)
5      (ans_j, C) ← DiSSolveSplit(φ, C, V_j, t_j)
6      if ans_j is sat or ans_j is unsat then
7          return ans_j
```

Algorithm 21: DissolveSplit(φ, C, V, t)

```
1  k ← |V|
2  {a_0, a_1, ..., a_{2^k − 1}} ← DilemmaAssumptions(V)
3  for i ∈ {0, 1, ..., 2^k − 1} do
4      (ans_i, C_i) ← Deduce(φ, C, a_i, t)
5      if ans_i is sat then
6          return (sat, ∅)
7  if all ans_i are unsat then
8      return (unsat, {ϵ})
9  C' ← UnionClauses(C, C_0, C_1, C_2 ..., C_{2^k − 1})
10 return (unknown, C')
```

In the loop from lines 2–7, `DiSSolve` repeatedly calls DiSSolveSplit (Algorithm 21). The loop invariant is $\varphi \Rightarrow C$. V_j is a set of variables to be used in iteration j, which are returned by a call to `SplittingVariables` (line 3). A call to `TimeBudget` returns the time budget t_j to be used by iteration j. The formula φ, the clauses C, the splitting variables V_j, and the time budget t_j are passed to the call to DiSSolveSplit on line 5. If the answer ans_j returned by DiSSolveSplit is sat or unsat, then `DiSSolve` returns; if the answer is unknown, then the set of clause C returned by DiSSolveSplit is used in the next iteration of the loop.

Algorithm 21 shows the algorithm for DiSSolveSplit, which forms the core of the `DiSSolve` algorithm. If the k represents the size of the set of variables V, the call to `DilemmaAssumptions`(V) returns the full set of 2^k assignments to the variables V (line 2). Each assignment (or *assumption*) a_i is passed to the call to `Deduce` on line 4, along with the formula φ, set of clauses C, and the time budget t. If any of the calls to `Deduce` returns sat within time budget t, then DiSSolveSplit return sat, which implies that the formula φ is satisfiable. If *all* the 2^k calls to `Deduce` return unsat,

then DiSSolveSplit returns `unsat`, which implies that φ is unsatisfiable. If neither of these two cases occur, then DiSSolveSplit performs a union of (i) the clauses C_i returned by the 2^k separate calls to `Deduce`, and (ii) the clause C (line 9).

Each call to function `Deduce` on line 4 attempts to find, within the time budget t, either a satisfying assignment for φ under the given assumption a_i, or to prove that φ is unsatisfiable under a_i. The set of clauses C_i returned by `Deduce` on line 4 is such that $\varphi \Rightarrow C_i$. This implication follows from the inherent property of conflict clauses that they are independent of the assumption a_i under which they occur (Eén and Sörensson, 2003). DiSSolveSplit partitions the search space, because each call to `Deduce` on line 4 works on a separate part of the search space. However, the space of deductions is common to all instances; that is, the learned clauses C_i are *pervasive*.

At first glance, the pseudo-code for DiSSolveSplit looks similar to that for $\mathrm{k\text{-}saturation}$ in Stålmarck's method (Algorithm 4, Section 3.4.2). In particular, the use of the 2^k assignments to split the search space is common between the two algorithms. However, at the end of the Dilemma Rule, we perform an *intersection* of the information from the various branches. DiSSolveSplit, on the other hand, performs a *union* of the information computed by each of the individual calls to `Deduce`.

10.1.2 Algorithm Details

This section presents the details not covered in Section 10.1.1; I explain the various details going "bottom-up": I start with `Deduce`, then DiSSolveSplit, and finally `DiSSolve`.

Implementation of Deduce. `Deduce` is implemented as a wrapper around the efficient SAT solver, Glucose 3.0 (Audemard and Simon, 2009), whose implementation is based on MiniSAT (Eén and Sörensson, 2004). The Glucose implementation was modified slightly to communicate with DiSSolveSplit via `protobufs` (protobuf, 2014). Apart from the convenience, the `protobuf` format also compresses the data.

Returned learned clauses. The learned clauses C_i returned by a call to `Deduce` on line 4 are ordered by priority. In particular, Glucose uses the Literals Blocks Distance (LBD) metric to

rank each learned clause (Audemard and Simon, 2009). Furthermore, the maximum length is restricted to be no greater than some fixed value m. (In practice, the value of m was chosen to be 30.)

Cancellation. If the call to `Deduce` on line 4 infers that φ is unsatisfiable under the assumption a_i, then it also returns a *final conflict clause*. A conflict clause c_f only contains a subset of the literals occurring in assumption a_i, and is such that $\varphi \wedge c$ is unsatisfiable. This conflict clause c_f is used to cancel other ongoing branches. For example, if $c_f \equiv (v_1 \vee \neg v_2)$, then `DiSSolveSplit` can safely cancel all pending calls to `Deduce` that have either $v_1 =$ **true** or $v_2 =$ **false** as part of their assumption. In practice, instead of merely canceling a call to `Deduce`, it is sent an interrupt. Upon receiving the interrupt, the call to `Deduce` terminates, but returns its current set of learned clauses.

Concurrent execution. Each call to `Deduce` in the loop in DiSSolveSplit (lines 3–6) can be executed concurrently, because each of the 2^k calls to `Deduce` are independent of each other.

DiSSolveSplit schedules the 2^k calls to `Deduce` onto the c compute engines provided. If `DiSSolve` is given access to $c = 2^n$ compute engines, picking $k = (n + 1)$ seems to work well. I call this technique *overclocking*. The intuition behind this is that short final conflict clauses often cause many of the `Deduce` calls to be terminated. Overclocking increases the utilization of the c compute engines.

Clause Management. On line 9, `UnionClauses` aggregates the learned clauses returned by each of the 2^k branches. If we simply perform a union of the clauses, then very quickly the number of clauses maintained by `DiSSolve` will grow prohibitively large. These clauses have to be communicated to each of the calls to `Deduce`, sometimes over a network. Thus, to control the number of clauses maintained, the call to `UnionClauses` on line 9 of DiSSolveSplit does the following:

- A UBTree (Hoffmann and Koehler, 1999), which is a trie data structure, is used to implement subsumption of clauses. For example, the clause $v_1 \vee v_4$ subsumes the (longer) clause $v_1 \vee v_2 \vee \neg v_3 \vee v_4$.

- A maximum budget on the number of clauses is used. As stated above, each `Deduce` call returns the clauses sorted based on their quality. If the number of clauses exceeds the clause budget, then clauses with lower priority are discarded until the number of clauses does not exceed the clause budget. Unary and binary clauses are never discarded.

Choosing splitting variables. Each call to `Deduce` returns an ordered sequence of k variables it considers as useful to split on; each variable is assigned a weight reflecting its priority. `SplittingVariables` (line 3 in `DiSSolve`) picks the k variables with the highest weight. This technique allows `DiSSolve` to reuse the decision-variable heuristics implemented in existing SAT solver. To make the pseudo-code simpler, the return of splitting variables is not shown on line 4 of DiSSolveSplit.

Time budget. As seen in line 4, the call to DiSSolveSplit in the jth iteration of the loop in `DiSSolve` is provided a time budget t_j. Thus, each iteration of the loop in `DiSSolve` can be viewed as performing a *restart* of the search. It has been argued in Gomes et al. (1998) that randomization and, in particular, restarts, counter the heavy-tail behavior found in combinatorial search. The question arises regarding what the restart schedule should be. One possibility is to use the same constant time budget for each t_j. However, it has been shown that restart schedules that vary the time budget work better in practice (Gomes et al., 1998, 2000). I experimented with the following two restart sequences:

- Luby restart sequence (Luby et al., 1993)
- PicoSAT restart sequence (Biere, 2008, Section 3.2)

In Section 10.2, I use DiSSolveSplit$_f$ to mean a call to DiSSolveSplit in which the time budget t is finite, and DiSSolveSplit$_\infty$ to mean a call to DiSSolveSplit in which the time budget t is infinite. Note that DiSSolveSplit$_\infty$ never performs a merge of the branches by calling `UnionClauses`, and, thus, does not make use of the repeated split-and-merge behavior of `DiSSolve`.

Phase Saving. One disadvantage of frequent restarts discussed by other reseachers is that the restart could erase partial solutions, and as a result might cause work to be repeated. *Phase saving*

(Pipatsrisawat and Darwiche, 2007) is a technique that simply saves the partial solutions. In particular, the technique maintains an additional array of literals, called the *saved-literal array*. Every time the solver backtracks and erases assignments, each erased assignment is saved in the saved-literal array. Now, any time the solver decides to branch on variable v, it uses the value in the saved-literal array, if one exists, to decide the polarity of the variable.

This phase-saving technique is implemented in the Glucose solver that is used to implement `Deduce`. Each call to `Deduce` is given as input and returns as output the saved-literal array. DiSSolveSplit aggregates the information from all 2^k saved-literal arrays to compute the saved-literal array for the next iteration. This technique allows some of the partial solutions to reused across iterations. To make the pseudo-code simpler, the use of a saved-literal array is not shown on line 4 of DiSSolveSplit.

10.2 Experimental Evaluation

This section describes an experimental evaluation of `DiSSolve`. Section 10.2.1 describes an experiment for configuring `DiSSolve`. Section 10.2.2 presents an evaluation of `DiSSolve` when it is deployed on a multicore machine. Section 10.2.3 presents an evaluation of `DiSSolve` when it is deployed on hundreds of machines on a cloud-computing platform.

10.2.1 Configuring `DiSSolve`

The experiments described in this section address the following questions:

1. What are good parameter settings for `DiSSolve`?

2. Does DiSSolveSplit$_f$ perform better than DiSSolveSplit$_\infty$?

ParamILS (Hutter et al., 2009) is an automatic algorithm configuration framework. It explores the parameter-configuration space using iterated local search along with solution perturbation to escape from local optima. I used ParamILS to find a good configuration out of a total of 1296 configurations corresponding to the various settings for 6 parameters for `DiSSolve`. One of the

parameters controlled the time budget for each round of DiSSolveSplit—in particular, whether the time budget should be (effectively) infinite (DiSSolveSplit$_\infty$), or whether it should be finite (DiSSolveSplit$_f$). If a finite time budget is used, then various restart sequences could be used: a fixed budget, the Luby restart sequence (Luby et al., 1993), or the PicoSAT restart sequence (Biere, 2008, Section 3.2).

ParamILS adaptively limits the time spent for evaluating individual configurations; the time limit for evaluating an individual configuration t_s can be specified by the user. In this experiment, I used $t_s = 250$ seconds.

The set of benchmark instances consisted of a total of 79 satisfiable and unsatisfiable formulas taken from the application track of SAT-COMP 2013 (SAT-COMP'13, 2013). These benchmarks were chosen because (sequential) Glucose took less than 200 seconds to solve each of them.

The parameter tuning was carried out for 26 hours on an 8-core Intel Xeon 2.4 Ghz machine with 32 GB of RAM running Red Hat Linux 6.5.

The parameter configuration provided by `DiSSolve` is used for the rest of the experiments described in this section.

ParamILS did not return DiSSolveSplit$_\infty$ as part of the final configuration. Instead, it picked a finite time budget per round; the specific restart sequence that was chosen was the PicoSAT restart sequence. This result provides confidence in the fact that the branch-and-merge behavior in Algorithm 20 is useful. This result answers question 2 posed above.

10.2.2 `DiSSolve` on Multi-Core Machine

In this experiment, `DiSSolve` was compared with `ppfolio`, a parallel portfolio solver (Roussel, 2012). The experiment was designed to answer the following question:

1. How does the performance of `DiSSolve` compare with `ppfolio` when solving application track benchmarks?

2. How does the performance of `DiSSolve` compare with `ppfolio` when solving *difficult* benchmarks?

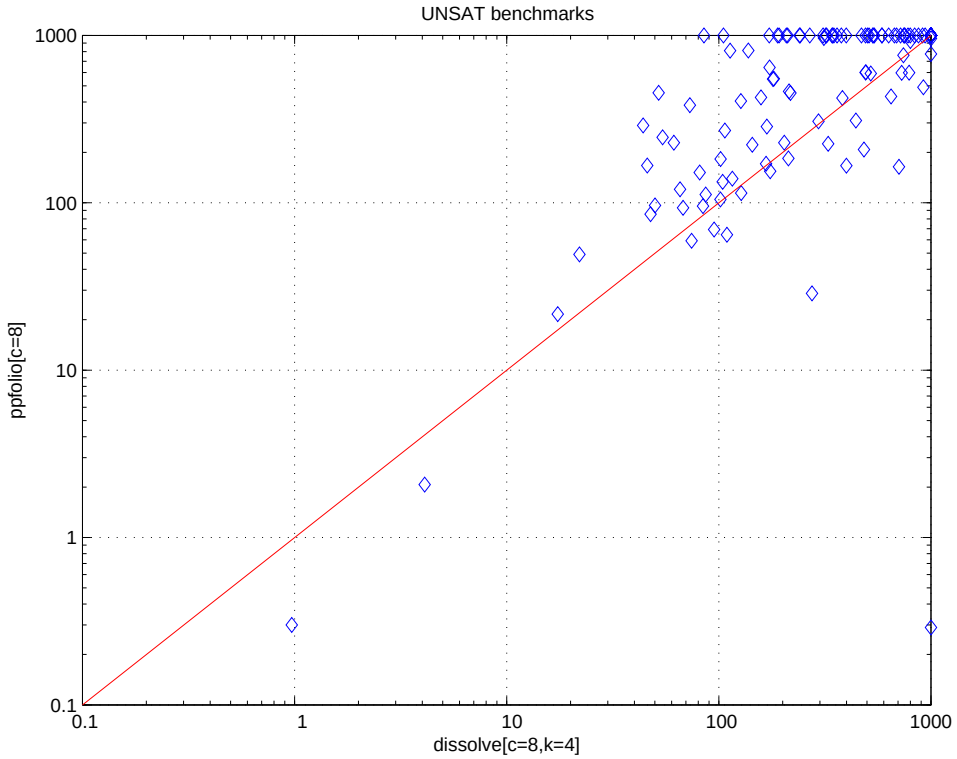


Figure 10.1: Log-log scatter plot comparing the running time (in seconds) of ppfolio$[c = 8]$ and `DiSSolve`$[c = 2^3, k = 4]$ on 150 unsatisfiable (UNSAT) benchmarks.

To answer question 1, we use the 150 unsatisfiable and 150 satisfiable benchmarks in the application track of SAT-COMP 2013 (SAT-COMP'13, 2013). The experiment was run on `n1-standard-8` machine type, which is a standard 1 CPU machine type with 8 virtual CPUs and 30 GB of memory available on the Google Compute Engine service. The time limit per benchmark was 1000 seconds. Figure 10.1 shows the log-log scatter plot comparing the time taken (in seconds) by `DiSSolve`$[c = 2^3, k = 4]$ and `ppfolio`$[c = 8]$ on the 150 unsatisfiable (UNSAT) benchmarks. This scatter plot shows that `DiSSolve` performs significantly better than `ppfolio` on the unsatisfiable benchmarks. This fact is also illustrated using Figure 10.2, which shows the cactus plot for `DiSSolve`$[c = 2^3, k = 4]$ and `ppfolio`$[c = 8]$ using the 150 UNSAT benchmarks. The x-axis shows the number of correctly solved benchmarks. The y-axis displays the cumulative time taken (in seconds) to correctly solve the benchmarks. A point (x, y) denotes that the solver was able to correctly solve at most x benchmarks in y seconds. Thus, a curve that is lower and

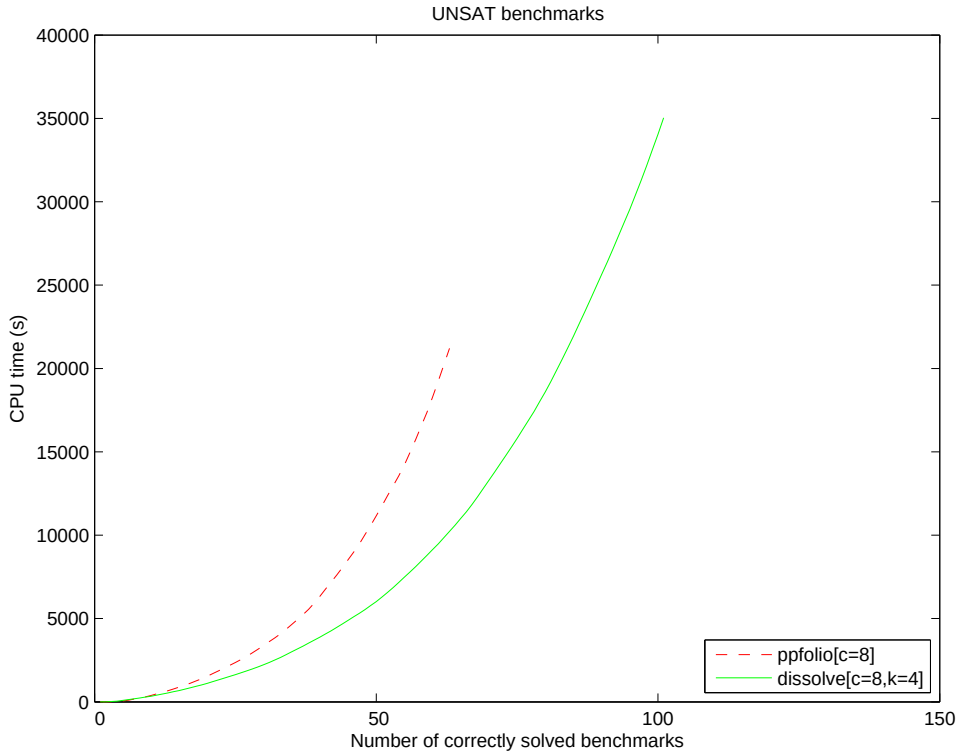


Figure 10.2: Cactus plot comparing the running time (in seconds) of ppfolio$[c = 8]$ and `DiSSolve`$[c = 2^3, k = 4]$ on 150 unsatisfiable (UNSAT) benchmarks.

to the left is better. Figure 10.3 shows the log-log scatter plot comparing the time taken (in seconds) by `DiSSolve`$[c = 2^3, k = 4]$ and `ppfolio`$[c = 8]$ on the 150 satisfiable (SAT) benchmarks. This scatter plot shows that `DiSSolve` performs slightly worse than `ppfolio` on the satisfiable benchmarks. This fact is also illustrated using Figure 10.4, which shows the cactus plot for `DiSSolve`$[c = 2^3, k = 4]$ and `ppfolio`$[c = 8]$ using the 150 SAT benchmarks.

To answer question 2 above, we used the 45 formulas in the application track of SAT-COMP 2013 that were not solved by any sequential solver within a time limit of 10000 seconds (SAT-COMP'13, 2013). Out of the 45 benchmarks, 44 were unsatisfiable instances. The experiment was run on the same 8-core machine described in Section 10.2.1. The time limit for `DiSSolve` and `ppfolio` was 1000 seconds. `ppfolio` timed out for all 45 benchmarks. `DiSSolve` was able to correctly prove unsatisfiability for 19 of the 45 benchmarks. Table 10.1 lists the times for the individual benchmarks.

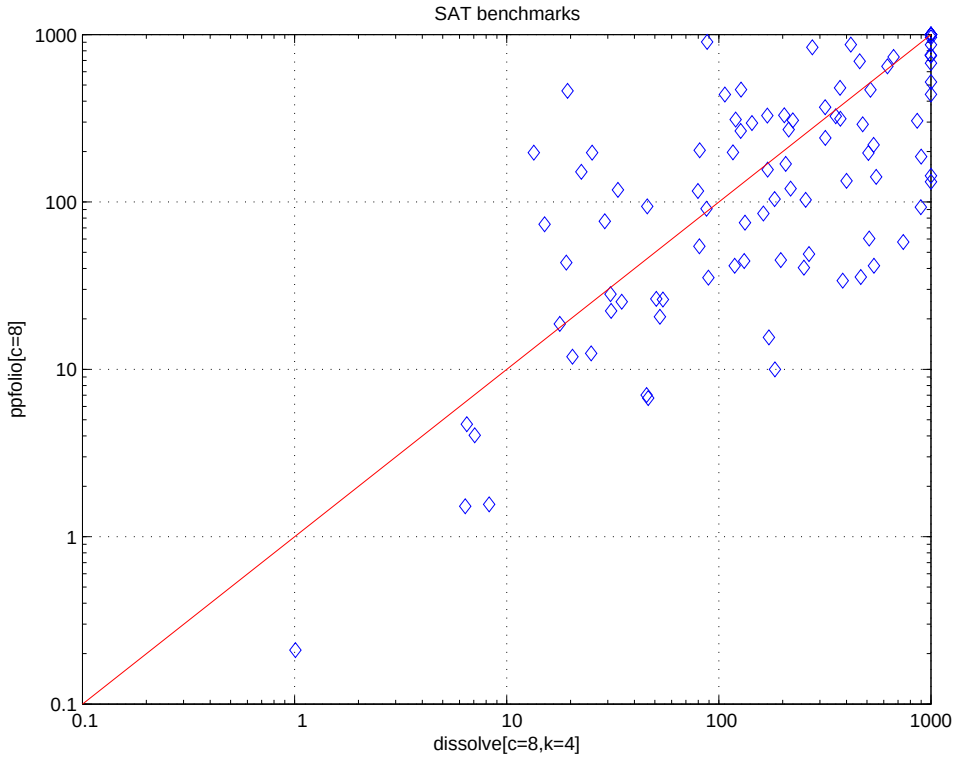


Figure 10.3: Log-log scatter plot comparing the running time (in seconds) of ppfolio$[c = 8]$ and `DiSSolve`$[c = 2^3, k = 4]$ on 150 satisfiable (SAT) benchmarks.

10.2.3 `DiSSolve` on the Cloud

The experiment in this section was designed to answer the following question:

- How well does `DiSSolve` perform when used as a distributed solver deployed on hundreds of machines on the cloud?

The benchmarks used in this experiment consisted of 10 formulas in the application track of SAT-COMP 2013 that were not solved by any parallel solver running on a 32-core machine within a time limit of 10000 seconds (SAT-COMP'13, 2013). Out of the 10 benchmarks, 8 were unsatisfiable instances.

