Automating Abstract Interpretation*

Thomas Reps[1,2] and Aditya Thakur[3]

[1] University of Wisconsin; Madison, WI, USA
[2] GrammaTech, Inc.; Ithaca, NY, USA
[3] Google, Inc.; Mountain View, CA USA

Abstract. Abstract interpretation has a reputation of being a kind of "black art," and consequently difficult to work with. This paper describes a twenty-year quest by the first author to address this issue by raising the level of automation in abstract interpretation. The most recent leg of this journey is the subject of the second author's 2014 Ph.D. dissertation. The paper discusses several different approaches to creating correct-by-construction analyzers. Our research has allowed us to establish connections between this problem and several other areas of computer science, including automated reasoning/decision procedures, concept learning, and constraint programming.

1 Introduction

Establishing that a program is correct is undecidable in general. Consequently, program-analysis and verification tools typically work on an *abstraction* of a program, which over-approximates the original program's behavior. The theory underlying this approach is called *abstract interpretation* [18]. Abstract interpretation provides a way to create program analyzers that obtain information about the possible states that a program reaches during execution, but without actually running the program on specific inputs. Instead, the analyzer executes the program using finite-sized descriptors that represent *sets* of states. For example, one can use descriptors that represent only the *sign* of a variable's value: `neg`, `zero`, `pos`, or `unknown`. If the abstract state maps variables x and y as follows, $[x \mapsto \texttt{neg},\ y \mapsto \texttt{neg}]$, the product "$x * y$" would be performed as "$\texttt{neg} * \texttt{neg}$," yielding `pos`. This approximation discards information about the specific *values* of x and y; $[x \mapsto \texttt{neg},\ y \mapsto \texttt{neg}]$ represents all concrete states in which x and y hold negative integers. By using such descriptors to explore the program's behavior for *all* possible inputs, the analyzer accounts for all possible states that the program can reach.

The tar-pit of undecidability is sidestepped via two concepts:

– ***Abstraction.*** In this context, abstraction means "representing an information space by a smaller space that captures its essential features." (The smaller space is called an *abstract domain*; an example of an abstract domain is the set of all descriptors that record the signs of variables, as used above.)

* Portions of this work appeared in [70, 63, 35, 45, 64, 26, 82, 78, 81, 66, 76]. T. Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology reported in this publication.

– ***One-sided analysis.*** Whenever the analyzer says "no" it means "no," but whenever it says "yes" it means "maybe-yes/maybe-no"—i.e., the property might or might not hold.

When the analyzer reports "no, a bad state is not reachable," one is guaranteed that only good states can arise—and hence that the program is correct with respect to the property being checked. If the analyzer reports "yes, a bad state might be reachable," it must try other techniques to attempt to establish the desired property (e.g., refining the abstraction in use).

However, there is a glitch: abstract interpretation has a reputation of being a kind of "black art," and consequently difficult to work with. This paper describes a twenty-year quest to make abstract interpretation easier to work with by (i) raising the level of discourse for specifying abstract interpreters, and (ii) automating some of abstraction interpretation's more difficult aspects, thereby making it possible to create correct-by-construction analyzers.

A major focus of the work has been how to automate the construction of the functions to transform abstract states—also known as *abstract transformers*. The motivation came from our experience with two challenging analysis contexts:

***Analysis of programs manipulating linked data structures*:** When analyzing such programs, the number of fine-grained details that one needs to track causes the abstractions to be inherently complex.

***Analysis of stripped machine code*:** Here an analyzer needs to use multiple (separate and cooperating) abstract interpretations [6, 45], and we also had the goal of creating machine-code-analysis tools for multiple instruction sets.

In both cases, our experience with hand construction of abstract transformers [69, 6] was that the process was tedious, time-consuming, and a source of errors.

The paper summarizes three major milestones of our research, based on different approaches that we explored.

1. The TVLA system [70, 42, 12] introduced a way to create abstractions of systems specified in first-order logic, plus transitive closure (§3). To construct abstract transformers in TVLA, we developed a non-standard approach to weakest precondition based on a finite-differencing transformation [63].
2. The TSL system [45] supports the creation of correct-by-construction implementations of the abstract transformers needed in tools that analyze machine code (§4). From a single specification of the concrete semantics of an instruction set, TSL can generate abstract transformers for static analysis, dynamic analysis, symbolic analysis, or any combination of the three.
3. Our work on symbolic methods for abstract interpretation [64, 82, 78] aims to bridge the gap between (i) the use of logic for specifying program semantics and program correctness, and (ii) abstract interpretation. Many of the issues, including the construction of abstract transformers, can be reduced to the problem of *symbolic abstraction* (§5):

> Given formula φ in logic $\mathcal{L}$, and abstract domain $\mathbb{A}$, find the most-precise descriptor $a^\sharp$ in $\mathbb{A}$ that over-approximates the meaning of φ.

A particularly exciting aspect of the work on symbolic abstraction is the number of links the problem has with other research areas that one would not normally think of as being connected to static program analysis. Our investigations have established connections with such areas as automated reasoning/decision procedures (§5.4), concept learning (§6.1), and constraint programming (§6.2).