In this experiment, `DiSSolve` was given 128 and 256 compute engines. The compute engines were deployed on the Google Compute Engine cloud; each compute engine was of the `n1-standard-1` machine type, which is a standard 1 CPU machine type with 1 virtual CPU

	Benchmark	DiSSolve $[c = 2^3, k = 4]$
1.	bivium-39-200-0s0-0x28df. . .ad91-98.cnf	450.11
2.	bivium-39-200-0s0-0x53e7. . .a300-63.cnf	802.99
3.	bivium-39-200-0s0-0x5fa9. . .80d7-30.cnf	634.16
4.	bivium-39-200-0s0-0xdcfb. . .b174-43.cnf	460.78
5.	dated-5-13-u.cnf	562.26
6.	hitag2-7-60-0-0x5f8ec0ffa4b15c6-25.cnf	875.06
7.	hitag2-7-60-0-0xe8fa35372ed37e2-80.cnf	729.38
8.	hitag2-7-60-0-0xe97b5f1bee04d70-47.cnf	412.08
9.	hitag2-8-60-0-0xa3b8497b8aad6d7-42.cnf	690.44
10.	hitag2-8-60-0-0xb2021557d918860-94.cnf	931.58
11.	hitag2-8-60-0-0xdcdbc8bf368ee73-37.cnf	683.33
12.	hitag2-10-60-0-0xa360966c6eb75c4-62.cnf	751.74
13.	hitag2-10-60-0-0x8edc44db7837bbf-65.cnf	803.38
14.	hitag2-10-60-0-0xac23f1205f76343-96.cnf	983.64
15.	hitag2-10-60-0-0xb7b72dfef34c17b-39.cnf	579.67
16.	hitag2-10-60-0-0xdf7fa6426edec07-17.cnf	432.34
17.	hitag2-10-60-0-0xe14721bd199894a-99.cnf	479.23
18.	hitag2-10-60-0-0xe6754daf48162bf-46.cnf	638.82
19.	hitag2-10-60-0-0xfee9637399d85a2-78.cnf	636.40

Table 10.1: Run time, in seconds, for DiSSolve on an 8-core machine. The number of compute engines (or cores) available to DiSSolve was 8; the value of k was chosen to be 4. (The names of some of the benchmarks have been truncated due to space constraints.)

	Benchmark	DiSSolve $[c = 2^7, k = 8]$	DiSSolve $[c = 2^8, k = 9]$
1.	k_unsat.cnf	719.58	586.46
2.	ctl_3791_556_unsat.cnf	965.67	799.83
3.	ctl_4291_567_5_unsat.cnf	408.96	270.23
4.	arcfour_initialPermutation_5_32.cnf	533.58	503.17
5.	arcfour_initialPermutation_6_24.cnf	685.66	576.59

Table 10.2: Run time, in seconds, for DiSSolve when run on the cloud. c denotes the number of compute engines available to DiSSolve, and k is the number of splitting variables.

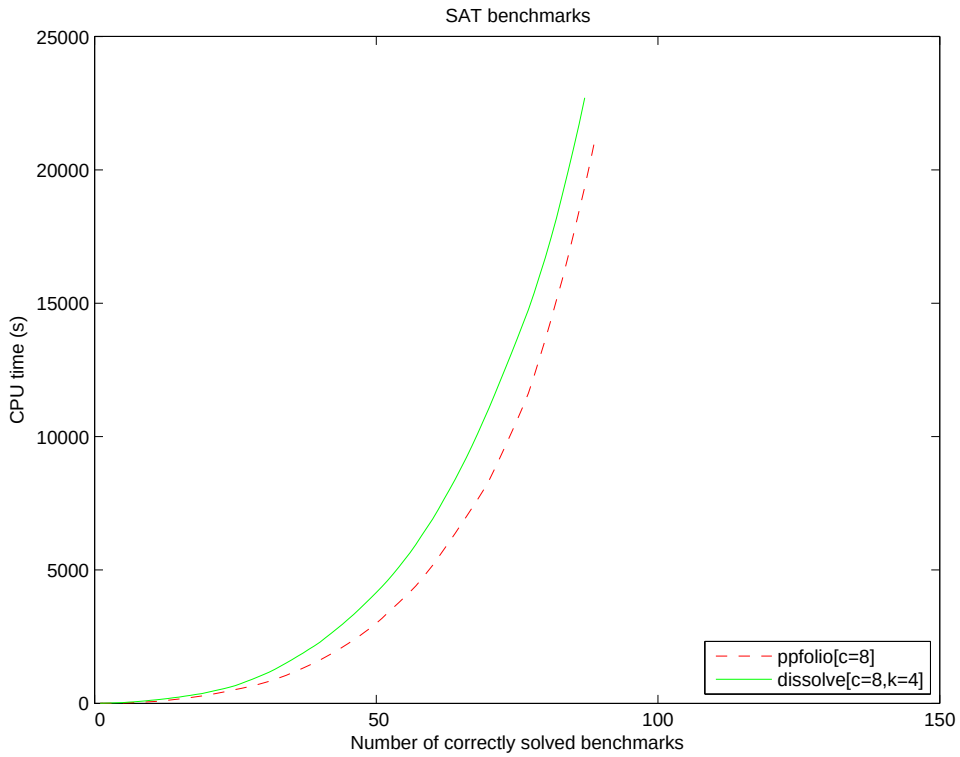


Figure 10.4: Cactus plot comparing the running time (in seconds) of ppfolio$[c = 8]$ and `DiSSolve`$[c = 2^3, k = 4]$ on 150 unsatisfiabile (UNSAT) benchmarks.

and 3.75 GB of memory available on the Google Compute Engine service. The time limit per benchmark for `DiSSolve` was 1000 seconds. `DiSSolve` was able to correctly prove unsatisfiability for 5 out of the 10 benchmarks. Table 10.2 lists the running times for the individual benchmarks. The second and third columns in Table 10.2 list the time taken by `DiSSolve` when using 128 and 256 compute engines, respectively.

10.3 Generalization

In this section, I first describe a natural extension of `DiSSolve` from SAT to SMT. I then describe the DiSSolveSplit function using abstract-interpretation terminology.

The `DiSSolve` algorithm described in Section 10.1 can be naturally generalized to handle richer logics by replacing the SAT solver used to implement `Deduce` on line 4 with an SMT

Algorithm 22: AbstractDissolveSplit(φ, A)

```
1 k = |V|
2 {A_1, A_2, ..., A_n} ← AbstractDilemmaAssumptions()
3 for i ∈ {1, 2, ..., n} do
4     A'_i ← Deduce^(φ, A ⊓ A_i)
5 A' ← A ⊓ A'_0 ⊓ A'_1 ⊓ A'_2 ... ⊓ A'_{2^k − 1}
6 return A'
```

solver. Consequently, the set of clauses returned would not be over propositional literals, but would instead contain literals of the particular theory. However, the algorithm as stated would still apply to this new setting; in particular, the `UnionClauses` method could ignore the theory interpretation of the literals, and the DiSSolveSplit method would remain sound. Furthermore, instead of allowing arbitrary literals of the theory, we could restrict the set of literals to a fixed set of predicates. In other words, we could use an abstract domain of clauses involving a fixed set of predicates (which is one way to implement the abstract domain of full predicate abstraction).

The implementation of the `DiSSolve` algorithm described in Section 10.1 can be viewed as using the *m-clausal abstract domain*; that is, an abstract domain of clauses of length up to m. The meet of two abstract values is the union of the sets of clauses that correspond to the abstract values.

Algorithm 22 presents the DiSSolveSplit function of Algorithm 21 using abstract-interpretation terminology. An abstract value A from an abstract domain $\mathcal{A}$ plays the role of the set of clauses C used in Algorithm 21. The function `AbstractDilemmaAssumptions` on line 2 returns a set of n abstract values A_i such that $\bigcup_i \gamma(A_i) \supseteq \gamma(A)$. The function $\widehat{\text{Deduce}}$ must return an abstract value A'_i such that $\varphi \Rightarrow \widehat{\gamma}(A'_i)$ (line 4).[1] Finally, on line 5, the abstract values returned by each call to $\widehat{\text{Deduce}}$ are merged by performing a meet ($\sqcap$).

Figures 10.5 and 10.6 illustrate the two distinct processes that occur in the calls to $\widehat{\text{Deduce}}$ on line 4, and shed light on the behavior of AbstractDissolveSplit in terms of the lattice of the abstract domain $\mathcal{A}$. Assume that `AbstractDilemmaAssumptions` returns two abstract values

[1] Note that, unlike Algorithm 18 and Figure 6.3(b), where $A'_i \sqsubseteq A_i$, at line 4 of Algorithm 22 $A'_i \sqsubseteq A_i$ does not hold, in general. The requirement on A'_i is $\varphi \Rightarrow \widehat{\gamma}(A'_i)$.

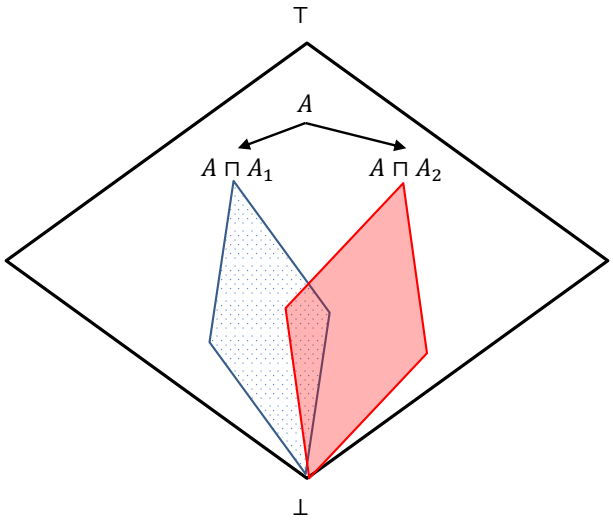


Figure 10.5: Each call to $\widehat{\text{Deduce}}$ in AbstractDissolveSplit works on a different, though not necessarily disjoint, portion of the search space. The dot-shaded sub-lattice (on the left) is searched for models of φ that satisfy the (abstract) assumption A_1. The solid-filled sub-lattice (on the right) is searched for models of φ that satisfy the (abstract) assumption A_2.

A_1 and A_2. Figure 10.5 illustrates the different, though not necessarily disjoint, search spaces, shown as different shaded sub-lattices, that are explored by the calls to $\widehat{\text{Deduce}}(\varphi, A \sqcap A_1)$ and $\widehat{\text{Deduce}}(\varphi, A \sqcap A_2)$ in the loop on lines 3–4 in Algorithm 22. Specifically, $\widehat{\text{Deduce}}(\varphi, A \sqcap A_1)$ restricts its search to models of φ that satisfy the (abstract) assumption A_1, while $\widehat{\text{Deduce}}(\varphi, A \sqcap A_2)$ restricts its search to models of φ that satisfy the (abstract) assumption A_2. Alternatively, in the case of an SMA-based solver, one can think about the two calls to $\widehat{\text{Deduce}}$ as trying to prove different goals. In particular, $\widehat{\text{Deduce}}(\varphi, A \sqcap A_1)$ is trying to show that φ implies $\neg\widehat{\gamma}(A_1)$, while $\widehat{\text{Deduce}}(\varphi, A \sqcap A_2)$ is trying to show that φ implies $\neg\widehat{\gamma}(A_2)$.

Figure 10.6 illustrates how each call to $\widehat{\text{Deduce}}$ infers (possibly different) globally true abstract values A'_1 and A'_2, each starting with the currently known abstract value A. Though the calls to $\widehat{\text{Deduce}}$ work over different search spaces, the space of deductions is common to both calls. For instance, Figure 10.6 illustrates how $\widehat{\text{Deduce}}(\varphi, A \sqcap A_1)$ deduces the abstract values A_{11}, A_{12}, and A'_1. These abstract values satisfy $A \sqsupseteq A_{11} \sqsupseteq A_{12} \sqsupseteq A'_1$ and $\varphi \Rightarrow \widehat{\gamma}(A), \varphi \Rightarrow \widehat{\gamma}(A_{11}), \varphi \Rightarrow \widehat{\gamma}(A_{12}), \varphi \Rightarrow \widehat{\gamma}(A'_1)$. The call $\widehat{\text{Deduce}}(\varphi, A \sqcap A_2)$ follows a different se-

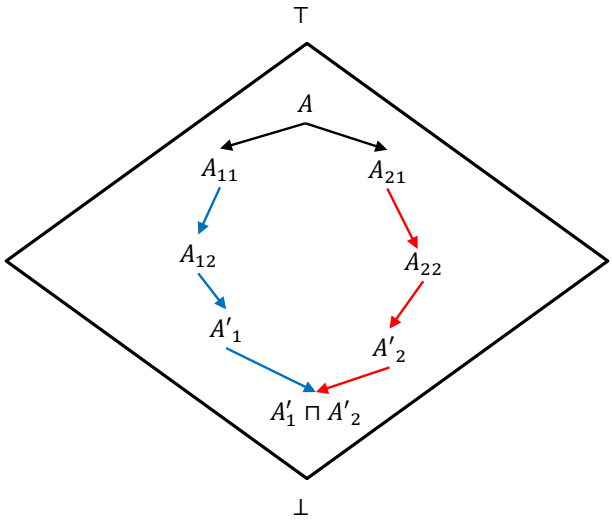


Figure 10.6: Each call to $\widehat{\text{Deduce}}$ in AbstractDissolveSplit infers (possibly different) globally true abstract values A'_1 and A'_2, where $\varphi \Rightarrow \widehat{\gamma}(A'_1)$ and $\varphi \Rightarrow \widehat{\gamma}(A'_2)$. These two branches are then merged by performing a meet.

quence of deductions to reach the abstract value A'_2, where $\varphi \Rightarrow \widehat{\gamma}(A'_2)$. Finally, the two branches of the proof are then merged by performing the meet of A'_1 and A'_2.

10.4 Related Work

This section reflects only a small portion of the literature on sequential and parallel SAT solvers. The introduction of this chapter already provided a qualitative comparison of `DiSSolve` with modern parallel SAT solvers based on divide-and-conquer and parallel portfolio approaches. Hamadi and Wintersteiger (2012) present an overview of parallel SAT solvers.

Katsirelos et al. (2013) explore bottlenecks to parallelization of SAT solvers. Their findings suggest that efficient parallelization of SAT is not merely a matter of designing the right clause-sharing heuristic. They suggest exploring solvers based on other proof systems that produce more parallelizable refutations. This observation provided motivation for exploring a technique that does not fall into the divide-and-conquer and parallel-portfolio approaches.

SatX10 (Bloom et al., 2012) is a framework for implementing parallel SAT solvers based on the parallel execution language X10. The implementation of `DiSSolve` is written in the go programming language (Go, 2014). The concurrency features built into go greatly simplified the design and implementation of `DiSSolve`.

Audemard et al. (2012) explore techniques for clause exchanges within a parallel SAT solver that incorporate the LBD heuristic developed for the Glucose sequential solver. They incorporate these techniques in Penelope, a portfolio-based parallel SAT solver.

Hamadi et al. (2011) describe a lazy-decomposition technique for distributing formulas, which is based on Craig interpolation. Instead of partitioning the search space, Hamadi et al. (2011) partition the clauses of the problem formula. Such a partitioning technique is especially useful when the input formula is too large to fit in main memory. This approach is orthogonal to the search-space decomposition technique used in `DiSSolve`.

The technique of choosing partition variables as a weighted sum of the decision variables returned by `Deduce` used in `DiSSolve` can also be found in Martins et al. (2010).

Currently, `DiSSolve` maintains a fixed bound on the number of clauses maintained, and the number of clauses returned by each call to `Deduce`. Hamadi et al. (2009a) described a dynamic control-based technique for guiding clause sharing.

10.5 Chapter Notes

Nick Kidd helped with the implementation of `DiSSolve` in the go programming language. He also directed me to the book *The Wisdom of Crowds* from which the quotation for this chapter was taken.

I had a great time learning how to use The Cloud.

Chapter 11

Satisfiability Modulo Abstraction for Separation Logic with Linked Lists

This chapter describes a sound procedure for checking the unsatisfiability of formulas in a fragment of separation logic. The procedure is designed using concepts from abstract interpretation, and is thus a Satisfiability Modulo Abstraction (SMA) solver.

Separation logic (Reynolds, 2002) is an expressive logic for reasoning about heap-allocated data structures in programs. It provides a mechanism for concisely describing program states by explicitly localizing facts that hold in separate regions of the heap. In particular, a "separating conjunction" ($\varphi_1 * \varphi_2$) asserts that the heap can be split into two disjoint regions ("heaplets") in which φ_1 and φ_2 hold, respectively (Reynolds, 2002). A "septraction" ($\varphi_1 \mathrel{-\!\circledast} \varphi_2$) asserts that a heaplet h can be extended by a disjoint heaplet h_1 in which φ_1 holds, to create a heaplet $h_1 \cup h$ in which φ_2 holds (Vafeiadis and Parkinson, 2007). The $-\!\circledast$ operator is sometimes called *existential magic wand*, because it is the DeMorgan-dual of the magic-wand operator "$-\!*$" (also called separating implication); i.e., $\varphi_1 \mathrel{-\!\circledast} \varphi_2$ iff $\neg(\varphi_1 \mathrel{-\!*} \neg\varphi_2)$.

The use of separation logic in manual, semi-automated, and automated verification tools is a burgeoning field (Berdine et al., 2005; Distefano et al., 2006; Magill et al., 2007; Distefano and Parkinson, 2008; Dudka et al., 2013). Most of these incorporate some form of automated reasoning for separation logic, but only limited fragments of separation logic are typically handled.

Given a formula φ, the unsatisfiability checker sets up an appropriate abstract domain that is tailored for representing information about the meanings of subformulas of φ. It uses an abstract domain of shape graphs (Sagiv et al., 2002) to represent a set of heap structures. The proof calculus that is presented performs a bottom-up evaluation of φ, using a particular shape-graph interpretation. It computes an abstract value that over-approximates the set of satisfying models of φ. If the over-approximation is the empty set of shape graphs, then φ is unsatisfiable. If φ is satisfiable, then the procedure reports a set of abstract models.

The unsatisfiability checker presented in this chapter is the first to apply the SMA approach to a fragment of separation logic. One of the main advantages of the SMA approach is that it is able to reuse abstract-interpretation machinery to implement decision procedures. In Chapter 6, for instance, the polyhedral abstract domain—implemented in PPL (Bagnara et al., 2008)—is used to implement a decision procedure for the logic of linear rational arithmetic. In this chapter, the abstract domain of shapes—implemented in TVLA (Sagiv et al., 2002)—is used in a novel way to implement a sound procedure for checking unsatisfiability of formulas in separation logic. The challenge was to instantiate the parametric framework of TVLA to precisely represent the literals and capture the spatial constraints of our fragment of separation logic.

The nature of this unsatisfiability checker is thus much different from other decision procedures for fragments of separation logic that I are aware of. Most previous decision procedures are *proof-theoretic*. In some sense, our method is *model-theoretic*: it uses explicitly instantiated sets of 3-valued structures to represent overapproximations of the models of subformulas.

The fragment of separation logic discussed in this chapter includes points-to assertions ($x \mapsto y$), acyclic-list-segment assertions (**ls**(x, y)), empty-heap assertions (**emp**), and their negations; separating conjunction; septraction; logical-and; and logical-or. The fragment considered only allows negation at the leaves of a formula (Section 11.1.1), but still contains formulas that lie outside of previously considered fragments (Berdine et al., 2004; Pérez and Rybalchenko, 2011; Park et al., 2013; Lee and Park, 2014; Hou et al., 2014). The procedure can prove *validity* of

implications of the form

$$\psi \Rightarrow (\varphi_i \wedge \bigwedge_j \psi_j \mathrel{-\!\!*} \varphi_j), \tag{11.1}$$

where φ_i and φ_j are formulas that contain only $\wedge$, $\vee$, and positive or negative occurrences of **emp**, points-to, or **ls** assertions; and ψ and ψ_j are arbitrary formulas in the logic fragment defined in Section 11.1.1. Consequently, we believe that ours is the first procedure that can prove the validity of formulas that contain both **ls** and the magic-wand operator $-\!\!*$. Furthermore, the procedure is able to prove *unsatisfiability* of interesting classes of formulas that are outside of previously considered fragments, including (i) formulas that use *conjunctions of separating-conjunctions with* ***ls*** or *negations below separating-conjunctions*, such as

$$(\mathbf{ls}(a1, a2) * \mathbf{ls}(a2, a3)) \wedge (\neg\mathbf{emp} * \neg\mathbf{emp})$$

$$\wedge (a1 \mapsto e1 * \mathbf{True}) \wedge e1 = \mathtt{nil},$$

and (ii) formulas that *contain both* ***ls*** *and septraction (*$-\!\!\circledast$*)*, such as

$$(a3 \mapsto a4 \mathrel{-\!\!\circledast} \mathbf{ls}(a1, a4)) \wedge (a3 = a4 \vee \neg\mathbf{ls}(a1, a3)).$$

The former are useful for describing overlaid data structures; the latter are useful in dealing with interference effects when using rely/guarantee reasoning to verify programs with fine-grained concurrency (Vafeiadis and Parkinson, 2007; Calcagno et al., 2007).

The contributions of this chapter include the following:

- I show how a canonical-abstraction domain can be used to overapproximate the set of heaps that satisfy a separation-logic formula (Section 11.1).
- I present rules for calculating the overapproximation of a separation-logic formula for a fragment of separation logic that consists of separating conjunction, septraction, logical-and, and logical-or (Section 11.3).
- The procedure is parameterized by a shape abstraction, and can be instantiated to handle (positive or negative) literals for points-to or acyclic-list-segment assertions—and hence can prove the validity of implications of the kind shown in formula (11.1) (Section 11.3).

Section 11.2 illustrates the key concepts used in the unsatisfiability-checking procedure. The procedure is implemented in a tool called SMASLTOV (Satisfiability Modulo Abstraction for Separation Logic ThrOugh Valuation), which is available at `https://www.github.com/smasltov-team/SMASLTOV`. I evaluated SMASLTOV on a set of formulas taken from the literature (Section 11.4). To the best of my knowledge, SMASLTOV is able to establish the unsatisfiability of formulas that cannot be handled by previous approaches.

11.1 Separation Logic and Canonical Abstraction

In this section, I provide background on separation logic and introduce the separation-logic fragment considered in the thesis. I then show how a canonical-abstraction domain can be used to approximate the set of models that satisfy a separation-logic formula.

11.1.1 Syntax and Semantics of Separation Logic

Formulas in our fragment of separation logic (SL) are defined as follows:

$$\varphi ::= \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi * \varphi \mid \varphi \mathrel{-\!\!\circledast} \varphi \mid \mathit{atom} \mid \neg \mathit{atom}$$

$$\mathit{atom} ::= \mathbf{true} \mid \mathbf{emp} \mid x = y \mid x \mapsto y \mid \mathbf{ls}(x, y)$$

The set of literals, denoted by Literals, is the union of the positive and negative atoms of SL.