§7 discusses related work. §8 concludes with a few final insights and takeaways.

2 Problem Statement

2.1 What Can Be Automated About Abstract Interpretation?

A static-analysis system can have many components, including

(i) construction and use of abstract transformers
  - an algorithm to construct sound abstract transformers to model the actions of language primitives and/or user-defined functions
  - an algorithm to apply or compose abstract transformers

(ii) state-space exploration
  - state-space-exploration algorithms (i.e, equation/constraint solvers)
  - methods to enforce termination via widening policies
  - containment algorithms (for determining whether state-space exploration should terminate)

(iii) mechanisms for improving precision
  - narrowing
  - reduced product
  - semantic reduction
  - construction of best transformers
  - determination of the best inductive invariant

(iv) abstraction refinement (enabled by item (i))

While the first author has also done a lot of work on state-space-exploration algorithms [62, 65, 67] and some on widening policies [29, 30], because so many of the other aspects of the problem of automating abstract interpretation are enabled by automating the construction (and use) of abstract transformers, the paper will focus on work he and his collaborators have carried out on that topic. In §5, we discuss recent work on a uniform mechanism to construct abstract transformers that also provides a way to address reduced product, semantic reduction, and (for some abstract domains) finding the best inductive invariant.

To create sound abstract transformers that use a given abstract domain, we need to have some way to create the abstract analogs of

(I) each constant that can be denoted in the programming language

(II) each primitive operation in the programming language

(III) each user-defined function in every program to be analyzed.

Task (I) is related to defining the *abstraction function* α; to create the abstract analog $k^\sharp$ of concrete constant k, apply α; i.e., $k^\sharp = \alpha(\{k\})$. By an abstract analog of a concrete operation/function f, we mean an abstract operation/function $f^\sharp$ that satisfies

$$\alpha(\tilde{f}(V_1, \ldots, V_k)) \sqsubseteq f^\sharp(\alpha(V_1), \ldots, \alpha(V_k)), \tag{1}$$

where $\tilde{f}$ denotes the lifting of f to operate on a set of values, i.e., $\tilde{f}(V_1, \ldots, V_k) = \{f(v_1, \ldots, v_k) \mid v_1 \in V_1, \ldots, v_k \in V_k\}$, and $\sqsubseteq$ denotes an ordering on abstract values that respects concrete containment; i.e., $a_1^\sharp \sqsubseteq a_2^\sharp$ implies $\gamma(a_1^\sharp) \subseteq \gamma(a_2^\sharp)$, where γ denotes the *concretization function* for the abstract domain.

The effort that has to go into task (II) is bounded—the language has a fixed number of primitive operations—and task (II) only has to be done once for a given abstract domain. However, task (III) needs automation, because it will be performed for all functions in all users' programs, which are not known *a priori*.

2.2 Non-Compositionality

Unfortunately, abstract interpretation is *inherently non-compositional*—meaning that one cannot create abstract analogs of operations/functions separately, and put them together without losing precision (see below). The non-compositionality property is the essence of what makes it hard to automate the construction of abstract transformers. This message is an uncomfortable one for computer scientists because compositionality is so ingrained in our training—e.g., our programming languages are defined using context-free grammars; many concepts and properties are defined using inductive definitions, and recursive tree traversals are a basic workhorse.

Syntax-Directed Replacement. A compositional approach to constructing sound abstract transformers is relatively easy to implement. In particular, Eqn. (1) makes possible a simple, compositional approach—namely, syntax-directed replacement of the concrete constants and concrete primitive operations by their abstract analogs. For instance, consider the following function: $f(x_1, x_2) = x_1 * x_2 + 1$. First, hoist f to $\tilde{f}$, i.e., $\tilde{f}(X_1, X_2) = X_1 \tilde{*} X_2 \tilde{+} \{1\}$. Then, by Eqn. (1), we have

$$\alpha(\tilde{f}(X_1, X_2)) = \alpha(X_1 \tilde{*} X_2 \tilde{+} \{1\}) \sqsubseteq \alpha(X_1 \tilde{*} X_2) +^\sharp \{1\}^\sharp \sqsubseteq \alpha(X_1) *^\sharp \alpha(X_2) +^\sharp \{1\}^\sharp.$$

Thus, one way to ensure that we have a sound $f^\sharp$ is to define $f^\sharp(x_1, x_2)$ by

$$f^\sharp(x_1, x_2) \stackrel{\text{def}}{=} x_1 *^\sharp x_2 +^\sharp \{1\}^\sharp.$$

Drawbacks of Syntax-Directed Replacement. Although syntax-directed replacement is simple and compositional, it can be quite myopic because it focuses solely on what happens at a single production in the abstract syntax tree. The approach can lead to a loss of precision by not accounting for correlations between operations at far-apart positions in the abstract syntax tree.