The semantics of SL is defined with respect to memory "statelets", which consist of a *store* and a *heaplet*. A store is a function from variables to values; a heaplet is a finite function from locations to locations.

$$\mathit{Val} \stackrel{\text{def}}{=} \mathit{Loc} \uplus \{\mathtt{nil}\} \qquad \mathit{Store} \stackrel{\text{def}}{=} \mathit{Var} \to \mathit{Val}$$

$$\mathit{Heaplet} \stackrel{\text{def}}{=} \mathit{Loc} \rightharpoonup_{\mathit{fin}} \mathit{Val} \qquad \mathit{Statelet} \stackrel{\text{def}}{=} \mathit{Store} \times \mathit{Heaplet}$$

Loc and *Var* are disjoint countably infinite sets, neither of which contain `nil`. *Loc* represents heap-node addresses. The domain of h, $\mathrm{dom}(h)$, represents the set of addresses of cells in the heaplet. Two heaplets h_1, h_2 are *disjoint*, denoted by $h_1 \# h_2$, if $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$. Given two disjoint heaplets h_1 and h_2, $h_1 \cdot h_2$ denotes their disjoint union $h_1 \uplus h_2$. A *statelet* is denoted by a pair (s, h).

$$
\begin{array}{lll}
(s,h) \models \varphi_1 \wedge \varphi_2 & \text{iff} & (s,h) \models \varphi_1 \text{ and } (s,h) \models \varphi_2 \\
(s,h) \models \varphi_1 \vee \varphi_2 & \text{iff} & (s,h) \models \varphi_1 \text{ or } (s,h) \models \varphi_2 \\
(s,h) \models \varphi_1 * \varphi_2 & \text{iff} & \exists h_1, h_2.\ h_1 \# h_2 \text{ and } h_1 \cdot h_2 = h \text{ and} \\
 & & (s,h_1) \models \varphi_1 \text{ and } (s,h_2) \models \varphi_2 \\
(s,h) \models \varphi_1 \mathrel{-\!\!\circledast} \varphi_2 & \text{iff} & \exists h_1.\ h_1 \# h \text{ and } (s,h_1) \models \varphi_1 \text{ and} \\
 & & (s, h_1 \cdot h) \models \varphi_2 \\
(s,h) \models \neg \mathit{atom} & \text{iff} & (s,h) \not\models \mathit{atom} \\
(s,h) \models \mathbf{true} & \text{iff} & \mathbf{true} \\
(s,h) \models \mathbf{emp} & \text{iff} & \mathrm{dom}(h) = \emptyset \\
(s,h) \models x = y & \text{iff} & s(x) = s(y) \\
(s,h) \models x \mapsto y & \text{iff} & \mathrm{dom}(h) = \{s(x)\} \text{ and } h(s(x)) = s(y) \\
(s,h) \models \mathbf{ls}(x,y) & \text{iff} & \text{if } s(x) = s(y) \text{ then } \mathrm{dom}(h) = \emptyset, \\
 & & \text{else there is a nonempty acyclic path from} \\
 & & s(x) \text{ to } s(y) \text{ in } h, \text{ and this path contains} \\
 & & \text{all heap cells in } h
\end{array}
$$

Figure 11.1: Satisfaction of an SL formula φ with respect to statelet (s,h).

Satisfaction of an SL formula φ with respect to statelet (s,h) is defined in Figure 11.1. Furthermore, in this chapter, we consider a formula to be satisfiable only if it is satisfiable over an *acyclic* heap. $[\![\varphi]\!]$ denotes the set of statelets that satisfy φ: $[\![\varphi]\!] \stackrel{\text{def}}{=} \{(s,h) \mid (s,h) \models \varphi\}$.

11.1.2 2-Valued Logical Structures

We model full states—not statelets—by 2-*valued logical structures*. A logical structure provides an interpretation of a vocabulary Voc $= \{eq, p_1, \ldots, p_n\}$ of predicate symbols (with given arities). Voc_k denotes the set of k-ary symbols.

Definition 11.1. A **2-valued logical structure** S over Voc is a pair $S = \langle U, \iota \rangle$, where U is the set of **individuals**, and ι is the **interpretation**. Let $\mathbb{B} = \{0,1\}$ be the domain of truth values. For $p \in \text{Voc}_i$, $\iota(p)\colon U^i \to \mathbb{B}$. We assume that $eq \in \text{Voc}_2$ is the identity relation: (i) for all $u \in U$, $\iota(eq)(u,u) = 1$, and (ii) for all $u_1, u_2 \in U$ such that u_1 and u_2 are distinct individuals, $\iota(eq)(u_1,u_2) = 0$.

The set of 2-valued logical structures over Voc is denoted by 2-STRUCT[Voc]. ■

A concrete state is modeled by a 2-valued logical structure over a fixed vocabulary $\mathcal{C}$ of *core predicates*. Core predicates are part of the underlying semantics of the linked structures that make

Predicate	Intended Meaning
$eq(v_1, v_2)$	Do v_1 and v_2 denote the same memory cell?
$q(v)$	Does pointer variable q point to memory cell v?
$n(v_1, v_2)$	Does the n-field of v_1 point to v_2?

Table 11.1: Core predicates used when representing states made up of acyclic linked lists.

up the states of interest. Table 11.1 lists the core predicates that are used when representing states made up of acyclic linked lists.

Without loss of generality, vocabularies exclude constant and function symbols. Constant symbols can be encoded via unary predicates, and n-ary functions via $n + 1$-ary predicates. In both cases, we need *integrity rules*—i.e., global constraints that restrict the set of structures considered to the ones that we intend. The set of unary predicates, Voc_1, always contains predicates that encode the variables of the formula. In a minor abuse of notation, we overload "x" to denote both the name of variable x and the unary predicate $x(\cdot)$ that encodes the variable. The binary predicate $n \in \text{Voc}_2$ encodes list-node linkages. In essence, the following integrity rules restrict each $x \in \mathit{Var} \subseteq \text{Voc}_1$ to serve as a constant, and restrict relation n to encode a partial function:

$$\text{for each } x \in \mathit{Var}, \forall v_1, v_2 : x(v_1) \wedge x(v_2) \quad \Rightarrow \quad eq(v_1, v_2)$$

$$\forall v_1, v_2, v_3 : n(v_3, v_1) \wedge n(v_3, v_2) \quad \Rightarrow \quad eq(v_1, v_2)$$

11.1.3 Connecting 2-Valued Logical Structures and SL Statelets

We use unary *domain predicates*, typically denoted by d, d', $d_1, \ldots, d_k \in \text{Voc}_1$, to pick out regions of the heap that are of interest in the state that a logical structure models. The connection between 2-valued logical structures and SL statelets is formalized by means of the operation $S|_{(d,\cdot)}$, which

performs a projection of structure S with respect to a domain predicate d:

$$S|_{(d,\cdot)} \stackrel{\text{def}}{=} (s, h), \text{where}$$

$$s = \left(\begin{array}{ll} & \{(p,u) \mid p \in \mathit{Var}^S, u \in U^S, \text{and } p(u)\} \\ \cup & \{(q, \texttt{nil}) \mid q \in \mathit{Var}^S \text{ and } \neg\exists v : q(v)\} \end{array} \right) \tag{11.2}$$

$$h = \{(u_1, u_2) \mid u_1, u_2 \in U^S, d(u_1), \text{and } n(u_1, u_2)\}. \tag{11.3}$$

The subscript "$(d, \cdot)$" serves as a reminder that in Equation (11.3), only u_1 needs to be in the region defined by d. We lift the projection operation to apply to a set SS of 2-valued logical structures as follows:

$$\text{SS}|_{(d,\cdot)} \stackrel{\text{def}}{=} \{S|_{(d,\cdot)} \mid S \in \text{SS}\}.$$

11.1.4 Representing Sets of SL Statelets using Canonical Abstraction

In the framework of Sagiv et al. (2002) for logic-based abstract-interpretation, 3-*valued logical structures* provide a way to overapproximate possibly infinite sets of 2-valued structures in a finite way that can be represented in a computer. The application of Equations (11.2) and (11.3) to 3-valued structures means that the abstract-interpretation machinery developed by Sagiv et al. provides a finite way to overapproximate a possibly infinite set of SL statelets.

In 3-valued logic, a third truth value, denoted by $1/2$, represents uncertainty. The set $\mathbb{T} \stackrel{\text{def}}{=} \mathbb{B} \cup \{1/2\}$ of 3-valued truth values is partially ordered "$l \sqsubset 1/2$ for $l \in \mathbb{B}$". The values 0 and 1 are *definite* values; $1/2$ is an *indefinite* value.

Definition 11.2. A **3-valued logical structure** $S = \langle U, \iota \rangle$ is almost identical to a 2-valued structure, except that ι maps each $p \in \text{Voc}_i$ to a 3-valued function $\iota(p) \colon U^i \to \mathbb{T}$. In addition, (i) for all $u \in U$, $\iota(\mathit{eq})(u, u) \sqsupseteq 1$, and (ii) for all $u_1, u_2 \in U$ such that u_1 and u_2 are distinct individuals, $\iota(\mathit{eq})(u_1, u_2) = 0$. (An individual u for which $\iota(\mathit{eq})(u, u) = 1/2$ is called a **summary individual**.)

The set of 3-valued logical structures over Voc is denoted by 3-STRUCT[Voc]. Note that 2-STRUCT[Voc] $\subsetneq$ 3-STRUCT[Voc]. ∎

As we will see below, a summary individual may represent more than one individual from certain 2-valued structures.

A 3-valued structure can be depicted as a directed graph with individuals as graph nodes (see Figure 11.2). A summary individual is depicted with a double-ruled border. A unary predicate $p \in \mathit{Var}$ is represented in the graph by having an arrow from the predicate name p to all nodes of individuals u for which $\iota(p)(u) \sqsupseteq 1$. An arrow between two nodes indicates that a binary predicate holds for the corresponding pair of individuals. (To reduce clutter, in the figures in this chapter, the only binary predicate shown is the predicate $n \in \mathrm{Voc}_2$.) A predicate value of $1/2$ is indicated by a dotted arrow, a value of 1 by a solid arrow, and a value of 0 by the absence of an arrow. A unary predicate $p \in (\mathrm{Voc}_1 - \mathit{Var})$ is listed, with its value, inside the node of each individual u for which $\iota(p)(u) \sqsupseteq 1$. A nullary predicate is displayed in a rectangular box.

To define a suitable abstraction of 2-valued logical structures, we start with the notion of structure embedding (Sagiv et al., 2002):

Definition 11.3. Given $S = \langle U, \iota \rangle$ and $S' = \langle U', \iota' \rangle$, two 3-valued structures over the same vocabulary Voc, and $f : U \to U'$, a surjective function, f **embeds** S **in** S', denoted by $S \sqsubseteq^f S'$, if for all $p \in \mathrm{Voc}$ and $u_1, \ldots, u_k \in U$,

$$\iota(p)(u_1, \ldots, u_k) \sqsubseteq \iota'(p)(f(u_1), \ldots, f(u_k))$$

If, in addition,

$$\iota'(p)(u'_1, \ldots, u'_k) = \bigsqcup_{u_1,\ldots,u_k \in U, s.t. f(u_i)=u'_i, 1 \le i \le k} \iota(p)(u_1, \ldots, u_k)$$

then S' is the **tight embedding of** S **with respect to** f, denoted by $S' = f(S)$. (Note that we overload f to also mean the mapping on structures $f : \text{3-STRUCT}[\mathrm{Voc}] \to \text{3-STRUCT}[\mathrm{Voc}]$ induced by $f : U \to U'$.) ■

Intuitively, $f(S)$ is obtained by merging individuals of S and by defining the valuation of predicates accordingly (in the most precise way). The relation $\sqsubseteq^{\mathrm{id}}$, which will be denoted by $\sqsubseteq$, is the natural information order between structures that share the same universe. One has $S \sqsubseteq^f S' \Leftrightarrow f(S) \sqsubseteq^{\mathrm{id}} S'$. Henceforth, we use $S \sqsubseteq^f S'$ to mean "there exists a surjective $f : U \to U'$ such that $f(S) \sqsubseteq^{\mathrm{id}} S'$".

However, embedding alone is not enough. The challenge for representing and manipulating sets of 2-valued structures is that the universe of a structure is of *a priori* unbounded size. Consequently, we need a method that, for a 2-valued structure $\langle U, \iota \rangle \in$ 2-STRUCT[Voc], abstracts U to an abstract universe $U^\sharp$ of bounded size. The idea behind *canonical abstraction* (Sagiv et al., 2002, Section 4.3) is to choose a subset $\mathbb{A} \subseteq \text{Voc}_1$ of *abstraction predicates*, and to define an equivalence relation $\simeq_{\mathbb{A}^S}$ on U that is parameterized by the logical structure $S = \langle U, \iota \rangle \in$ 2-STRUCT[Voc] to be abstracted:

$$u_1 \simeq_{\mathbb{A}^S} u_2 \;\Leftrightarrow\; \forall p \in \mathbb{A} : \iota(p)(u_1) = \iota(p)(u_2).$$

This equivalence relation defines the surjective function $f^S_{\mathbb{A}} \colon U \to (U/\simeq_{\mathbb{A}^S})$, which maps an individual to its equivalence class. We thus have the Galois connection

$$\begin{gathered}\wp(\text{2-STRUCT[Voc]}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \wp(\text{3-STRUCT[Voc]}) \\ \alpha(X) = \{f^S_{\mathbb{A}}(S) \mid S \in X\} \\ \gamma(Y) = \{S \mid S^\sharp \in Y \wedge S \sqsubseteq^f S^\sharp\}\end{gathered}$$

where $f^S_{\mathbb{A}}$ in the definition of α denotes the tight-embedding function for logical structures induced by the node-embedding function $f^S_{\mathbb{A}} \colon U \to (U/\simeq_{\mathbb{A}^S})$. The abstraction function α is referred to as *canonical abstraction*. Note that there is an upper bound on the size of each structure $\langle U^\sharp, \iota^\sharp \rangle \in$ 3-STRUCT[Voc] that is in the image of α: $|U^\sharp| \leq 2^{|\mathbb{A}|}$—and thus the power-set of the image of α is a finite sublattice of $\wp$(3-STRUCT[Voc]).

For technical reasons, it turns out to be convenient to work with 3-valued structures other than the ones in the image of α; however, we still want to restrict ourselves to a finite sublattice of $\wp$(3-STRUCT[Voc]). With this motivation, we make the following definition (Arnold et al., 2006):

Definition 11.4. A 3-valued structure $\langle U^\sharp, \iota^\sharp \rangle \in$ 3-STRUCT[Voc] is **bounded** (with respect to abstraction predicates $\mathbb{A}$) if for every $u_1, u_2 \in U^\sharp$, where $u_1 \neq u_2$, there exists an abstraction

predicate symbol $p \in \mathbb{A} \subseteq \text{Voc}_1$ such that $\iota^\sharp(p)(u_1) = 0$ and $\iota^\sharp(p)(u_2) = 1$, or $\iota^\sharp(p)(u_1) = 1$ and $\iota^\sharp(p)(u_2) = 0$. B-STRUCT[Voc, $\mathbb{A}$] denotes the set of such structures. ■

Definition 11.4 also imposes an upper bound on the size of a structure $\langle U^\sharp, \iota^\sharp \rangle \in$ B-STRUCT[Voc, $\mathbb{A}$]—again, $|U^\sharp| \leq 2^{|\mathbb{A}|}$—and thus $\wp$(B-STRUCT[Voc, $\mathbb{A}$]) is a finite sublattice of $\wp$(3-STRUCT[Voc]). It defines the abstract domain that we use, the *abstract domain whose elements are subsets of B-STRUCT*[*Voc*, $\mathbb{A}$], which will be denoted by $\mathcal{A}$[Voc, $\mathbb{A}$]. (For brevity, we call such a domain a "*canonical-abstraction domain*", and denote it by $\mathcal{A}$ when Voc and $\mathbb{A}$ are understood.) The Galois connection we work with is thus

$$\wp(\text{2-STRUCT[Voc]}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \wp(\text{B-STRUCT[Voc, } \mathbb{A}]) = \mathcal{A}[\text{Voc}, \mathbb{A}]$$
$$\alpha(X) = \{f^S_{\mathbb{A}}(S) \mid S \in X\}$$
$$\gamma(Y) = \{S \mid S^\sharp \in Y \wedge S \sqsubseteq^f S^\sharp\}.$$

The ordering on $\wp$(B-STRUCT[Voc, $\mathbb{A}$]) $= \mathcal{A}$[Voc, $\mathbb{A}$] is the Hoare ordering: $S_1 \sqsubseteq S_2$ if for all $s_1 \in S_1$ there exists $s_2 \in S_2$ such that $s_1 \sqsubseteq^f s_2$.

11.2 Overview

This section presents two examples to illustrate the concepts that we use in the unsatisfiability-checking procedure:

- a formula that is unsatisfiable over acyclic linked lists: $(x \mapsto y) * (y \mapsto x)$
- a formula that is satisfiable over acyclic linked lists: $(x \mapsto y) \mathrel{-\!\!\circledast} \mathbf{ls}(x, z)$.

11.2.1 An Unsatisfiable Formula

Consider $\varphi \stackrel{\text{def}}{=} x \mapsto y * y \mapsto x$. We want to compute $A \in \mathcal{A}$ such that $\gamma(A)|_{(d,\cdot)} \supseteq [\![\varphi]\!]$. The key to handling the $*$ operator is to introduce two new domain predicates d_1 and d_2, which are used to demarcate the heaplets that must satisfy $\varphi_1 \stackrel{\text{def}}{=} x \mapsto y$ and $\varphi_2 \stackrel{\text{def}}{=} y \mapsto x$, respectively. We have

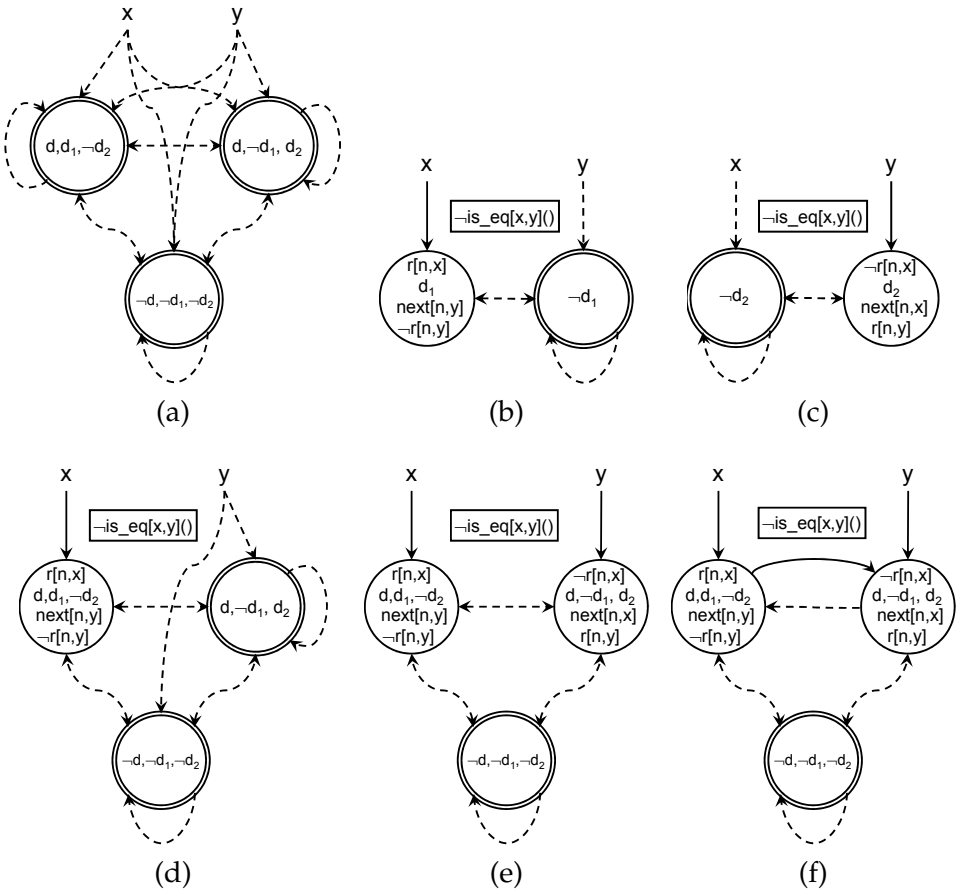


Figure 11.2: Structures that arise in the meet operation used to analyze $x \mapsto y * y \mapsto x$.

designed $\mathcal{A}$ so that there exist $A_1, A_2 \in \mathcal{A}$ such that $\gamma(A_1)|_{(d_1,\cdot)} = [\![x \mapsto y]\!]$ and $\gamma(A_2)|_{(d_2,\cdot)} = [\![y \mapsto x]\!]$, respectively. Table 11.2 describes the abstraction predicates we use. A_1 and A_2 each consist of a single 3-valued structure, shown in Figure 11.2(b) and Figure 11.2(c), respectively. Furthermore, to satisfy $\varphi_1 * \varphi_2$, d_1 and d_2 are required to be disjoint regions whose union is d. $\mathcal{A}$ also contains an abstract value, which we will call D, that represents this disjointness constraint exactly. D consists of four 3-valued structures. Figure 11.2(a) shows the "most general" of them: it represents two disjoint regions, d_1 and d_2, that partition the d region (where each of d_1 and d_2 contain at least one cell). The summary individual labeled $\neg d, \neg d_1, \neg d_2$ in Figure 11.2(a) represents a region that is disjoint from d. (See also Figure 11.6.)

Note that here and throughout the paper, for brevity the figures only show predicates that are relevant to the issue under discussion.

Meet for a Canonical-Abstraction Domain. To impose a necessary condition for $x \mapsto y * y \mapsto x$ to be satisfiable, we take the *meet* of D, A_1, and A_2: $[\![x \mapsto y * y \mapsto x]\!] \subseteq D \sqcap A_1 \sqcap A_2$. Figs. 11.2(d), (e), and (f) show some of the structures that arise in $D \sqcap A_1 \sqcap A_2$.

The meet operation in $\mathcal{A}$ is defined in terms of the greatest-lower-bound operation induced by the approximation order in the lattice B-STRUCT$[\mathrm{Voc}, \mathbb{A}]$. Arnold et al. Arnold et al. (2006) show that in general this operation is NP-complete; however, they define an algorithm based on graph matching that typically performs well in practice (Jeannet et al., 2010, §8.3). To understand some of the subtleties of meet, consider Figure 11.2(d), which shows one of the structures in $D \sqcap A_1$ (i.e., Figure 11.2(a) $\sqcap$ Figure 11.2(b)).

- From the standpoint of Figure 11.2(b), meet caused the summary individual labeled "$\neg d_1$" to be split into two summary individuals: "$\neg d, \neg d_1, \neg d_2$" and "$d, \neg d_1, d_2$".
- From the standpoint of Figure 11.2(a), meet caused the summary individual labeled "$d, d_1, \neg d_2$" to (i) become a non-summary individual, (ii) acquire the value 1 for x, $r[n, x]$, and $\mathit{next}[n, y]$, and (iii) acquire the value 0 for y and $r[n, y]$.

Figure 11.2(e) shows one of the structures in $(D \sqcap A_1) \sqcap A_2$, i.e., Figure 11.2(d) $\sqcap$ Figure 11.2(c), which causes further (formerly indefinite) elements to acquire definite values.

Arnold et al. develop a graph-theoretic notion of the possible correspondences among individuals in the bounded structures that are arguments to meet, and structure the meet algorithm around the set of possible correspondences (Arnold et al., 2006, §4.2).

Improving Precision Using Semantic-Reduction Operators. Figure 11.2(e) still contains a great deal of indefinite information because the meet operation does not take into account the integrity constraints on structures. For instance, for the structures that we use to represent states and SL statelets, we use a unary predicate $\mathit{next}[n, y]$, which holds for individuals whose n-link points to the individual that is pointed to by y. This predicate has an associated integrity constraint

$$\forall v_1, v_2.\mathit{next}[n, y](v_1) \wedge y(v_2) \Rightarrow n(v_1, v_2). \tag{11.4}$$

In particular, in Figure 11.2(e) the individual pointed to by x has $next[n, y] = 1$; however, the edge to the individual pointed to by y has the value $1/2$. Similarly, we force the procedure to consider only acyclic heaps by imposing the integrity constraint $\neg\exists v_1, v_2.n(v_1, v_2) \wedge t[n](v_2, v_1)$.

To improve the precision of the (graph-theoretic) meet, the procedure makes use of *semantic-reduction operators*. The notion of semantic reduction was introduced by Cousot and Cousot Cousot and Cousot (1979). Semantic-reduction operators are useful when an abstract domain is a lattice that has multiple elements that represent the same set of states. A semantic reduction operator ρ maps an abstract-domain element A to $\rho(A)$ such that (i) $\rho(A) \sqsubseteq A$, and (ii) $\gamma(\rho(A)) = \gamma(A)$. In other words, ρ maps A to an element that is lower in the lattice—and hence a "better" representation of $\gamma(A)$ in $\mathcal{A}$—while preserving the meaning. In our case, the semantic-reduction operations that we use convert a set of 3-valued structures *XS* into a "better" set of 3-valued structures *XS'* that describe the same set of 2-valued structures.

A semantic-reduction operator can have two effects:

1. In some structure $S \in XS$, some tuple $p(u)$ with indefinite value $1/2$ may be changed to have a definite value (0 or 1).

2. It may be determined that some structure $S \in XS$ is infeasible: i.e., $\gamma(S) = \emptyset$. In this case, S is removed from *XS*.

The effect of a precision improvement from a type-1 effect can cause a type-2 effect to occur. For instance, let u_1 and u_2 be the individuals pointed to by x and y, respectively, in Figure 11.2(e).

- Figure 11.2(f) is Figure 11.2(e) after integrity constraint (11.4) has triggered a type-1 change that improves the value of $n(u_1, u_2)$ from $1/2$ to 1.

- A type-2 rule can then determine that the structure shown in Figure 11.2(f) is infeasible. In particular, the predicate $r[n, x](v)$ means that individual v is reachable from the individual pointed to by x along n-links. The semantic-reduction rule would find that the values $x(u_1) = 1$, $n(u_1, u_2) = 1$, and $r[n, x](u_2) = 0$ represent an irreconcilable inconsistency in

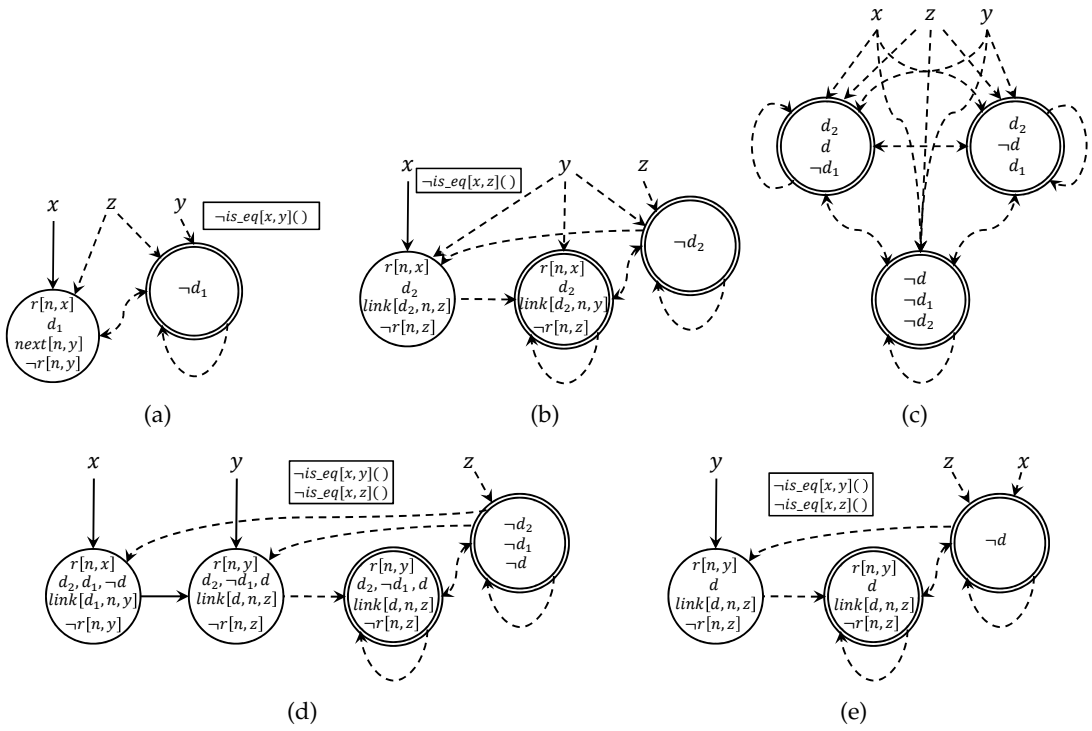


Figure 11.3: Some of the structures that arise in the meet operation used to evaluate $x \mapsto y \mathbin{-\!\circledast} \mathbf{ls}(x, z)$.

Figure 11.2(f): the first two predicate values mean that u_2 is reachable from the individual pointed to by x along n-links, which contradicts $r[n, x](u_2) = 0$.

The operation that applies type-1 and type-2 rules until no more changes are possible is called *coerce* (because it coerces XS to a better representation XS'). Sagiv et al. (Sagiv et al., 2002, §6.4) and Bogudlov et al. Bogudlov et al. (2007a,b) discuss algorithms for *coerce*.

11.2.2 A Satisfiable Formula

Consider the formula $\varphi \stackrel{\text{def}}{=} x \mapsto y \mathbin{-\!\circledast} \mathbf{ls}(x, z)$. We want to compute $A \in \mathcal{A}$ such that $\gamma(A)|_{(d,\cdot)} \supseteq [\![\varphi]\!]$. Similar to what was done in Section 11.2.1 for the $*$ operator, we introduce two new domain predicates d_1 and d_2, which are used to demarcate the heaplets that must satisfy $\varphi_1 \stackrel{\text{def}}{=} x \mapsto y$ and $\varphi_2 \stackrel{\text{def}}{=} \mathbf{ls}(x, z)$. By design, there exist $A_1, A_2 \in \mathcal{A}$ such that $\gamma(A_1)|_{(d_1,\cdot)} = [\![x \mapsto y]\!]$ and $\gamma(A_2)|_{(d_2,\cdot)} = [\![\mathbf{ls}(x, z)]\!]$, respectively. A_1 consists of the single 3-valued structure shown in

Figure 11.3(a). Figure 11.3(b) shows one of the structures in A_2; it represents an acyclic linked list from x to z whose length is greater than 1. Furthermore, to satisfy $\varphi_1 \mathrel{-\!\circledast} \varphi_2$, d and d_1 are required to be disjoint regions whose union is d_2. $\mathcal{A}$ also contains an abstract value, which we will call D, that represents this disjointness constraint exactly. D consists of four 3-valued structures. Figure 11.3(c) shows the "most general" of them: it represents two disjoint regions, d and d_1, that partition the d_2 region (where each of d and d_1 contain at least one cell). The summary individual labeled $\neg d, \neg d_1, \neg d_2$ in Figure 11.3(c) represents a region that is disjoint from d_2.

To impose a necessary condition for $x \mapsto y \mathrel{-\!\circledast} \mathbf{ls}(x, z)$ to be satisfiable, we take the *meet* of D, A_1, and A_2: $[\![x \mapsto y \mathrel{-\!\circledast} \mathbf{ls}(x, z)]\!] \subseteq D \sqcap A_1 \sqcap A_2$. Figure 11.3(d) shows one of the structures that arises in $D \sqcap A_1 \sqcap A_2$, after the semantic-reduction operators have been applied. A few points to note about this resultant structure:

- The summary individual in region d_2 present in the $\mathbf{ls}(x, z)$ structure in Figure 11.3(b) is split in Figure 11.3(d) into a singleton individual pointed to by y and a summary individual.
- The individual pointed to by x is in regions d_1 and d_2, but not d.
- The individual pointed to by y is in regions d and d_2, but not d_1.
- The variables x and y are not equal.
- All the individuals in d are reachable from y, not reachable from z, and have $\mathit{link}[d, n, z]$ true.

Figure 11.3(e) shows the structure after we have projected the heap onto the heap region d; that is, the values of the domain predicates d_1 and d_2 have been set of $1/2$ on all individuals, and all the abstraction predicates have been set to $1/2$ on all individuals not in d. In effect, this operation blurs the distinction between the region that is outside d, but in d_2, and the region that is outside of d and d_2. Note that the fact that x and y are not equal is preserved by the projection operation. This projection operation, denoted by $(\cdot)\!\downarrow^d$ in Section 11.3, serves as an abstract method for quantifier elimination.

$$\frac{}{\ell \in \text{Literals}, d \Vdash A_\ell}\ (\ell) \qquad \frac{\varphi_1, d \Vdash A_1 \qquad \varphi_2, d \Vdash A_2}{\varphi_1 \wedge \varphi_2, d \Vdash A_1 \sqcap A_2}\ (\wedge) \qquad \frac{\varphi_1, d \Vdash S_1 \qquad \varphi_2, d \Vdash A_2}{\varphi_1 \vee \varphi_2, d \Vdash A_1 \sqcup A_2}\ (\vee)$$

$$\frac{\varphi_1, d_1 \Vdash A_1 \qquad \varphi_2, d_2 \Vdash A_2}{\varphi_1 * \varphi_2, d \Vdash ([d = d_1 \cdot d_2]^\sharp \sqcap A_1 \sqcap A_2)\lightning^d}\ (*) \qquad \frac{\varphi_1, d_1 \Vdash A_1 \qquad \varphi_2, d_2 \Vdash A_2}{\varphi_1 \mathrel{-\!\!\circledast} \varphi_2, d \Vdash ([d_2 = d \cdot d_1]^\sharp \sqcap A_1 \sqcap A_2)\lightning^d}\ (\mathrel{-\!\!\circledast})$$

Figure 11.4: Rules for computing an abstract value that overapproximates the meaning of a formula in SL

.

Predicate	Intended Meaning
is_eq$[x, y]()$	Are x and y equal?
next$[n, y](v)$	The target of the n-edge from v is pointed to by y
$t[n](v_1, v_2)$	Is v_2 reachable via zero or more n-edges from v_1?
$r[n, y](v)$	$\exists v_1.y(v_1) \wedge t[n](v_1, v)$
$d(v)$	Is v in heap domain d?
link$[d, n, y](v)$	The target of the n-edge from v is either in d or is pointed to by y

Table 11.2: Voc consists of the predicates shown above, together with the ones in Table 11.1. All unary predicates are abstraction predicates; that is, $\mathbb{A} = \text{Voc}_1$.

Note that Figure 11.3(e) represents an acyclic linked-list from y to z with $x \neq y$, which is one of the models that satisfies $x \mapsto y \mathrel{-\!\!\circledast} \mathbf{ls}(x, z)$.

11.3 Proof System for Separation Logic

This section describes how we compute $A \in \mathcal{A}[\text{Voc}, \mathbb{A}]$ such that A overapproximates the satisfying models of $\varphi \in$ SL. The vocabulary Voc and abstraction predicates $\mathbb{A}$ are listed in Table 11.2.

The procedure works with judgments of the form "$\varphi, d \Vdash A$", where d is a domain predicate. The invariant maintained by the procedure is that, whenever it establishes a judgment $\varphi, d \Vdash A$, $A \in \mathcal{A}$ overapproximates φ in the following sense: $\gamma(A)|_{(d,\cdot)} \supseteq [\![\varphi]\!]$. Figure 11.4 lists the rules used for calculating $\varphi, d \Vdash A$ for $\varphi \in$ SL. Using these rules, the procedure performs a bottom-up

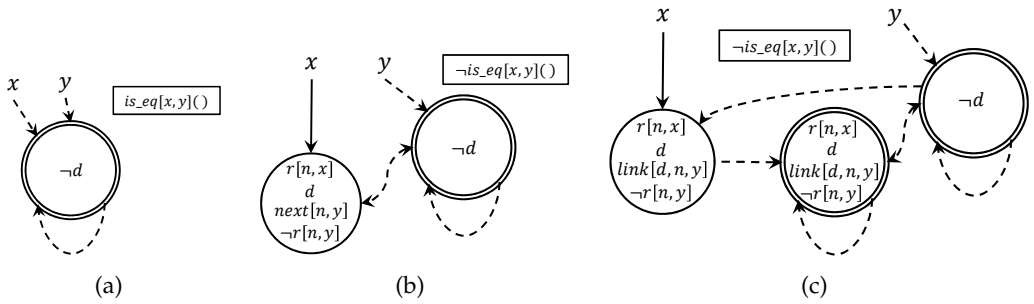


Figure 11.5: The abstract value for $\mathbf{ls}(x, y) \in \mathit{atom}$ in the canonical-abstraction domain.

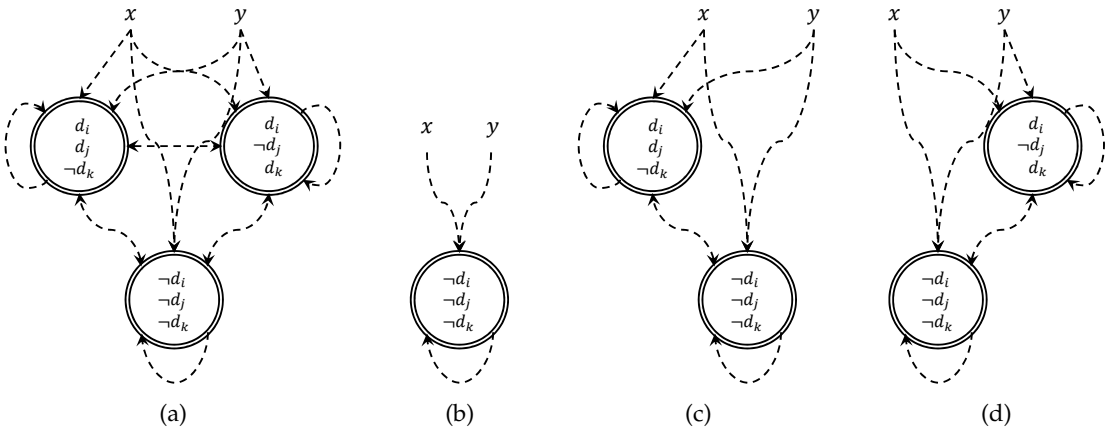


Figure 11.6: The abstract value for $[d_i = d_j \cdot d_k]^\sharp$ in the canonical-abstraction domain.

evaluation of the formula φ; if the answer is the empty set of 3-valued structures, then φ is unsatisfiable.

For each literal $\ell \in$ Literals, there is an abstract value $A_\ell \in \mathcal{A}$ such that $\gamma(A_\ell)|_{(d,\cdot)} = [\![\ell]\!]$. These A_ℓ values are used in the (ℓ)-rule of Figure 11.4. Figure 11.5 shows the abstract value $A_{\mathbf{ls}}$ used for $\mathbf{ls}(x, y)$. $A_{\mathbf{ls}}$ consists of three structures:

- Figure 11.5(a) represents the empty list from x to y. That is, $x = y$ and region d is empty.
- Figure 11.5(b) represents a singleton list from x to y. That is, $x \neq y$ and $x \neq \texttt{nil}$, and for all individuals v in d, v is reachable from x and $\mathit{link}[d, n, y](v)$ is true. (See line 6 of Table 11.2.)
- Figure 11.5(c) represents acyclic linked lists of length two or more from x to y.

Figure 11.5(b) is the single structure in $A_{x\mapsto y}$. The abstract values for atoms $x = y$, **true**, and **emp** are straightforward. We see that it is possible to represent the positive literals **True**, **emp**, $x = y$, $x \mapsto y$, and $\mathbf{ls}(x, y)$ precisely in $\mathcal{A}$; that is, we have $\gamma A_l|_{(d,\cdot)} = [\![l]\!]$. Furthermore, because the canonical-abstraction domain $\mathcal{A}$ is closed under negation (Kuncak and Rinard, 2003; Yorsh et al., 2007), we are able to represent the negative literals $x \neq y$, $\neg$**True**, $\neg$**emp**, $\neg\mathbf{ls}(x, y)$, and $\neg x \mapsto y$ precisely in $\mathcal{A}$, as well.

The rest of the rules in Figure 11.4 can be derived by reinterpreting the concrete logical operators using an appropriate abstract operator. In particular, logical-and is reinterpreted as meet, and logical-or is reinterpreted as join. Consequently, the $(\wedge)$-rule and $(\vee)$-rule are straightforward. The $(\wedge)$-rule and $(\vee)$-rule are justified by the following observation: if $\gamma(A_1)|_{(d,\cdot)} \supseteq [\![\varphi_1]\!]$ and $\gamma(A_2)|_{(d,\cdot)} \supseteq [\![\varphi_2]\!]$, then $\gamma(A_1 \sqcap A_2)|_{(d,\cdot)} \supseteq [\![\varphi_1 \wedge \varphi_2]\!]$ and $\gamma(A_1 \sqcup A_2)|_{(d,\cdot)} \supseteq [\![\varphi_1 \vee \varphi_2]\!]$.

For a given structure $A = \langle U, \iota\rangle$ and unary domain predicate d_i, we use the phrase "*individuals in* d_i" to mean the set of individuals $\{u \in U \mid \iota(d_i)(u) = 1\}$.

The $(*)$-rule computes $A \in \mathcal{A}$ such that $\gamma(A)|_{(d,\cdot)} \supseteq [\![\varphi_1 * \varphi_2]\!]$. The handling of separating conjunction $\varphi_1 * \varphi_2$ is based on the following insights:

- The domain predicates d_1 and d_2 are used to capture the heaplets h_1 and h_2 that satisfy φ_1 and φ_2, respectively. That is,
$$\gamma(A_1)|_{(d_1,\cdot)} \supseteq [\![\varphi_1]\!] \text{ and } \gamma(A_2)|_{(d_2,\cdot)} \supseteq [\![\varphi_2]\!]. \tag{11.5}$$
- $[d = d_1 \cdot d_2]^\sharp \in \mathcal{A}$ is used to express the constraint that the individuals in d_1 are disjoint from d_2, and that the individuals in d are the disjoint union of the individuals in d_1 and d_2. With only a slight abuse of notation, the meaning of $[d = d_1 \cdot d_2]^\sharp$ can be expressed as follows:
$$\gamma([d = d_1 \cdot d_2]^\sharp)|_{(d,\cdot)} \supseteq \{(s, h, h_1, h_2) \mid h_1 \# h_2 \text{ and } h_1 \cdot h_2 = h\}. \tag{11.6}$$
Figure 11.6 shows the four structures in the abstract value $[d_i = d_j \cdot d_k]^\sharp$, where d_i, d_j, and d_k are domain predicates.

- $(\cdot)\lightning^{d}$ denotes the structure that results from setting the abstraction predicates to $1/2$ for all individuals not in d, and setting all domain predicates other than d to $1/2$. In effect, this operation blurs the distinction between individuals in d_1 and d_2, and serves as an abstract method for quantifier elimination.

Using Equations (11.5) and (11.6) in the definition of $\varphi_1 * \varphi_2$, we have

$$
\begin{aligned}
&[\![\varphi_1 * \varphi_2]\!] \\
&= \{(s,h) \mid \exists h_1, h_2.\ h_1 \# h_2 \text{ and } h_1 \cdot h_2 = h \text{ and } (s,h_1) \models \varphi_1 \text{ and } (s,h_2) \models \varphi_2\} \\
&\subseteq \qquad\qquad ([d = d_1 \cdot d_2]^{\sharp} \qquad\quad \sqcap \quad A_1 \qquad\quad \sqcap \qquad A_2)\lightning^{d}
\end{aligned}
$$

The handling of septraction in the $(-\circledast)$-rule is similar to the handling of separating conjunction in the $(*)$-rule, except for the condition that $h_2 = h \cdot h_1$. This requirement is easily handled by using $[d_2 = d \cdot d_1]^{\sharp}$. Section 11.2.2 illustrates the application of the $(-\circledast)$-rule.

Theorem 11.5. *The rules in Figure 11.4 are sound; that is, if the rules in Figure 11.4 say that $\varphi, d \Vdash A$, then $\gamma(A)|_{(d,\cdot)} \supseteq [\![\varphi]\!]$.* □

The proof follows from the fact that each of the abstract operators is sound.

Discussion. As discussed in (Piskac et al., 2013, Section 4), there exist no methods that handle negations below a separating conjunction. Our fragment of separation logic admits negations at the leaves of formulas, and, thus, is the first approach that can handle formulas with negations below a separating conjunction.

It is, however, non-trivial to extend our technique to handle general negation. Let $(\cdot)^{\mathrm{c}}$ denote the set-complement operation. Let $\neg^{\#}(\cdot)$ denote the abstract negation operation; that is, $\gamma(\neg^{\#}(A)) \supseteq \gamma(A)^{\mathrm{c}}$, and $\neg^{\#}(A) \sqsupseteq \alpha(\gamma(A)^{\mathrm{c}})$. Suppose that $\gamma(A)|_{(d,\cdot)} \supseteq [\![\varphi]\!]$; in general, $\gamma(\neg^{\#}(A))|_{(d,\cdot)}$ is not guaranteed to overapproximate the models of $\neg\varphi$.

Furthermore, it is non-trivial to extend our technique to prove validity of general implications. Suppose that we would like to prove the validity of $\varphi_1 \Rightarrow \varphi_2$, where $\varphi_1, \varphi_2 \in \mathtt{SL}$. Let A_1 overap-

	emp		$x = y$		$x \mapsto y$		**ls**(x, y)		$\wedge$	$\vee$	$*$	$-\circledast$	Full
	$+$	$-$	$+$	$-$	$+$	$-$	$+$	$-$					Corpus
Group 1	1	5	8	8	13	1	19	10	22	4	12	10	23
Group 2	64	22	0	0	22	22	22	22	64	0	64	0	64
Group 3	512	218	0	0	218	218	218	218	512	0	512	512	512
Total	577	245	8	8	253	241	259	250	598	4	588	522	599

Table 11.3: Number of formulas that contain each of the SL operators in Groups 1, 2, and 3. The columns labeled "$+$" and "$-$" indicate the number of atoms occurring as positive and negative literals, respectively.

proximate the set of models of φ_1, and A_2 overapproximate the set of models of φ_2. $A_1 \sqsubseteq A_2$ does not imply $[\![\varphi_1]\!] \subseteq [\![\varphi_2]\!]$.

11.4 Experimental Evaluation

This section presents the results of our experiments to evaluate the costs and benefits of our approach. The implementation, which is called SMASLTOV, is available together with our benchmarks at `https://www.github.com/smasltov-team/SMASLTOV`. The experiments were designed to shed light on the following questions:

1. How fast is SMASLTOV?
2. How often is SMASLTOV able to determine that a formula is unsatisfiable?
3. For unsatisfiable formulas that are beyond the capabilities of other tools, is SMASLTOV actually able to prove the formulas unsatisfiable?

Setup. The unsatisfiability-checking procedure is written in OCaml; it compiles a formula to a proof DAG written in the language of ITVLA (Jeannet et al., 2010, §8). We ported the frontend of ITVLA to the latest version of TVLA Lev-Ami and Sagiv (2000) in order to make use of TVLA's enhanced speed Bogudlov et al. (2007a) and ITVLA's language features. ITVLA (i) replaces TVLA's notion of an intraprocedural control-flow graph by the more general notion of *equation system*, in which transfer functions may depend on more than one argument, and (ii) supports a

more general language in which to specify equation systems. In particular, the ITVLA language supports explicit use of the meet operator Arnold et al. (2006) for a canonical-abstraction domain. The abstract-value manipulations in the proof rules of Figure 11.4 are performed by the TVLA backend. TVLA has a significant startup cost and a smaller shutdown cost. We chose to amortize these costs by running TVLA in a batch mode, in which a single invocation of TVLA checks several separation-logic formulas.

We report trimmed means of all time measurements; that is, we made each measurement five times, discarded the highest and lowest values, and report the mean of the remaining three values. Experiments were run on a single core of a 2-processor, 4-core-per-processor 2.27 GHz Xeon computer running Red Hat Linux 6.5.

Test Suite. Our test suite consists of three groups of unsatisfiable formulas. We tested each group with a single invocation of TVLA.

- Group 1, shown in Table 11.4, was chosen to evaluate our procedure on a wide spectrum of formulas.
- Group 2 was created by replacing the Boolean variables a and b in the template $T_1 \stackrel{\text{def}}{=} \neg a \wedge \mathbf{emp} \wedge (a * b)$ with the 8 literals Literals of SL; that is, **true**, **emp**, $x \mapsto y$, $\mathbf{ls}(x, y)$, and their negations. Five of the 64 instantiations of template T_1 are shown in Table 11.5.
- Group 3 was created by replacing the Boolean variables a, b, and c in the template $T_2 \stackrel{\text{def}}{=} \mathbf{emp} \wedge a \wedge (b * (c \mathrel{-\!\!\circledast} (\mathbf{emp} \wedge \neg a)))$ with the 8 literals Literals of SL. Five of the 512 instantiations of template T_2 are shown in Table 11.6.

Templates T_1 and T_2 are based on work by Hou et al. Hou et al. (2014) on Boolean separation logic. Templates T_1 and T_2 are listed as formulas 15 and 19, respectively, in (Hou et al., 2014, Tab. 2). In total, there were 599 formulas in our test suite. Table 11.3 summarizes the characteristics of the corpus based on the occurrences of the SL operators.

In Tables 11.4, 11.5, and 11.6, a ✓ in the U-column indicates that SMASLTOV was able to prove the formula unsatisfiable; a ? indicates that the SMASLTOV was not able to prove the formula unsatisfiable.

	Formula	U	Time
(1)	$a1 \mapsto a2 \wedge \neg \mathbf{ls}(a1, a2)$	✓	0.12
(2)	$a1 \mapsto a2 * a2 \mapsto a1$	✓	0.08
(3)	$\neg \mathbf{emp} \wedge (\mathbf{ls}(a1, a2) * \mathbf{ls}(a2, a1))$	✓	0.27
(4)	$a1 \neq a2 \wedge (\mathbf{ls}(a1, a2) * \mathbf{ls}(a2, a1))$	✓	0.25
(5)	$(\mathbf{ls}(a1, a2) * \mathbf{ls}(a2, a3)) \wedge \neg \mathbf{ls}(a1, a3)$	✓	0.85
(6)	$\mathbf{ls}(a1, a2) \wedge \mathbf{emp} \wedge a1 \neq a2$	✓	0.09
(7)	$(a1 \mapsto a2 * \mathbf{True}) \wedge (a2 \mapsto a3 * \mathbf{True}) \wedge (\mathbf{True} * a3 \mapsto a1)$	✓	0.72
(8)	$(a1 \mapsto a2 \mathrel{-\!\!\circledast} \mathbf{True}) \wedge (a1 \mapsto a2 * \mathbf{True})$	✓	0.77
(9)	$(\mathbf{ls}(a1, a2) * \neg \mathbf{ls}(a2, a3)) \wedge \mathbf{ls}(a1, a3)$	✓	2.02
(10)	$\mathbf{ls}(a1, a2) \wedge \mathbf{ls}(a1, a3) \wedge \neg \mathbf{emp} \wedge a2 \neq a3$	✓	0.13
(11)	$(\mathbf{ls}(a1, a2) * \mathbf{True} * a3 \mapsto a4) \wedge (\mathbf{True} * (\mathbf{ls}(a2, a1) \wedge a2 \neq a1))$	✓	7.94
(12)	$(a1 \mapsto a2 * \mathbf{ls}(e1, e2)) \wedge (a2 \mapsto a3 * \neg \mathbf{emp}) \wedge (a3 \mapsto a1 * \neg a5 \mapsto a6 * \mathbf{True})$	✓	4.64
(13)	$(\neg \mathbf{emp} * \neg \mathbf{emp}) \wedge (a1 = \mathtt{nil} \vee a1 \mapsto e1 \vee ((a1 \mapsto e1 \wedge e1 = \mathtt{nil}) * \mathbf{True})) \wedge \mathbf{ls}(a1, a2)$	✓	0.20
(14)	$((\mathbf{ls}(a1, a2) \wedge a1 \neq a2) * (\mathbf{ls}(a2, a3) \wedge a2 \neq a3)) \wedge ((\mathbf{ls}(a4, a1) \wedge a4 \neq a1) * a1 \mapsto e1 * \mathbf{True})$	✓	1.45
(15)	$(\mathbf{ls}(a1, a2) \mathrel{-\!\!\circledast} \mathbf{ls}(a1, a2)) \wedge \neg \mathbf{emp}$	✓	0.18
(16)	$(a3 \mapsto a4 \mathrel{-\!\!\circledast} \mathbf{ls}(a1, a4)) \wedge (a3 = a4 \vee \neg \mathbf{ls}(a1, a3))$	✓	0.20
(17)	$((a2 \mapsto a3 \mathrel{-\!\!\circledast} \mathbf{ls}(a2, a4)) \mathrel{-\!\!\circledast} \mathbf{ls}(a1, a4)) \wedge \neg \mathbf{ls}(a1, a3)$	✓	0.65
(18)	$((a2 \mapsto a3 \mathrel{-\!\!\circledast} \mathbf{ls}(a2, a4)) \mathrel{-\!\!\circledast} \mathbf{ls}(a3, a1)) \wedge a2 = a4$	✓	0.62
(19)	$(a1 \mapsto a2 \mathrel{-\!\!\circledast} \mathbf{ls}(a1, a3)) \wedge (\neg \mathbf{ls}(a2, a3) \vee (\mathbf{True} \wedge (a1 \mapsto e1 * \mathbf{True})) \vee a1 = a3)$	✓	0.45
(20)	$((\mathbf{ls}(a1, a2) \wedge a1 \neq a2) \mathrel{-\!\!\circledast} \mathbf{ls}(e1, e2)) \wedge e1 \neq a1 \wedge e2 = a2 \wedge \neg \mathbf{ls}(e1, a1)$	✓	0.88
(21)	$a1 \neq a4 \wedge (\mathbf{ls}(a1, a4) \mathrel{-\!\!\circledast} \mathbf{ls}(e1, e2)) \wedge a4 = e2 \wedge \neg \mathbf{ls}(e1, a1)$	✓	1.23
(22)	$((\mathbf{ls}(a1, a2) \wedge a1 \neq a2) \mathrel{-\!\!\circledast} \mathbf{ls}(e1, e2)) \wedge e2 \neq a2 \wedge e1 = a1 \wedge \neg \mathbf{ls}(a2, e2)$	✓	0.89
(23)	$((a2 \mapsto a3 \mathrel{-\!\!\circledast} \mathbf{ls}(a2, a4)) \mathrel{-\!\!\circledast} \mathbf{ls}(a3, a1)) \wedge (\neg \mathbf{ls}(a4, a1) \vee a2 = a4)$	?	0.71

Table 11.4: Unsatisfiable formulas. The time is in seconds.

	Formula	U	Time
(1)	$\neg(a1 \mapsto a2) \wedge \mathbf{emp} \wedge (a1 \mapsto a2 * a3 \mapsto a4)$	✓	0.83
(2)	$a1 \mapsto a2 \wedge \mathbf{emp} \wedge (\neg(a1 \mapsto a2) * a3 \mapsto a4)$	✓	0.32
(3)	$\neg(a1 \mapsto a2) \wedge \mathbf{emp} \wedge (a1 \mapsto a2 * \mathbf{ls}(a3, a4))$	✓	0.62
(4)	$\mathbf{ls}(a1, a2) \wedge \mathbf{emp} \wedge (\neg\mathbf{ls}(a1, a2) * \mathbf{ls}(a3, a4))$	✓	8.46
(5)	$\mathbf{ls}(a1, a2) \wedge \mathbf{emp} \wedge (\neg\mathbf{ls}(a1, a2) * \neg\mathbf{ls}(a3, a4))$	✓	10.3

Table 11.5: Example instantiations of $T_1 \stackrel{\text{def}}{=} \neg a \wedge \mathbf{emp} \wedge (a * b)$, where $a, b \in$ Literals. The time is in seconds.

Though not shown in this section, I also evaluated our procedure on a set of satisfiable formulas. The procedure reports a set of abstract models when given a satisfiable formula (see Section 11.2.2).

We now answer Questions 1–3 posed at the beginning of this section using the three groups of formulas.

Group 1 Results. The running time of our procedure on the formulas listed in Table 11.4 was often on the order of one second. The TVLA startup and shutdown time for Group 1 was 10.9 seconds. The procedure was able to prove unsatisfiability for all formulas, except (23). I believe that formulas (9)–(23) are beyond the scope of previously existing tools. Formulas (9)–(14) demonstrate that we can handle formulas that describe overlapping data structures, including conjunctions of separating conjunctions. Formulas (15)–(21) demonstrate that we can handle formulas containing **ls** and septraction together.

Group 2 Results. The 64 formulas instantiated from the template $T_1 \stackrel{\text{def}}{=} \neg a \wedge \mathbf{emp} \wedge (a * b)$ took between 0.0003 and 10.31 seconds to check, with a mean of 0.56 and a median of 0.03 seconds. Our procedure was able to prove unsatisfiability for all 64 formulas. The TVLA startup and shutdown time for Group 2 was 3.39 seconds. All instantiations of T_1 that contain an occurrence of the **ls** predicate are beyond the capabilities of existing tools. The formulas that took the most time were (5) and (4) in Table 11.5. In both cases, a large amount of time was required because of the presence of $\neg\mathbf{ls}$, which is represented by 24 structures—a much larger number than is needed for the other literals.

	Formula	U	Time
(1)	$\mathbf{emp} \wedge \mathbf{ls}(a1, a2) \wedge (\mathbf{ls}(a3, a4) * (\mathbf{ls}(a5, a6) \mathrel{-\!\!\circledast} (\mathbf{emp} \wedge \neg\mathbf{ls}(a1, a2))))$	✓	0.37
(2)	$\mathbf{emp} \wedge \neg\mathbf{emp} \wedge (\mathbf{ls}(a3, a4) * (\neg(a5 \mapsto a6) \mathrel{-\!\!\circledast} (\mathbf{emp} \wedge \mathbf{emp})))$	✓	0.17
(3)	$\mathbf{emp} \wedge a1 \mapsto a2 \wedge (a3 \mapsto a4 * (a5 \mapsto a6 \mathrel{-\!\!\circledast} (\mathbf{emp} \wedge \neg(a1 \mapsto a2))))$	✓	0.49
(4)	$\mathbf{emp} \wedge \neg\mathbf{ls}(a1, a2) \wedge (\neg\mathbf{ls}(a3, a4) * (\mathbf{ls}(a5, a6) \mathrel{-\!\!\circledast} (\mathbf{emp} \wedge \mathbf{ls}(a1, a2))))$	✓	3.97
(5)	$\mathbf{emp} \wedge \neg\mathbf{ls}(a1, a2) \wedge (\neg\mathbf{ls}(a3, a4) * (\mathbf{emp} \mathrel{-\!\!\circledast} (\mathbf{emp} \wedge \mathbf{ls}(a1, a2))))$	✓	9.51

Table 11.6: Example instantiations of $T_2 \stackrel{\text{def}}{=} \mathbf{emp} \wedge a \wedge (b * (c \mathrel{-\!\!\circledast} (\mathbf{emp} \wedge \neg a)))$, where $a, b, c \in$ Literals. The time is in seconds.

Group 3 Results. The 512 formulas instantiated from the template $T_2 \stackrel{\text{def}}{=} \mathbf{emp} \wedge a \wedge (b * (c \mathrel{-\!\!\circledast} (\mathbf{emp} \wedge \neg a)))$ took between 0.0001 and 9.51 seconds to check using our procedure, with a mean of 0.12, and a median of 0.04 seconds. Our procedure was able to prove unsatisfiability for all 512 formulas. The TVLA startup and shutdown time for Group 3 was 10.12 seconds. All instantiations of T_2 that contain an occurrence of **ls** are beyond the capabilities of existing tools.

11.5 Related Work

The literature related to reasoning about separation logic is vast, and I mention only a small portion of it in this section. Decidability results related to first-order separation logic are discussed in Calcagno et al. (2001); Brochenin et al. (2012). A fragment of separation logic for which it is decidable to check validity of entailments was introduced by Berdine et al. (2004). The fragment includes points-to and linked-list predicates, but no septraction, or negations of points-to or linked-list predicates. More recent approaches deal with fragments of separation logic that are incomparable to ours (Park et al., 2013; Lee and Park, 2014; Hou et al., 2014); in particular, none of the latter papers handle linked lists. I based the experiments on formulas listed in Hou et al.'s work on Boolean separation logic (Hou et al., 2014)—the only paper I found that listed formulas outside the syntactic fragment defined by Berdine et al. I believe that our technique

represents the first important step in designing a verification system that uses a richer fragment of separation logic.

Most approaches to separation-logic reasoning use a syntactic proof-theoretic procedure (Berdine et al., 2004; Pérez and Rybalchenko, 2011). Two exceptions are the approaches of Cook et al. (2011) and Enea et al. (2013), which use a more semantics-based approach: they represent separation-logic formulas as graphs in a particular normal form, and then prove that one formula entails another by finding a homomorphism between the corresponding graphs. Our approach is also semantics-based, but has more of an algebraic flavor: our method performs a bottom-up evaluation of a formula φ using a particular shape-analysis interpretation (Figure 11.4); if the answer is the empty set of 3-valued structures, then φ is unsatisfiable.

To deal with overlaid data-structures, Enea et al. (2013) introduce the $*_w$ operator: the $*_w$ operator specifies data structures that share sets of objects as long as they are built over disjoint sets of *fields*. Their approach, however, does not handle conjunctions of separating conjunctions or negations of the **ls**-predicate. Thus, Enea et al. (2013) cannot handle formulas (9)–(14) in Table 11.4, even though these formulas do not contain septraction. Note that, for instance, the logical conjunction in formula (9) cannot be replaced by the $*_w$ operator.

Piskac et al. (2013) present a decision procedure for a decidable fragment of separation logic based on a reduction to a particular decidable first-order theory. Unlike our approach, the approach in Piskac et al. (2013) does not handle septraction or negations below a separating conjunction.

Many researchers pigeonhole TVLA (Lev-Ami and Sagiv, 2000) as a system exclusively tailored for "shape analysis". In fact, it is actually a metasystem for (i) defining a family of logical structures 2-STRUCT[Voc], and (ii) defining canonical-abstraction domains whose elements represent sets of 2-STRUCT[Voc]. The ITVLA (Jeannet et al., 2010, Section 8) variant of TVLA is a different packaging of the classes that make up the TVLA implementation, and demonstrates better that canonical abstraction is a general-purpose method for abstracting the structures that are a logic's domain of discourse.

To simplify matters, the separation-logic fragment addressed in this chapter does not allow one to make assertions about numeric-valued variables and numeric-valued fields. Our approach could be extended to support such capabilities using methods developed in work on abstract interpretation that combines canonical abstraction with numeric abstractions (Gopan et al., 2004; McCloskey et al., 2010).

11.6 Future Work

If φ is satisfiable, then the procedure described in this chapter reports a set of abstract models—i.e., a value in the canonical-abstraction domain that overapproximates $[\![\varphi]\!]$. As shown in Chapter 6 (using a variety of other techniques, and for a variety of other logics), a decision-procedure-like method that is prepared to return such "residual" answers provides a way to generate sound abstract transformers automatically. In particular, when φ specifies the transition relation between the pre-state and post-state of a concrete transformer τ, a residuating decision procedure provides a way to create a sound abstract transformer $\tau^\sharp$ for τ, directly from a specification in logic of τ's concrete semantics. Using our procedure, we now have a way to create abstract transformers based on canonical-abstraction domains directly from a specification of the semantics of a language's concrete transformers, written in SL.

Although TVLA and separation logic have both been applied to the problem of analyzing programs that manipulate linked data structures, there has been only rather limited crossover of ideas between the two approaches. Our procedure is built on the connection between TVLA states and SL statelets described in Section 11.1.3, which represents the first formal connection between the two approaches. For this reason, the procedure should be of interest to both communities:

- For the TVLA community, the procedure illustrates a different and intriguing use for canonical-abstraction domains. The domains that we use are tailored for the particular formula, but, more importantly, provide an encoding that can be connected to the SL semantics: see Equations (11.2) and (11.3) in Section 11.1.3, and the use of domain predicates to express disjointness in Section 11.2.

- For the separation-logic community, the procedure shows how using TVLA and canonical-abstraction domains leads to a model-theoretic approach to the decision problem for SL that is capable of handling formulas that are beyond the capabilities of existing tools.

The question of whether or not the separation-logic fragment considered in this chapter is decidable remains an open question. However, the eventual goal of this research direction is to extend the approach to deal with richer fragments of separation logic. In particular, I believe that our approach could be extended to perform unsatisfiability checking (and symbolic abstraction) for undecidable fragments that contain both separating implication and acyclic linked-list predicates.

11.7 Chapter Notes

The idea of building an SMA solver for separation logic using the abstract domain of TVLA shape graphs seemed intuitive and straight forward at first. Devising the right abstraction predicates for the TVLA abstract domain, and the correct integrity constraints for the coerce operation turned out to be fiddly work. Further work in this direction would require automation of these two tasks.

Jason Breck helped with the implementation of SMASLTOV, especially with the integration of ITVLA with the latest implementation of TVLA.

Jason Breck and Thomas Reps came up with the name “SMASLTOV” for the SMA solver for separation logic with only minor input from me.

Chapter 12

Property-Directed Symbolic Abstraction

This chapter presents a framework for computing inductive invariants for a program that are sufficient to prove that a given pre/post-condition holds, which I call the *property-directed inductive-invariant (PDII) framework*. To understand the particular properties of the framework described in this chapter I contrast it with the framework for computing best inductive $\mathcal{A}$-invariants of Chapter 7, which I will refer to as the BII framework:

1. The PDII framework computes an inductive invariant that might not be the best (or most precise), but is sufficient to prove a given program property. The BII framework, in general, aims to compute the best inductive $\mathcal{A}$-invariant (in the given resource/time constraints).

2. In case the program does not satisfy the given property, the PDII framework reports a concrete counter-example to the property. The BII framework is agnostic to the particular property to be proved.

3. The PDII framework is applicable to the abstract domain of full predicate abstraction; the BII framework can be instantiated with a larger class of abstract domains, which includes full predicate abstraction.

The advantages of the PDII method described in this chapter are two-fold:

1. The PDII framework obtains the same precision as the best abstract transformer for full predicate abstraction, without ever constructing the transformers explicitly.

2. The PDII framework is *relatively complete with respect to the given abstraction*. That is, the analysis is guaranteed to terminate and either

   a) verifies the given property,

   b) generates a concrete counterexample to the given property, or

   c) reports that the abstract domain is not expressive enough to establish the proof.

   Note that outcome c) is a much stronger guarantee than what other approaches provide in such cases when they neither succeed nor give a concrete counterexample.

The PDII framework is based on the IC3 algorithm (Bradley, 2011), which is also referred to as *property-directed reachability* (PDR). The PDR algorithm was chosen for the following two reasons:

- The PDR algorithm has been shown to work extremely well in other domains, such as hardware verification (Bradley, 2011; Eén et al., 2011). Subsequently, it was generalized to software model checking for program models that use linear real arithmetic (Hoder and Bjørner, 2012) and linear rational arithmetic (Cimatti and Griggio, 2012).

- The PDR algorithm maintains a "frame" structure, which can be used to build a trace formula; if the formula is satisfiable, the model can be presented to the user as a concrete counterexample (outcome b)).

In this chapter I describe how to integrate PDR with full predicate abstraction, and cast PDR as a *framework* that is parameterized on

- the logic $\mathcal{L}$ in which the semantics of program statements are expressed, and

Algorithm 23: PDII$_{\mathcal{A}}(\mathit{Init}, \rho, \mathit{Bad})$

```
1  R[−1] ← false
2  R[0] ← true
3  N ← 0
4  while true do
5      if there exists 0 ≤ i < N such that R[i] = R[i + 1] then
6          return valid
7      (r, A) ← Check_A(Bad, R[N])
8      if r = unsat then
9          N ← N + 1
10         R[N] ← true
11     else
12         reduce_A(N, A)
```

- the finite set of predicates that define the abstract domain $\mathcal{A}$ in which invariants can be expressed. An element of $\mathcal{A}$ is an arbitrary Boolean combination of the predicates.

The PDII framework has been instantiated to prove shape properties of programs manipulating singly linked lists. This instantiation represents the first algorithm for shape analysis that either (i) succeeds, (ii) returns a concrete counterexample, or (iii) returns an abstract trace showing that the abstraction in use is not capable of proving the property in question. Details of the instantiation to shape analysis and the experimental evaluation can be found in Itzhaky et al. (2014).

The contribution of this chapter is:

- The PDII framework, based on the PDR algorithm, for finding an inductive invariant in a certain logic fragment (abstract domain) that allows one to prove that a given pre-/post-condition holds or find a concrete counter-example to the property, or, in the case of a negative result, the information that there is no inductive invariant expressible in the abstract domain (Section 12.1).

Algorithm 24: reduce$_{\mathcal{A}}(j, A)$

```
1  (r, A_1) ← Check_A(Init, A)
2  if r = sat then
3      σ ← Model(Init ∧ ρ^{N−j} ∧ (Bad)'^{×(N−j)})
4      if σ is None then
5          error "abstraction failure"
6      else
7          error "concrete counterexample(σ)"
8  while true do
9      (r, A_2) ←
10         Check_A((Init)' ∨ (R[j − 1] ∧ ρ), (A)')
11     if r = unsat then
12         break
13     else
14         reduce_A(j − 1, A_2)
15 for i = 0 . . . j do
16     R[i] ← R[i] ∧ (¬A_1 ∨ ¬A_2)
```

12.1 The Property-Directed Inductive Invariant (PDII) Algorithm

In this section, I present the PDII framework, which is an adaptation of the PDR algorithm that uses predicate abstraction. In this chapter, by *predicate abstraction* I mean the technique that performs verification using a given *fixed* set of abstraction predicates (Flanagan and Qadeer, 2002), and not techniques that incorporate automatic refinement of the abstraction predicates; e.g., CEGAR. The PDII algorithm shown in Algorithm 23 is parameterized by a given finite set of predicates $\mathcal{P}$ expressed in a logic $\mathcal{L}$. The requirements on the logic $\mathcal{L}$ are:

R1 $\mathcal{L}$ is decidable (for satisfiability).

R2 The transition relation for each statement of the programming language can be expressed as a two-vocabulary $\mathcal{L}$ formula.

Then for a particular program, we are given:

- A finite set of predicates $\mathcal{P} = \{p_i \in \mathcal{L}\}, 1 \leq i \leq n$.
- The transition relation of the system as a two-vocabulary formula $\rho \in \mathcal{L}$.

- The initial condition of the system, $\mathit{Init} \in \mathcal{L}$.
- The formula specifying the set of bad states, $\mathit{Bad} \in \mathcal{L}$.

Let $\mathcal{A}$ be the full predicate-abstraction domain over the predicates $\mathcal{P}$. That is, each element $A \in \mathcal{A}$ is an *arbitrary* Boolean combination of the predicates $\mathcal{P}$. $A \in \mathcal{A}$ is inductive with respect to *Init* and ρ if and only if $\mathit{Init} \Rightarrow A$ and $A \wedge \rho \Rightarrow (A)'$. $(\varphi)'$ renames the vocabulary of constant symbols and relation symbols occurring in φ from $\{c, \ldots, r, \ldots\}$ to $\{c', \ldots, r', \ldots\}$. φ is $(\varphi)'$ stripped of primes.

If the logic $\mathcal{L}$ is propositional logic, then Algorithm 23 is an instance of IC3 (Bradley, 2011). This presentation is a simplification of more advanced variants (Bradley, 2011; Eén et al., 2011; Hoder and Bjørner, 2012). For instance, the presentation omits inductive generalization, although the implementation of the framework does implement inductive generalization (Itzhaky et al., 2014). Furthermore, this simplified presentation brings out the fact that the PDR algorithm is really an analysis *framework* that is parameterized on the set of abstraction predicates $\mathcal{P}$.

The algorithm employs an unbounded array R, where each *frame* $R[i] \in \mathcal{A}$ over-approximates the set of concrete states after executing the loop at most i times. The algorithm maintains an integer N, called the *frame counter*, such that the following invariants hold for all $0 \leq i < N$:

1. *Init* is a subset of all $R[i]$, i.e., $\mathit{Init} \Rightarrow R[i]$.
2. The safety requirements are satisfied, i.e., $R[i] \Rightarrow \neg \mathit{Bad}$.
3. Each of the $R[i+1]$ includes the states in $R[i]$, i.e., $R[i] \Rightarrow R[i+1]$.
4. The successors of $R[i]$ are included in $R[i+1]$, i.e., for all σ, σ' if $\sigma \models R[i]$ and $\langle \sigma, \sigma' \rangle \models \rho$, then $\sigma' \models R[i+1]$.

Some terminology used in the PDII algorithm:

- `Model`(φ) returns a model σ satisfying φ if it exists, and `None` if it doesn't.
- The abstraction of a model σ, denoted by $\beta_{\mathcal{A}}(\sigma)$, is the cube of predicates from $\mathcal{P}$ that hold in σ: $\beta_{\mathcal{A}}(\sigma) = \bigwedge\{p \mid \sigma \models p, p \in \mathcal{P}\} \wedge \bigwedge\{\neg q \mid \sigma \models \neg q, q \in \mathcal{P}\}$.

- Let $\varphi \in \mathcal{L}$ be a formula, and let $A \in \mathcal{A}$ be a value in the unprimed or primed vocabulary. $Check_{\mathcal{A}}(\varphi, A)$ returns a pair (r, A_1) such that
  - if $\varphi \wedge A$ is satisfiable, then $r = \mathsf{sat}$ and A_1 is the abstraction of a concrete state in the unprimed vocabulary. That is, if the given A is in the unprimed vocabulary, then $\beta_{\mathcal{A}}(\sigma)$ for some $\sigma \models \varphi \wedge A$; else if A is in the primed vocabulary, then $A_1 = \beta_{\mathcal{A}}(\sigma)$ for some $(\sigma, \sigma') \models \varphi \wedge A$.
  - if $\varphi \wedge A$ is unsatisfiable, then $r = \mathsf{unsat}$, and A_1 is a predicate such that $A \Rightarrow A_1$ and $\varphi \wedge A_1$ is unsatisfiable. The vocabulary of A_1 is the same as that of A. If A is in the primed vocabulary (as in line 10 of Algorithm 24), $Check_{\mathcal{A}}$ drops the primes from A_1 before returning the value.

  A valid choice for A_1 in the unsatisfiable case would be $A_1 = A$ (and indeed the algorithm would still be correct), but ideally A_1 should be the weakest such predicate. For instance, $Check_{\mathcal{A}}(\textbf{false}, A)$ should return $(\mathsf{unsat}, \textbf{true})$. In practice, when $\varphi \wedge A$ is unsatisfiable, the A_1 returned is an unsat core of $\varphi \wedge A$ constructed exclusively from conjuncts of A. Such an unsat core is a Boolean combination of predicates in $\mathcal{P}$, and thus is an element of $\mathcal{A}$.

I now give a more detailed explanation of Algorithm 23. Each $R[i], i \geq 0$ is initialized to **true** (lines 2 and 10), and $R[-1]$ is **false**. N is initialized to 0 (line 3). At line 5, the algorithm checks whether $R[i] = R[i+1]$ for some $0 \leq i < N$. If true, then an inductive invariant proving unreachability of *Bad* has been found, and the algorithm returns valid (line 6).

At line 7, the algorithm checks whether $R[N] \wedge \mathit{Bad}$ is satisfiable. If it is unsatisfiable, it means that $R[N]$ excludes the states described by *Bad*, and the frame counter N is incremented (line 9). Otherwise, $A \in \mathcal{A}$ represents an abstract state that satisfies $R[N] \wedge \mathit{Bad}$. PDII then attempts to reduce $R[N]$ to try and exclude this abstract counterexample by calling $\text{reduce}_{\mathcal{A}}(N, A)$ (line 12).

The reduce algorithm (Algorithm 24) takes as input an integer $j, 0 \leq j \leq N$, and an abstract state $A \in \mathcal{A}$ such that there is a path starting from A of length $N - j$ that reaches *Bad*. Algorithm 24 tries to strengthen $R[j]$ so as to exclude A. At line 1, reduce first checks whether $\mathit{Init} \wedge A$ is

satisfiable. If it is satisfiable, then there is an abstract trace of length $N - j$ from *Init* to *Bad*, using the transition relation ρ. The call to `Model` at line 3 checks whether there exists a concrete model corresponding to the abstract counterexample. ρ^k denotes k unfoldings of the transition relation ρ; ρ^0 is **true**. $(Bad)'^{\times k}$ denotes k applications of the renaming operation $(\cdot)'$ to *Bad*. If no such concrete model is found, then the abstraction was not precise enough to prove the required property (line 5); otherwise, a concrete counterexample to the property is returned (line 7).

Now consider the case when $Init \wedge A$ is unsatisfiable on line 1. $A_1 \in \mathcal{A}$ returned by the call to $Check_{\mathcal{A}}$ is such that $Init \wedge A_1$ is unsatisfiable; that is, $Init \Rightarrow \neg A_1$.

The while-loop on lines 8–14 checks whether the $(N - j)$-length path to *Bad* can be extended backward to an $(N - j + 1)$-length path. In particular, it checks whether $R[j-1] \wedge \rho \wedge (A)'$ is satisfiable. If it is satisfiable, then the algorithm calls reduce recursively on $j - 1$ and A_2 (line 14). If no such backward extension is possible, the algorithm exits the while loop (line 12). Note that if $j = 0$, $Check_{\mathcal{A}}(R[j-1] \wedge \rho, A)$ returns $(\mathsf{unsat}, \textbf{true})$, because $R[-1]$ is set to **false**.

The conjunction of $(\neg A_1 \vee \neg A_2)$ to $R[i], 0 \leq i \leq j$, in the loop on lines 15–16 eliminates abstract counterexample A while preserving the required invariants on R. In particular, the invariant $Init \Rightarrow R[i]$ is maintained because $Init \Rightarrow \neg A_1$, and hence $Init \Rightarrow (R[i] \wedge (\neg A_1 \vee \neg A_2))$. Furthermore, A_2 is the abstract state from which there is a (spurious) path of length $N - j$ to *Bad*. By the properties of $Check_{\mathcal{A}}$, $\neg A_1$ and $\neg A_2$ are each disjoint from A, and hence $(\neg A_1 \vee \neg A_2)$ is also disjoint from A. Thus, conjoining $(\neg A_1 \vee \neg A_2)$ to $R[i], 0 \leq i \leq j$ eliminates the spurious abstract counterexample A. Lastly, the invariant $R[i] \Rightarrow R[i+1]$ is preserved because $(\neg A_1 \vee \neg A_2)$ is conjoined to all $R[i], 0 \leq i \leq j$, and not just $R[j]$.

Formally, the output of $\text{PDII}_{\mathcal{A}}(Init, \rho, Bad)$ is captured by the following theorem. The proof of Theorem 12.1 is based on the observation that, when "abstraction failure" is reported by $\text{reduce}_{\mathcal{A}}(j, A)$, the set of models $\sigma_i \models R[i]$ $(j \leq i < N)$ represents an abstract error trace.

Theorem 12.1. *Given (i) the set of abstraction predicates $\mathcal{P} = \{p_i \in \mathcal{L}\}, 1 \leq i \leq n$ where $\mathcal{L}$ is a decidable logic, and the full predicate-abstraction domain $\mathcal{A}$ over $\mathcal{P}$, (ii) the initial condition $Init \in \mathcal{L}$, (iii) a transition relation ρ expressed as a two-vocabulary formula in $\mathcal{L}$, and (iv) a formula $Bad \in \mathcal{L}$*

specifying the set of bad states, $PDII_{\mathcal{A}}(Init, \rho, Bad)$ terminates, and reports either

1. valid *if there exists $A \in \mathcal{A}$ s.t. (i) $Init \Rightarrow A$, (ii) A is inductive, and (iii) $A \Rightarrow \neg Bad$,*
2. *a concrete counterexample trace, which reaches a state satisfying Bad, or*
3. *an abstract trace, if the inductive invariant required to prove the property cannot be expressed as an element of $\mathcal{A}$.*

Proof. The first two cases are trivial: if PDII$_{\mathcal{A}}$ terminates returning some $R[j]$, $j < N$, then $Init \Rightarrow R[j]$ by virtue of $Init \Rightarrow R[0]$ and $R[i] \Rightarrow R[i+1]$ for every $i < N$, and $R[j] \Rightarrow \neg Bad$ or the check at line 7 would have failed. Also, $R[j-1] \equiv R[j]$ so $R[j]$ is inductive.

If PDII$_{\mathcal{A}}$ returns a set of concrete states, then they have to be a concrete counterexample trace, because they are a model of $Init \wedge \rho^{N-j} \wedge (Bad)'^{\times(N-j)}$ (line 3 of reduce$_{\mathcal{A}}$).

For the third case, we show that if the check on the first line of "reduce" is "sat", then there exists a chain of concrete states, $\sigma_j \quad \sigma_{j+1} \cdots \sigma_N$, such that $\sigma_j \models Init$, $\sigma_N \models Bad$, and for any $j \leq i < N$ there exist two concrete states σ, σ' satisfying:

- $\sigma \in \gamma(\beta_{\mathcal{A}}(\sigma_i))$
- $\sigma' \in \gamma(\beta_{\mathcal{A}}(\sigma_{i+1}))$
- $\langle \sigma, \sigma' \rangle \models \rho$

The key point is that, because the given abstraction can never distinguish any two states in $\gamma(\beta_{\mathcal{A}}(\sigma_i))$, the chain $\sigma_j \quad \sigma_{j+1} \cdots \sigma_N$ cannot be excluded by the abstract domain $\mathcal{A}$, no matter what Boolean combination of the predicates of $\mathcal{P}$ is used. Moreover, the chain $\beta_{\mathcal{A}}(\sigma_j)\, \beta_{\mathcal{A}}(\sigma_{j+1}) \cdots \beta_{\mathcal{A}}(\sigma_N)$ is an abstract trace that leads from an initial state to an error state.

Notice that the chain above may not be a concrete trace, there can be "breaks" between adjacent σ_is, within the same abstract element.

Construction of $(\sigma_i)_{i=j \ldots N}$: Follow the chain of recursive calls to "reduce" with index values N down to j. The parameter A is always a cube of the form $\beta_{\mathcal{A}}(\sigma)$; take one $\sigma \models A$ for each

call, forming a series that we denote by σ_j, σ_{j+1}, etc. We show that this series satisfies the above properties: At each call except the innermost, "reduce" made a recursive call at line 7, which means that $R[j-1] \wedge \rho \wedge (A)'$ was satisfiable; the returned cube A_2 becomes $\beta_{\mathcal{A}}(\sigma_{j-1})$. Let $\langle \sigma, \sigma' \rangle \models R[j-1] \wedge \rho \wedge (A)'$, then $\sigma \models A_2 = \beta_{\mathcal{A}}(\sigma_{j-1})$; $\sigma' \models A = \beta_{\mathcal{A}}(\sigma_j)$; and $\langle \sigma, \sigma' \rangle \models \rho$ as required. □

12.2 Chapter Notes

The research presented in this chapter was done in collaboration with Shachar Itzhaky, Nikolaj Bjorner, Mooly Sagiv, and Thomas Reps. In this chapter, I have presented my primary contribution—that is, the formulation of the generalized PDII framework (Algorithms 23 and 24). The implementation of this generalized PDII framework, its instantiation to shape analysis of programs manipulating singly linked lists, and the experimental evaluation of this instantiation was mainly carried out by Shachar Itzhaky and Nikolaj Bjorner. Because I did not play a major role in those aspects of the research, I chose to omit them from the thesis.

Key to instantiating the PDII framework for shape analysis of programs manipulating singly linked lists was the recent development of the AF^R and EA^R logics for expressing properties of linked lists (Itzhaky et al., 2013). AF^R is used to define abstraction predicates, and EA^R is used to express the language semantics. AF^R is a decidable, alternation-free fragment of first-order logic with transitive closure (FO^{TC}). When applied to list-manipulation programs, atomic formulas of AF^R can denote reachability relations between memory locations pointed to by pointer variables, where reachability corresponds to repeated dereferences of *next* or *prev* fields. One advantage of AF^R is that it does not require any special-purpose reasoning machinery: an AF^R formula can be converted to a formula in "effectively propositional" logic, which can be reduced to SAT solving. That is, in contrast to much previous work on shape analysis, this PDR-based method makes use of *a general purpose SMT solver*, Z3 (de Moura and Bjørner, 2008) (rather than specialized tools developed for reasoning about linked data structures, e.g., (Sagiv et al., 2002; Distefano

et al., 2006; Berdine et al., 2007; Garg et al., 2013)). The reader is encouraged to read Itzhaky et al. (2014) to learn more about this application of the PDII framework to shape analysis.

Chapter 13

Conclusion

This chapter summarizes the results developed during my dissertation research, and presents my contributions to advancing the field.

The thesis explored the use of *abstraction* in two areas of automated reasoning: verification of programs, and decision procedures for logics. Chapter 1 laid the groundwork for the thesis by viewing these two research areas through the lens of abstraction. Many of the applications discussed in the thesis dealt with verification of machine code, a burgeoning subfield of program verification (Chapter 2).

The unifying theme behind the thesis is the concept of *symbolic abstraction* (Reps et al., 2004):

> Given a formula φ in logic $\mathcal{L}$ and an abstract domain $\mathcal{A}$, the symbolic abstraction of φ, denoted by $\widehat{\alpha}(\varphi)$, is the strongest consequence of φ expressible in $\mathcal{A}$.

Symbolic abstraction bridges the gap between abstraction and logic, and many of the operations required by an abstract interpreter can be implemented using symbolic abstraction (Section 3.2.3). Furthermore, this thesis showed how the concept of symbolic abstraction brings forth the connection between abstract interpretation (Cousot and Cousot, 1977) and decision procedures for logics.

Instead of summarizing each individual chapter, I chose to present the results for the three research threads discussed in this thesis: abstract interpretation (Section 13.1), machine-code

verification (Section 13.2), and decision procedures (Section 13.3). Note that because some of the chapters touch on more than one research thread, there is some repetition of text between the sections in this chapter.

13.1 Abstract Interpretation

The contributions of the thesis in the field of abstract interpretation are:

- New algorithms for performing symbolic abstraction (Chapters 5 and 6).
- New algorithms for computing inductive invariants for programs (Chapters 7 and 12).
- A new abstract domain for bit-vector inequalities implemented using symbolic abstraction (Chapter 8).

Chapter 4 reviewed two prior algorithms for performing symbolic abstraction:

- The RSY algorithm: a framework for computing $\widehat{\alpha}$ that applies to any logic and abstract domain that satisfies certain conditions (Reps et al., 2004).
- The KS algorithm: an algorithm for computing $\widehat{\alpha}$ that applies to QFBV logic and the domain $\mathcal{E}_{2^w}$ of affine equalities (Elder et al., 2011).

Both algorithms compute $\widehat{\alpha}(\varphi)$ via successive approximation from "below", computing a sequence of successively "larger" approximations to the set of states described by φ. Section 4.3 presented an empirical comparison of the RSY and KS algorithms that showed that the KS algorithm was ten times faster than the RSY algorithm. The main insight from this experiment was that, while the KS algorithm invokes the decision procedure more often, each of the calls to the decision procedure is significantly less expensive, compared to that in the RSY algorithm.

In Chapter 5, I used the insights gained from the algorithms presented in Chapter 4 to design a new *framework* for symbolic abstraction that

- is *parametric* and is applicable to any abstract domain that satisfies certain conditions (similar to the RSY algorithm)
- uses a successive-approximation algorithm that is *parsimonious* in its use of the decision procedure (similar to the KS algorithm)
- is *bilateral*; that is, it maintains both an under-approximation and a (non-trivial) over-approximation of the desired answer, and hence is *resilient to timeouts*: the procedure can return the over-approximation if it is stopped at any point (unlike the RSY and KS algorithms).

In contrast, neither the RSY algorithm nor KS algorithm is resilient to timeouts. A decision-procedure query—or the cumulative time for the algorithm—might take too long, in which case the only safe answer that can be returned is $\top$.

The key insight behind the new bilateral framework for symbolic abstraction, denoted by $\widetilde{\alpha}^{\updownarrow}$, was the notion of an abstract consequence (Definition 5.1).

Chapter 6 presented an algorithm for symbolic abstraction that is based on much different principles from the RSY, KS, and bilateral algorithms. The latter frameworks use an *inductive-learning approach* to learn from examples. In contrast, the symbolic-abstraction framework discussed in Chapter 6 uses a *deductive approach*, and is based on the following two insights:

- Each of the key components in Stålmarck's method (Sheeran and Stålmarck, 2000), an algorithm for satisfiability checking of propositional formulas, can be explained in terms of concepts from the field of abstract interpretation. In particular, I showed that Stålmarck's method can be viewed as a general framework, which I call Stålmarck[$\mathcal{A}$], that is parameterized by an abstract domain $\mathcal{A}$ and operations on $\mathcal{A}$.

- When viewed through the lens of abstraction, there is a connection between this generalized Stålmarck's method and symbolic abstraction. In particular, to check whether a formula φ is unsatisfiable, Stålmarck[$\mathcal{A}$] computes $\widehat{\alpha}(\varphi)$.

This new algorithm for symbolic abstraction based on Stålmarck's method approaches its result from "above", and is denoted by $\widetilde{\alpha}^{\downarrow}$. Because the method approaches its result from "above", if the computation takes too much time, it can be stopped to yield a safe result—i.e., an over-approximation to the best abstract operation—at any stage, similar to the bilateral framework (Chapter 5). Section 6.4.1 presented an empirical comparison between the Stålmarck-based algorithm ($\widetilde{\alpha}^{\downarrow}$), and the bilateral algorithm ($\widetilde{\alpha}^{\downarrow}$), which showed that $\widetilde{\alpha}^{\downarrow}$ is faster and more precise compared to $\widetilde{\alpha}^{\updownarrow}$.

Chapters 7 and 12 present two different algorithms for computing inductive invariants for a program. Chapter 7 presents a framework for computing best inductive $\mathcal{A}$-invariants, which is based on the following insights:

- The BII problem reduces to the problem of applying $\widehat{\mathrm{Post}}$.
- The problem of applying $\widehat{\mathrm{Post}}$ reduces to the problem of symbolic abstraction.

This work provided insight on fundamental limits in abstract interpretation. Furthermore, the BII algorithm presented is also practical: Santini is an invariant-generation tool based on the principles of symbolic abstraction. The performance of the Corral model checker (Lal et al., 2012) improved when using invariants supplied by Santini, compared to using invariants supplied by Houdini (Flanagan and Leino, 2001); see (Thakur et al., 2013, Section 5). This experiment shows that symbolic abstraction provides a powerful tool that can be used to implement automatically a correct and precise invariant generator that uses an expressive abstract domain.

Chapter 12 presents a framework for computing inductive invariants for a program that are sufficient to prove that a given pre/post-condition holds, which I called the *property-directed inductive-invariant (PDII) framework*. The PDII framework computes an inductive invariant that might not be the best (or most precise), but is sufficient to prove a given program property. In

case the program does not satisfy the given property, the PDII framework reports a concrete counter-example to the property. The PDII framework is applicable to the abstract domain of full predicate abstraction. The advantages of the PDII method are two-fold:

1. The PDII framework obtains the same precision as the best abstract transformer for full predicate abstraction, without ever constructing the transformers explicitly.

2. The PDII framework is *relatively complete with respect to the given abstraction*. That is, the analysis is guaranteed to terminate and either

   a) verifies the given property,

   b) generates a concrete counterexample to the given property, or

   c) reports that the abstract domain is not expressive enough to establish the proof.

   Note that outcome c) is a much stronger guarantee than what other approaches provide in such cases when they neither succeed nor give a concrete counterexample.

The PDII framework is based on the IC3 algorithm (Bradley, 2011) (sometimes called *property-directed reachability* (PDR)). An instantiation of the PDII framework to prove shape properties of programs manipulating singly linked lists can be found in Itzhaky et al. (2014).

Chapter 8 presents the design and implementation of a new abstract domain for bit-vector inequalities, called the $\mathcal{BVI}$ domain. The key insights behind the design of the $\mathcal{BVI}$ domain were:

- We extend the standard vocabulary of an abstract domain with *view expressions*, which are w-bit terms expressed in some logic $\mathcal{L}$. Thus, view expressions are capable of holding onto richer constraints about the program state than the unenriched abstract domain alone.

- We construct the $\mathcal{BVI}$ domain as a reduced product between the bit-vector affine-equality domain $\mathcal{E}_{2^w}$ and the bit-vector interval domain $\mathcal{I}_{2^w}$.

- We use symbolic abstraction to implement precise versions of the abstract-domain operations for $\mathcal{BVI}$.

13.2 Machine-Code Verification

Chapter 2 discussed the unique challenges and opportunities involved in analyzing machine code. The contributions of the thesis in the field of machine-code analysis and verification are:

- New techniques for computing abstract transformers used to perform machine-code analysis (Chapters 5 and 6).
- A new abstract domain for bit-vector inequalities (Chapter 8).
- A new model-checking algorithm for stripped machine-code, called McVeto (Chapter 9).

The use of symbolic abstraction greatly reduces the burden on the analysis writer when implementing abstract operations required for performing analysis. This benefit is especially helpful when implementing machine-code analyses, because most machine-code instructions involve bit-wise operations. Section 1.1 illustrated how existing approaches to computing abstract transformers based on quantifier elimination are not applicable to machine-code analysis. On the other hand, computing abstract transformers based on operator-by-operator reinterpretation can be tedious, error-prone, and not precise (Section 3.1.2). The experiments described in Sections 5.5 and 6.4.1 illustrate how the new algorithms for symbolic abstraction can be used to compute invariants for x86 machine code.

Chapter 8 described how symbolic abstraction enabled us to define a new abstract domain, called the *Bit-Vector Inequality* ($\mathcal{BVI}$) domain, that addresses the following challenges: (1) identifying affine-inequality invariants while handling overflow in arithmetic operations over bit-vector data-types, and (2) holding onto invariants about values in memory during machine-code analysis. The experiments in Section 9.4 showed that an analysis based on the $\mathcal{BVI}$ domain is capable of proving programs correct that could not be proved correct using existing abstract domains.

Chapter 9 described a new model-checking algorithm for stripped machine-code, called McVeto (**M**achine-**C**ode **VE**rification **TO**ol). McVeto is able to detect and explore "deviant behavior" in machine code. An example of such deviant behavior is when the program overwrites the return address stored on the stack frame. Moreover, McVeto is capable of verifying

(or detecting flaws in) self-modifying code (SMC). To the best of my knowledge, McVeto is the first model checker to handle SMC. The key insights behind the design of McVeto are:

- McVeto adapts *directed proof generation* (DPG) (Gulavani et al., 2006) for model checking stripped machine code.
- McVeto uses *trace-based generalization* to build and refine an abstraction of the program's state space entirely on-the-fly. Trace-based generalization enables McVeto to handle instruction aliasing and SMC.
- McVeto uses *speculative trace refinement* to identify *candidate invariants* that can speed up the convergence of DPG.

The experiments in Section 9.4 showed that McVeto is able to prove properties of small, but complicated, programs. The experiments also showed how candidate invariants obtained using the $\mathcal{BVI}$ domain can help speed up the convergence of DPG.

13.3 Decision Procedures

This thesis introduced a new approach for designing and implementing decision procedures based on the use of abstraction. This abstraction-centric view of decision procedures is called *Satisfiability Modulo Abstraction (SMA)*. Abstraction provides a new language for the description of decision procedures, and lead to new insights and new ways of thinking about decision procedures. Furthermore, the SMA approach is able to reuse abstract-interpretation machinery to implement decision procedures.

The contributions of the thesis in the field of decision procedures are:

- An SMA solver based on a generalization of Stålmarck's method (Sheeran and Stålmarck, 2000) (Chapter 6).
- A new distributed SAT solver, called `DiSSolve` (Chapter 10).
- An SMA solver for a new fragment of separation logic (Chapter 11).

Chapter 6 presented a new SMA solver that is based on the following two insights:

- Each of the key components in Stålmarck's method (Sheeran and Stålmarck, 2000), an algorithm for satisfiability checking of propositional formulas, can be explained in terms of concepts from the field of abstract interpretation. In particular, I showed that Stålmarck's method can be viewed as a general framework, which I call Stålmarck[$\mathcal{A}$], that is parameterized on an abstract domain $\mathcal{A}$ and operations on $\mathcal{A}$.
- When viewed through the lens of abstraction, Stålmarck's method can be lifted from propositional logic to *richer logics*, such as LRA: to obtain a method for richer logics, instantiate the parameterized version of Stålmarck's method with *richer abstract domains*, such as the polyhedral domain (Cousot and Halbwachs, 1978).

This abstraction-based approach provides new insights into the working Stålmarck's method. In particular, at each step, Stålmarck[$\mathcal{A}$] holds some $A \in \mathcal{A}$; each of the proof rules employed in Stålmarck's method improves A by finding a semantic reduction (Cousot and Cousot, 1979) of A with respect to φ. Furthermore, Section 6.4.2 shows how this generalized Stålmarck's method outperforms an existing SMT solver on a family of LRA formulas.

Chapter 10 described a new distributed SAT solver, called `DiSSolve`, which uses a new proof rule that combines concepts from Stålmarck's method with those found in modern DPLL/CDCL solvers:

1. `DiSSolve` partitions the search space using k variables in the same fashion that the Dilemma Rule partitions the search space in Stålmarck's method.

2. Each of the 2^k branches can be solved concurrently with the help of a sequential DPLL/CDCL solver, similar to what is done in the divide-and-conquer approach to parallel SAT solvers.

3. The DPLL/CDCL solver assigned to a branch is allotted a finite amount of time, after which the DPLL/CDCL solver returns a set of learned clauses. Such a branch-and-merge approach does not have to as careful about load balancing, unlike the divide-and-conquer approach.

4. `DiSSolve` performs a union of the information from all the branches, instead of an intersection as done in Stålmarck's method. Performing a union of clauses is more effective at pruning the search space compared to computing an intersection: with intersection only *common* information learned by every process can be used to prune the search space, while with union, *all* of the information learned by *each* process can be used to prune the search space. In abstract-interpretation terms, `DiSSolve` combines the information from the branches using a *meet* ($\sqcap$), while the Dilemma Rule in Stålmarck's method combines the information using a *join* ($\sqcup$).

Section 10.2 illustrated the effectiveness of `DiSSolve` when deployed on a multi-core machine, and on a cloud-computing platform. Section 10.3 described a natural extension of `DiSSolve` from SAT to SMT, and also presented the algorithm using abstract-interpretation terminology.

Chapter 11 described an SMA solver for checking the unsatisfiability of formulas in a new fragment of separation logic. Separation logic (Reynolds, 2002) is an expressive logic for reasoning about heap-allocated data structures in programs. The SMA solver uses the abstract domain of shape graphs—implemented in TVLA (Sagiv et al., 2002)—to represent a set of heap structures. The SMA solver performs a bottom-up evaluation of the given formula φ to compute an abstract value that over-approximates the set of satisfying models of φ. If the over-approximation is the empty set of shape graphs, then φ is unsatisfiable. If φ is satisfiable, then the procedure reports a set of abstract models.

The SMA solver for separation logic is implemented in a tool called SMASLTOV (Satisfiability Modulo Abstraction for Separation Logic ThrOugh Valuation), which is available at `https://www.`

`github.com/smasltov-team/SMASLTOV`. Section 11.4 described an evaluation of SMASLTOV on a set of formulas taken from the literature. To the best of my knowledge, SMASLTOV is able to establish the unsatisfiability of formulas that cannot be handled by previous approaches.

> The wind blew southward, through knotted forests, over shimmering plains and toward land unexplored. This wind, it was not the ending. There are no endings, and never will be endings, to the turning of the Wheel of Time.
>
> But it was *an* ending.
>
> — Robert Jordan and Brandon Sanderson, *A Memory of Light*

References

Abramson, Darren, and Lee Pike. 2011. When formal systems kill: Computer ethics and formal methods. *APA Newsletter on Philosophy and Computers*.

Allauzen, C., M. Riley, J. Schalkwyk, W. Skut, and M. Mohri. 2007. OpenFst: A general and efficient weighted finite-state transducer library. In *Conference on implementation and application of automata (CIAA)*.

Alur, R., and P. Madhusudan. 2009. Adding nesting structure to words. *Journal of ACM (JACM)* 56(3).

Alur, Rajeev, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal methods in computer-aided design (FMCAD)*.

AMD. 2012. AMD cpu bug update – AMD confirms! `http://article.gmane.org/gmane.os.dragonfly-bsd.kernel/14518`.

Angluin, Dana. 1987. Learning regular sets from queries and counterexamples. *Information and computation* 75(2):87–106.

Arnold, G., R. Manevich, M. Sagiv, and R. Shaham. 2006. Combining shape analyses by intersecting abstractions. In *Verification, model checking, and abstract interpretation (VMCAI)*.

Audemard, G., B. Hoessen, S. Jabbour, J. Lagniez, and C. Piette. 2012. Revisiting clause exchange in parallel SAT solving. In *International conference on theory and applications of satisfiability testing (SAT)*.

Audemard, Gilles, and Laurent Simon. 2009. Predicting learnt clauses quality in modern SAT solvers. In *International joint conference on artificial intelligence (IJCAI)*, 399–404.

Bagnara, R., P. M. Hill, and E. Zaffanella. 2008. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming (SCP)* 72(1–2):3–21.

Balakrishnan, G., and T. Reps. 2008. Analyzing stripped device-driver executables. In *Tools and algorithms for the construction and analysis of systems (TACAS)*.

———. 2010. WYSINWYX: What You See Is Not What You eXecute. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 32(6).

Balakrishnan, G., T. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S. Yong, C.-H. Chen, and T. Teitelbaum. 2005. Model checking x86 executables with CodeSurfer/x86 and WPDS++. In *Computer aided verification (CAV)*.

Balakrishnan, G., T. Reps, D. Melski, and T. Teitelbaum. 2007. WYSINWYX: What You See Is Not What You eXecute. In *Verified software: Theories, tools, experiments (VSTTE)*.

Ball, T., A. Podelski, and S.K. Rajamani. 2001. Boolean and Cartesian abstraction for model checking C programs. In *Tools and algorithms for the construction and analysis of systems (TACAS)*, 268–283.

Ball, T., and S.K. Rajamani. 2000. Bebop: A symbolic model checker for Boolean programs. In *SPIN model checking and software verification*.

———. 2001. The SLAM toolkit. In *Computer aided verification (CAV)*.

Ball, Thomas, Byron Cook, Vladimir Levin, and Sriram K Rajamani. 2004. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *Integrated formal methods*.

Barrett, Clark, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. 2009. Satisfiability modulo theories. In *Handbook of satisfiability*, ed. Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, vol. 185 of *Frontiers in Artificial Intelligence and Applications*, 825–886. IOS Press.

Barrett, E., and A. King. 2010. Range and set abstraction using SAT. *Electronic Notes in Theoretical Computer Science (ENTCS)*.

Beckman, N.E., A.V. Nori, S.K. Rajamani, and R.J. Simmons. 2008. Proofs from tests. In *International symposium on software testing and analysis (ISSTA)*.

Berdine, J., C. Calcagno, B. Cook, D. Distefano, P.W. O'Hearn, T. Wies, and H. Yang. 2007. Shape analysis for composite data structures. In *Computer aided verification (CAV)*.

Berdine, J., C. Calcagno, and P. O'Hearn. 2004. A decidable fragment of separation logic. In *Foundations of software technology and theoretical computer science (FSTTCS)*.

———. 2005. Smallfoot: modular automatic assertion checking with separation logic. In *Formal methods for components and objects (FMCO)*.

Bessey, Al, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM (CACM)* 53(2):66–75.

Beyer, D., T.A. Henzinger, R. Majumdar, and A. Rybalchenko. 2007. Path invariants. In *ACM SIGPLAN conference on programming language design and implementation (PLDI)*.

Biere, Armin. 2008. PicoSAT essentials. *Journal of Satisfiability, Boolean Modeling and Computation (JSAT)* 4(2-4):75–97.

———. 2009. Bounded model checking. In *Handbook of satisfiability*, ed. Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, vol. 185 of *Frontiers in Artificial Intelligence and Applications*, 457–482. IOS Press.

Bjesse, Per, and Koen Claessen. 2000. SAT-based verification without state space traversal. In *Formal methods in computer-aided design (FMCAD)*.

Björk, M. 2009. First order Stålmarck. *Journal of Automated Reasoning (JAR)* 42(1):99–122.

Bjørner, N., and L. de Moura. 2008. Accelerated lemma learning using joins–DPLL(⊔). In *Logic for programming, artificial intelligence, and reasoning (LPAR)*.

Bloch, Joshua. 2014. Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken. `googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html`.

Bloom, B., D. Grove, B. Herta, A. Sabharwal, H. Samulowitz, and V. A. Saraswat. 2012. SatX10: A scalable plug&play parallel sat framework - (tool presentation). In *International conference on theory and applications of satisfiability testing (SAT)*.

Bogudlov, I., T. Lev-Ami, T. Reps, and M. Sagiv. 2007a. Revamping TVLA: Making parametric shape analysis competitive. In *Computer aided verification (CAV)*.

———. 2007b. Revamping TVLA: Making parametric shape analysis competitive. Tech. Rep. TR-2007-01-01, Tel-Aviv Univ., Tel-Aviv, Israel.

Bourdoncle, F. 1993. Efficient chaotic iteration strategies with widenings. In *Formal methods in programming and their applications (FMPA)*.

Bowen, Jonathan. 2000. The ethics of safety-critical systems. *Communications of the ACM (CACM)* 43(4):91–97.

Bradley, Aaron R, Zohar Manna, and Henny B Sipma. 2006. What's decidable about arrays? In *Verification, model checking, and abstract interpretation (VMCAI)*.

Bradley, A.R. 2011. SAT-based model checking without unrolling. In *Verification, model checking, and abstract interpretation (VMCAI)*.

Brauer, J., and A. King. 2010. Automatic abstraction for intervals using Boolean formulae. In *Static analysis symposium (SAS)*.

———. 2011. Transfer function synthesis without quantifier elimination. In *European symposium on programming (ESOP)*.

Brochenin, R., S. Demri, and E. Lozes. 2012. On the almighty wand. *Information and Computation* 211:106–137.

Brumley, D., C. Hartwig, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. 2008. Automatically identifying trigger-based behavior in malware. In *Botnet detection*, ed. W. Lee, C. Wang, and D. Dagon, vol. 36 of *Advances in Information Security Series*, 65–88. Springer.

Bush, William R, Jonathan D Pincus, and David J Sielaff. 2000. A static analyzer for finding dynamic programming errors. *Software-Practice and Experience* 30(7):775–802.

Cai, H., Z. Shao, and A. Vaynberg. 2007. Certified self-modifying code. In *ACM SIGPLAN conference on programming language design and implementation (PLDI)*.

Calcagno, C., V. Vafeiadis, and M. Parkinson. 2007. Modular safety checking for fine-grained concurrency. In *Static analysis symposium (SAS)*.

Calcagno, C., H. Yang, and P. O'Hearn. 2001. Computability and complexity results for a spatial assertion language for data structures. In *Foundations of software technology and theoretical computer science (FSTTCS)*.

Chaki, S., E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. 2004. Efficient verification of sequential and concurrent C programs. *Formal Methods in Systems Design (FMSD)* 25(2–3).

Chaki, S., and J. Ivers. 2009. Software model checking without source code. In *NASA formal methods symposium*.

Chang, B.-Y.E., and K.R.M. Leino. 2005. Abstract interpretation with alien expressions and heap structures. In *Verification, model checking, and abstract interpretation (VMCAI)*.

Chapman, Roderick, and Florian Schanda. 2014. Are we there yet? 20 years of industrial theorem proving with spark. In *Interactive theorem proving (ITP)*, 17–26. Springer.

Charette, Robert N. 2005. Why software fails. *IEEE Spectrum* 42(9):36.

Chen, L., A. Miné, and P. Cousot. 2008. A sound floating-point polyhedra abstract domain. In *Asian symposium on programming languages and systems(APLAS)*.

Chrabakh, Wahid, and Rich Wolski. 2003. Gridsat: A chaff-based distributed sat solver for the grid. In *Proceedings of the 2003 acm/ieee conference on supercomputing*, 37. ACM.

Chu, Geoffrey, Peter J Stuckey, and Aaron Harwood. 2008. Pminisat: a parallelization of minisat 2.0. *SAT race*.

Cimatti, A., and A. Griggio. 2012. Software model checking via IC3. In *Computer aided verification (CAV)*.

Cipra, Barry. 1995. How number theory got the best of the Pentium chip. *Science* 267(5195).

Clarke, E.M., D. Kroening, N. Sharygina, and K. Yorav. 2004. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in Systems Design (FMSD)* 25(2–3).

Cook, B., and G. Gonthier. 2005. Using Stålmarck's algorithm to prove inequalities. In *International conference on formal engineering methods (ICFEM)*.

Cook, B., C. Haase, J. Ouaknine, M. Parkinson, and J. Worrell. 2011. Tractable reasoning in a fragment of separation logic. In *Conference on concurrency theory (CONCUR)*.

Cousot, P., and R. Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*.

———. 1979. Systematic design of program analysis frameworks. In *ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*.

Cousot, P., R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. 2006. Combination of abstractions in the ASTRÉE static analyzer. In *Asian computing science conference (ASIAN)*.

Cousot, P., R. Cousot, and L. Mauborgne. 2011a. Logical abstract domains and interpretations. In *The future of software engineering*.

———. 2011b. The reduced product of abstract domains and the combination of decision procedures. In *Foundations of software science and computation structure (FoSSaCS)*.

Cousot, P., and N. Halbwachs. 1978. Automatic discovery of linear constraints among variables of a program. In *ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*.

Cousot, Patrick, and Michael Monerau. 2012. Probabilistic abstract interpretation. In *European symposium on programming (ESOP)*.

Cova, M., V. Felmetsger, G. Banks, and G. Vigna. 2006. Static detection of vulnerabilities in x86 executables. In *Annual computer security applications conference (ACSAC)*.

Craig, W. 1957. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic* 22(3).

Cuoq, Pascal, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. 2012. Testing static analyzers with randomly generated programs. In *International conference on NASA formal methods (NFM)*.

Darwiche, Adnan, and Knot Pipatsrisawat. 2009. Complete algorithms. In *Handbook of satisfiability*, ed. Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, vol. 185 of *Frontiers in Artificial Intelligence and Applications*, 99–130. IOS Press.

Davis, Martin, George Logemann, and Donald Loveland. 1962. A machine program for theorem-proving. *Communications of the ACM (CACM)* 5(7):394–397.

Del Val, Alvaro. 1999. A new method for consequence finding and compilation in restricted languages. In *AAAI conference on artificial intelligence (AAAI)*.

Delmas, David, and Jean Souyris. 2007. Astrée: From research to industry. In *Static analysis symposium (SAS)*.

Distefano, D., P. O'Hearn, and H. Yang. 2006. A local shape analysis based on separation logic. In *Tools and algorithms for the construction and analysis of systems (TACAS)*.

Distefano, D., and M. Parkinson. 2008. jStar: towards practical verification for Java. In *ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications(OOPSLA)*.

Driscoll, E., A. Thakur, and T. Reps. 2012. OpenNWA: A nested-word automaton library. In *Computer aided verification (CAV)*.

D'Silva, V., L. Haller, and D. Kroening. 2012. Satisfiability solvers are static analyzers. In *Static analysis symposium (SAS)*.

———. 2013. Abstract conflict driven learning. In *ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*.

D'Silva, Vijay, Leopold Haller, and Daniel Kroening. 2014. Abstract satisfaction. In *ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*.

dSPACE. 2014. dspace targetlink. `www.dspaceinc.com/en/inc/home/products/sw/pcgs/targetli.cfm`.

Dudka, K., P. Muller, P. Peringer, and T. Vojnar. 2013. Predator: A tool for verification of low-level list manipulations. In *Tools and algorithms for the construction and analysis of systems (TACAS)*.

Dutertre, B., and L. de Moura. 2006. Yices: An SMT solver. `http://yices.csl.sri.com/`.

Eén, N., A. Mishchenko, and R.K. Brayton. 2011. Efficient implementation of property directed reachability. In *Formal methods in computer-aided design (FMCAD)*.

Eén, Niklas, and Niklas Sörensson. 2003. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science (ENTCS)* 89(4):543–560.

———. 2004. An extensible SAT-solver. In *International conference on theory and applications of satisfiability testing (SAT)*.

van Eijk, C. A. J. 1998. Sequential equivalence checking without state space traversal. In *Design, automation and test in europe (DATE)*.

Eiter, Thomas, and Kewen Wang. 2008. Semantic forgetting in answer set programming. *Artificial Intelligence* 172(14):1644 – 1672.

Elder, M., J. Lim, T. Sharma, T. Andersen, and T. Reps. 2011. Abstract domains of affine relations. In *Static analysis symposium (SAS)*.

———. 2014. Abstract domains of affine relations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. (To appear).

Enea, C., V. Saveluc, and M. Sighireanu. 2013. Compositional invariant checking for overlaid and nested linked lists. In *European symposium on programming (ESOP)*.

Ernst, M.D., J.H. Perkins, P.J. Guo, S. McCamant, C. Pacheco, M.S. Tschantz, and C. Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming (SCP)* 69(1–3).

Feynman, Richard P. 1960. There's plenty of room at the bottom. *Engineering and science* 23(5): 22–36.

Flanagan, C., and K. Rustan M. Leino. 2001. Houdini, an annotation assistant for Esc/Java. In *Formal methods for increasing software productivity (FME)*.

Flanagan, C., and S. Qadeer. 2002. Predicate abstraction for software verification. In *ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*.

Garg, P., P. Madhusudan, and G. Parlato. 2013. Quantified data automata on skinny trees: An abstract domain for lists. In *Static analysis symposium (SAS)*.

Garner, H. L. 1978. Theory of computer addition and overflow. *IEEE Transactions on Computers (TOC)* C-27(4).

Garoche, P.-L., T. Kahsai, and C. Tinelli. 2012. Invariant stream generators using automatic abstract transformers based on a decidable logic. Tech. Rep. CoRR abs/1205.3758, arXiv.

Gerth, R. 1991. Formal verification of self modifying code. In *International conference for young computer scientists*, ed. Y. Liu and X. Li, 305–311. Beijing, China: Int. Acad. Pub.

Go. 2014. The go programming language. `golang.org`.

Godefroid, P., N. Klarlund, and K. Sen. 2005. DART: Directed automated random testing. In *ACM SIGPLAN conference on programming language design and implementation (PLDI)*.

Godefroid, P., M.Y. Levin, and D. Molnar. 2008. Automated whitebox fuzz testing. In *Network and distributed systems security (NDSS)*.

Godefroid, P., A.V. Nori, S.K. Rajamani, and S.D. Tetali. 2010. Compositional may-must program analysis: Unleashing the power of alternation. In *ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*.

Gomes, Carla P, Bart Selman, Nuno Crato, and Henry Kautz. 2000. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning (JAR)* 24(1-2): 67–100.

Gomes, Carla P, Bart Selman, and Henry Kautz. 1998. Boosting combinatorial search through randomization. *AAAI Conference on Artificial Intelligence (AAAI)* 98:431–437.

Gopan, D., F. DiMaio, N. Dor, T. Reps, and M. Sagiv. 2004. Numeric domains with summarized dimensions. In *Tools and algorithms for the construction and analysis of systems (TACAS)*.

Graf, S., and H. Saïdi. 1997. Construction of abstract state graphs with PVS. In *Computer aided verification (CAV)*.

Graham, S.L., and M. Wegman. 1976. A fast and usually linear algorithm for data flow analysis. *Journal of ACM (JACM)* 23(1):172–202.

Gulavani, B.S., T.A. Henzinger, Y. Kannan, A.V. Nori, and S.K. Rajamani. 2006. SYNERGY: A new algorithm for property checking. In *Foundations of software engineering (FSE)*.

Gulwani, S., and M. Musuvathi. 2008. Cover algorithms and their combination. In *European symposium on programming (ESOP)*.

Haller, Leopold. 2013. Abstract satisfaction. Ph.D. thesis, University of Oxford, Oxford, United Kingdom.

Hamadi, Y., and C. M. Wintersteiger. 2012. Seven challenges in parallel SAT solving. In *AAAI conference on artificial intelligence (AAAI)*.

Hamadi, Youssef, Said Jabbour, and Jabbour Sais. 2009a. Control-based clause sharing in parallel SAT solving. In *International joint conference on artificial intelligence (IJCAI)*.

Hamadi, Youssef, Said Jabbour, and Lakhdar Sais. 2009b. ManySAT: a parallel SAT solver. *Journal of Satisfiability, Boolean Modeling and Computation (JSAT)* 6(4):245–262.

Hamadi, Youssef, João Marques-Silva, and Christoph M. Wintersteiger. 2011. Lazy decomposition for distributed decision procedures. In *Workshop on Parallel and Distributed Methods in verifiCation (PDMC)*.

Harrison, J. 1996. Stålmarck's algorithm as a HOL Derived Rule. In *Theorem proving in higher order logics (TPHOLs)*.

Harrison, John. 2009. *Handbook of practical logic and automated reasoning*. Cambridge University Press.

Heizmann, M., J. Hoenicke, and A. Podelski. 2010. Nested interpolants. In *ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*.

Henry, Julien, Mihail Asavoae, David Monniaux, and Claire Maıza. 2014. How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. In *Languages, compilers and tools for embedded systems (LCTES)*.

Henzinger, T.A., R. Jhala, R. Majumdar, and K. L. McMillan. 2004. Abstractions from proofs. In *ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*, 232–244.

Henzinger, T.A., R. Jhala, R. Majumdar, and G. Sutre. 2002. Lazy abstraction. In *ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*, 58–70.

Hoder, K., and N. Bjørner. 2012. Generalized property directed reachability. In *International conference on theory and applications of satisfiability testing (SAT)*.

Hoffmann, Jörg, and Jana Koehler. 1999. A new method to index and query sets. In *International joint conference on artificial intelligence (IJCAI)*, vol. 99, 462–467.

Holzmann, Gerard J. 2004. *The SPIN model checker: Primer and reference manual*, vol. 1003. Addison-Wesley Reading.

Hou, Z., R. Clouston, R. Gore, and A. Tiu. 2014. Proof search for propositional abstract separation logics via labelled sequents:. In *ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*.

Howard, M. 2002. Some bad news and some good news. Http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure10102002.asp.

Hutter, Frank, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. 2009. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research* 36: 267–306.

Itzhaky, S., A. Banerjee, N. Immerman, A. Nanevski, and Mooly Sagiv. 2013. Effectively-propositional reasoning about reachability in linked data structures. In *Computer aided verification (CAV)*.

Itzhaky, S., N. Bjorner, T. Reps, M. Sagiv, and A. Thakur. 2014. Property-directed shape analysis. In *Computer aided verification (CAV)*.

Jeannet, B., A. Loginov, T. Reps, and M. Sagiv. 2010. A relational approach to interprocedural shape analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 32(2).

Jetley, Raoul Praful, Paul L Jones, and Paul Anderson. 2008. Static analysis of medical device software using codesonar. In *Proceedings of the 2008 workshop on static analysis (saw)*.

Jhala, Ranjit, and Kenneth L McMillan. 2006. A practical and complete approach to predicate refinement. In *Tools and algorithms for the construction and analysis of systems (TACAS)*.

Karp, Richard M. 2011. Understanding science through the computational lens. *Journal of Computer Science and Technology* 26(4):569–577.

Karr, M. 1976. Affine relationship among variables of a program. *Acta Informatica* 6:133–151.

Katsirelos, G., A. Sabharwal, H. Samulowitz, and L. Simon. 2013. Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers. In *AAAI conference on artificial intelligence (AAAI)*.

Kautz, Henry, and Bart Selman. 1992. Planning as satisfiability. In *Proceedings of the 10th european conference on artificial intelligence*, 359–363. ECAI '92.

Kearns, Michael J., and Umesh V. Vazirani. 1994. *An introduction to computational learning theory*. Cambridge, MA, USA: MIT Press.

Kidd, N., A. Lal, and T. Reps. 2007. WALi: The Weighted Automaton Library. `www.cs.wisc.edu/wpis/wpds/download.php`.

King, A., and H. Søndergaard. 2010. Automatic abstraction for congruences. In *Verification, model checking, and abstract interpretation (VMCAI)*.

Knoop, J., and B. Steffen. 1992. The interprocedural coincidence theorem. In *Compiler construction (CC)*.

Konev, Boris, Dirk Walther, and Frank Wolter. 2009. Forgetting and uniform interpolation in large-scale description logic terminologies. In *International joint conference on artificial intelligence (IJCAI)*, 830–835.

Kovásznai, Gergely, Andreas Fröhlich, and Armin Biere. 2012. On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In *International workshop on satisfiability modulo theories (SMT)*.

Kozen, Dexter. 1981. Semantics of probabilistic programs. *Journal of Computer and System Sciences* 22(3):328–350.

Kroening, D., and O. Strichman. 2008. *Decision procedures: An algorithmic point of view*. Texts in Theoretical Computer Science, Springer-Verlag.

Kruegel, C., E. Kirda, D. Mutz, W. Robertson, and G. Vigna. 2005. Automating mimicry attacks using static binary analysis. In *USENIX security symposium*.

Kuncak, V., and M. Rinard. 2003. On the boolean algebra of shape analysis constraints. Technical Report MIT-LCS-TR-916, M.I.T. CSAIL.

Kunz, W., and D.K. Pradhan. 1994. Recursive learning: A new implication technique for efficient solutions to CAD problems–test, verification, and optimization. *IEEE Transactions on CAD of Integrated Circuits and Systems* 13(9):1143–1158.

Lal, A., J. Lim, and T. Reps. 2009. MCDASH: Refinement-based property verification for machine code. TR 1659, UW-Madison.

Lal, Akash, Shaz Qadeer, and Shuvendu K. Lahiri. 2012. A solver for reachability modulo theories. In *Computer aided verification (CAV)*.

Lassez, J, M. Maher, and K. Marriott. 1988. Unification revisited. In *Foundations of logic and functional programming*, vol. 306, 67–113. Springer.

Lee, W., and S. Park. 2014. A proof system for separation logic with magic wand. In *ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*.

Lev-Ami, T., and M. Sagiv. 2000. TVLA: A system for implementing static analyses. In *Static analysis symposium (SAS)*.

Li, Yi, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. 2014. Symbolic optimization with smt solvers. In *ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*.

Lim, J., A. Lal, and T. Reps. 2009. Symbolic analysis via semantic reinterpretation. In *SPIN model checking and software verification*.

Lim, J., and T. Reps. 2008. A system for generating static analyzers for machine instructions. In *Compiler construction (CC)*.

———. 2013. Tsl: A system for generating abstract interpreters and its application to machine-code analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 35(1):4:1–4:59.

Lin, Fangzhen, and Ray Reiter. 1994. Forget it! In *Working notes of aaai fall symposium on relevance*.

Lin, Fangzhen, and Raymond Reiter. 1997. How to progress a database. *Artificial Intelligence* 92(1-2):131–167.

Linn, C., and S.K. Debray. 2003. Obfuscation of executable code to improve resistance to static disassembly. In *ACM conference on computer and communications security (CCS)*.

Livshits, B., and M. Lam. 2005. Finding security vulnerabilities in Java applications with static analysis. In *USENIX security symposium*.

Luby, Michael, Alistair Sinclair, and David Zuckerman. 1993. Optimal speedup of las vegas algorithms. *Information Processing Letters (IPL)* 47(4):173–180.

Magill, S., J. Berdine, E. Clarke, and B. Cook. 2007. Arithmetic strengthening for shape analysis. In *Static analysis symposium (SAS)*.

Marques-Silva, Joao, Ines Lynce, and Sharad Malik. 2009. CDCL solvers. In *Handbook of satisfiability*, ed. Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, vol. 185 of *Frontiers in Artificial Intelligence and Applications*, 131–154. IOS Press.

Marques Silva, J.P., and K.A. Sakallah. 1996. GRASP – a new search algorithm for satisfiability. In *International conference on computer-aided design (ICCAD)*.

Martins, Ruben, Vasco Manquinho, and Inês Lynce. 2010. Improving search space splitting for parallel SAT solving. In *IEEE international conference on tools with artificial intelligence (ICTAI)*.

Masdupuy, F. 1992. Array abstractions using semantic analysis of trapezoid congruences. In *International conference on supercomputing (ICS)*.

Massacci, Fabio, and Laura Marraro. 2000. Logical cryptanalysis as a SAT problem. *Journal of Automated Reasoning (JAR)* 24(1-2):165–203.

McCloskey, B., T.W. Reps, and M. Sagiv. 2010. Statically inferring complex heap, array, and numeric invariants. In *Static analysis symposium (SAS)*.

McIlraith, Sheila, and Eyal Amir. 2001. Theorem proving with structured theories. In *International joint conference on artificial intelligence (IJCAI)*.

McMillan, Kenneth L. 2005. Don't-care computation using k-clause approximation. *International Workshop on Logic and Synthesis (IWLS)* 153–160.

McMillan, Kenneth L. 2010. Lazy annotation for program testing and verification. In *Computer aided verification (CAV)*.

McMillan, Kenneth L., Andreas Kuehlmann, and Mooly Sagiv. 2009. Generalizing DPLL to richer logics. In *Computer aided verification (CAV)*.

Miné, A. 2001. The octagon abstract domain. In *Working conference on reverse engineering (WCRE)*, 310–322.

———. 2004. Relational abstract domains for the detection of floating-point run-time errors. In *European symposium on programming (ESOP)*.

Mitchell, T.M. 1997. *Machine learning*. Boston, MA: WCB/McGraw-Hill.

Monniaux, D. 2009. Automatic modular abstractions for linear constraints. In *ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*.

———. 2010. Automatic modular abstractions for template numerical constraints. *Logical Methods in Computer Science (LMCS)* 6(3).

Monniaux, David. 2000. Abstract interpretation of probabilistic semantics. In *Static analysis symposium (SAS)*.

Moskewicz, M., C. Madigan, Y. Zhao, L. Zhang, and S. Malik. 2001. Chaff: Engineering an efficient SAT solver. In *Design automation conference (DAC)*, 530–535.

de Moura, L., and N. Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and algorithms for the construction and analysis of systems (TACAS)*.

Müller-Olm, M., and H. Seidl. 2004. Precise interprocedural analysis through linear algebra. In *ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*.

———. 2005. Analysis of modular arithmetic. In *European symposium on programming (ESOP)*.

Nielson, F., H.R. Nielson, and C. Hankin. 1999. *Principles of program analysis*. Springer-Verlag.

Park, J., J. Seo, and S. Park. 2013. A theorem prover for Boolean BI. In *ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*.

Pérez, J. A. Navarro, and A. Rybalchenko. 2011. Separation logic + superposition calculus = heap theorem prover. In *ACM SIGPLAN conference on programming language design and implementation (PLDI)*.

Pipatsrisawat, Knot, and Adnan Darwiche. 2007. A lightweight component caching scheme for satisfiability solvers. In *International conference on theory and applications of satisfiability testing (SAT)*.

Piskac, R., T. Wies, and D. Zufferey. 2013. Automating separation logic using SMT. In *Computer aided verification (CAV)*.

Plotkin, G. 1970. A note on inductive generalization. *Machine Intelligence* 5(1):153–165.

protobuf. 2014. Protocol buffers - google's data interchange format. `https://code.google.com/p/protobuf/`.

Regehr, J., and A. Reid. 2004. HOIST: A system for automatically deriving static analyzers for embedded systems. In *Architectural support for programming languages and operating systems (ASPLOS)*.

Reps, T., G. Balakrishnan, and J. Lim. 2006. Intermediate-representation recovery from low-level code. In *Partial evaluation and semantics-based program manipulation (PEPM)*.

Reps, T., A. Lal, and N. Kidd. 2007. Program analysis using weighted pushdown systems. In *Foundations of software technology and theoretical computer science (FSTTCS)*.

Reps, T., M. Sagiv, and G. Yorsh. 2004. Symbolic implementation of the best transformer. In *Verification, model checking, and abstract interpretation (VMCAI)*.

Reynolds, J.C. 1970. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence* 5(1):135–151.

———. 2002. Separation logic: A logic for shared mutable data structures. In *Logic in computer science (LICS)*.

Rintanen, Jussi. 2009. Planning and sat. In *Handbook of satisfiability*, ed. Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, vol. 185 of *Frontiers in Artificial Intelligence and Applications*, 483–504. IOS Press.

Roussel, Olivier. 2012. Description of ppfolio 2012. In *Proceedings of SAT competition*.

Sagiv, M., T. Reps, and R. Wilhelm. 2002. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 24(3):217–298.

Sankaranarayanan, S., H.B. Sipma, and Z. Manna. 2005. Scalable analysis of linear systems using mathematical programming. In *Verification, model checking, and abstract interpretation (VMCAI)*.

SAT-COMP'13. 2013. SAT competition 2013. `http://satcompetition.org/2013/`.

Schlich, B. 2008. Model checking of software for microcontrollers. Ph.D. thesis, RWTH Aachen University, Germany.

Selman, B., and H. Kautz. 1996. Knowledge compilation and theory approximation. *Journal of ACM (JACM)* 43(2):193–224.

Sen, R., and Y.N. Srikant. 2007. Executable analysis using abstract interpretation with circular linear progressions. In *Executable analysis using abstract interpretation with circular linear progressions (MEMOCODE)*.

Sharir, M., and A. Pnueli. 1981. Two approaches to interprocedural data flow analysis. In *Program flow analysis: Theory and applications*. Prentice-Hall.

Sheeran, M., and G. Stålmarck. 2000. A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods in Systems Design (FMSD)* 16(1):23–58.

Simon, A., and A. King. 2007. Taming the wrapping of integer arithmetic. In *Static analysis symposium (SAS)*.

Simon, Laurent, and Alvaro Del Val. 2001. Efficient consequence finding. In *International joint conference on artificial intelligence (IJCAI)*, vol. 1, 359–365.

Srivastava, A., A. Edwards, and H. Vo. 2001. Vulcan: Binary transformation in a distributed environment. MSR-TR-2001-50, Microsoft Research.

Steffen, B., J. Knoop, and O. Rüthing. 1990. The value flow graph: A program representation for optimal program transformations. In *European symposium on programming (ESOP)*.

———. 1991. Efficient code motion and an adaption to strength reduction. In *Theory and practice of software development (TAPSOFT)*.

Storjohann, A. 2000. Algorithms for matrix canonical forms. Ph.D. thesis, ETH Zurich, Zurich, Switzerland. Diss. ETH No. 13922.

Stålmarck, G. 1989. A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula. Swedish Patent No. 467,076 (approved 1992); U.S. Patent No. 5,276,897 (approved 1994); European Patent No. 403,454 (approved 1995).

Thakur, A., A. Lal, J. Lim, and T. Reps. 2013. Posthat and all that: Attaining most-precise inductive invariants. TR 1790, CS Dept., Univ. of Wisconsin, Madison, WI.

Thakur, A., J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. Reps. 2010. Directed proof generation for machine code. TR 1669, UW-Madison.

Thakur, A., and T. Reps. 2012. A Generalization of Stålmarck's Method. In *Static analysis symposium (SAS)*.

Vafeiadis, V., and M. Parkinson. 2007. A marriage of rely/guarantee and separation logic. In *Conference on concurrency theory (CONCUR)*.

Valiant, Leslie G. 1984. A theory of the learnable. *Communications of the ACM (CACM)* 27(11): 1134–1142.

Visser, Albert. 1996. Uniform interpolation and layered bisimulation. In *Gödel*, 139–164. Springer.

Wagner, D., J. Foster, E. Brewer, and A. Aiken. 2000. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and distributed systems security (NDSS)*.

Wang, Xi, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *ACM symposium on operating systems principles (SOSP)*.

Wang, Zhe, Kewen Wang, Rodney Topor, and Jeff Z Pan. 2010. Forgetting for knowledge bases in dl-lite. *Annals of Mathematics and Artificial Intelligence* 58(1-2):117–151.

Warren, H.S., Jr. 2003. *Hacker's delight*. Addison-Wesley.

Xie, Y., and A. Aiken. 2006. Static detection of security vulnerabilities in scripting languages. In *USENIX security symposium*.

Yang, Xuejun, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *ACM SIGPLAN conference on programming language design and implementation (PLDI)*.

Yorsh, G., T. Ball, and M. Sagiv. 2006. Testing, abstraction, theorem proving: Better together! In *International symposium on software testing and analysis (ISSTA)*.

Yorsh, G., T. Reps, and M. Sagiv. 2004. Symbolically computing most-precise abstract operations for shape analysis. In *Tools and algorithms for the construction and analysis of systems (TACAS)*.

Yorsh, G., T. Reps, M. Sagiv, and R. Wilhelm. 2007. Logical characterizations of heap abstractions. *ACM Transactions on Computational Logic (TOCL)* 8(1).