To illustrate the issue, consider the function $h(x) \stackrel{\text{def}}{=} x + (-x)$. Obviously, $h(x)$ always returns 0. Now suppose that we apply syntax-directed replacement, $h^\sharp(x) \stackrel{\text{def}}{=} x +^\sharp (-^\sharp x)$, and evaluate $h^\sharp$ over the *sign abstract domain*, which consists of six values: $\{\texttt{neg}, 0, \texttt{pos}, \texttt{nonpos}, \texttt{nonneg}, \top\}$. In particular, the abstract unary-minus operation is defined as follows:

x	$\top$	nonneg	nonpos	pos	zero	neg
$-^\sharp x$	$\top$	nonpos	nonneg	neg	zero	pos

Consider evaluating $h^\sharp(x)$ with the abstract value pos for the value of x. (Abstract values at leaves and internal nodes of the AST of $h^\sharp$'s defining expression

are shown within square brackets in the tree in Fig. 1.) Because $\texttt{pos} +^\sharp \texttt{neg} = \top$, we obtain no useful information from the abstract interpretation. In contrast, the concrete value is always 0, and therefore the most-precise abstract answer is `zero` (because $\alpha(\{0\}) = \texttt{zero}$).

Artificially imposing compositionality on an abstract interpreter has a number of drawbacks:

- compositionality at expression granularity may not produce the best abstraction, even if all abstract program primitives are best abstract primitives
- compositionality at statement or basic-block level may not produce the best transformer, even if each abstract transformer being composed is a best transformer

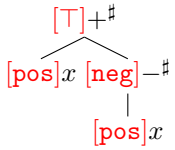

Fig. 1: Abstract subtraction when leaves are correlated.

Moreover, if an analyzer loses precision at one point in a program, it can provoke a cascade of precision loss throughout the program.

2.3 What Does It Mean to Automate the Construction of Abstract Transformers?

We sometimes describe our work by saying that we are working on "a `yacc` for automating the construction of abstract transformers," by which we mean a tool that automates the task to an extent similar to the automation of the construction of parsers achieved by `yacc` [36]. As a model for what we would like to achieve, consider the problem that `yacc` addresses:

- An instance of a parsing problem, Parse(L,s), has two parameters: L, a context-free language; and s, a string to be parsed. String s changes more frequently than language L.
- Context-free grammars are a formalism for specifying context-free languages.
- Create a tool that implements the following specification:
  - Input: a context-free grammar that describes language L.
  - Output: a parsing function, `yyparse()`, for which executing `yyparse()` on string s computes Parse(L,s).

Thus, we would like to follow a similar scheme.

- An abstract interpreter $\text{Interp}^\sharp(M_s, \mathbb{A}, a^\sharp)$ has three inputs
  - M_s = the meaning function for a programming-language statement s
  - $\mathbb{A}$ = an abstract domain
  - $a^\sharp$ = an abstract-domain value (which represents a set of pre-states)

  $a^\sharp$ changes more frequently than M_s and $\mathbb{A}$.
- Find appropriate formalisms F_1 and F_2 for specifying M_s and $\mathbb{A}$.
- Create a tool that implements the following specification:
  - Input:
    - an F_1 specification of the programming language's semantics
    - an F_2 specification that characterizes the abstraction that $\mathbb{A}$ supports
  - Output: a function $I_{s,\mathbb{A}}(\cdot)$ such that $I_{s,\mathbb{A}}(a^\sharp)$ computes $\text{Interp}^\sharp(M_s, \mathbb{A}, a^\sharp)$

An alternative goal for the tool's output is as follows:

Output: a *representation* of the function $I_{s,\mathbb{A}}(\cdot)$ that can be used in the function-composition operations performed by interprocedural dataflow analyzers [74].

Relationship to Partial Evaluation. Readers who are familiar with partial evaluation [28, 37] may be struck by how similar the problem statement above is to the specification of partial evaluation, which suggests that partial evaluation could play a role in automating abstract interpretation. However, we believe that this observation is a red herring: whereas partial evaluation provides a mechanism to speed up computations by removing interpretive overhead, the key question in automating the construction of abstract transformers is "*Given the specification of an abstraction, how does one create an execution engine for an analyzer that performs computations in an over-approximating fashion?*"

2.4 Four Questions

The above discussion suggests four questions to ask about methods for automating the construction of abstract transformers:

Q1. What formalism is used to specify M_s?
Q2. What formalism is used to specify $\mathbb{A}$?
Q3. What is the engine at work that applies/constructs abstract transformers?
  (a) What method is used to create $I_{s,\mathbb{A}}(\cdot)$?
  (b) Can it be used to create a representation of $I_{s,\mathbb{A}}(\cdot)$?
Q4. How is the non-compositionality issue discussed in §2.2 addressed?

The answers given in §3, §4, and §5 explain how these issues are addressed in the three approaches described in the paper.

3 TVLA: 3-Valued Logic Analyzer

In 1999, Sagiv, Reps, and Wilhelm devised an abstraction method, called *canonical abstraction* [70], for analyzing the properties of evolving logical structures. The original motivation for developing canonical-abstraction domains was the desire to apply abstract interpretation to imperative programs that manipulate linked data structures, to check such properties as

– when the input to a list-insert program is an acyclic list, the output is an acyclic list, and
– when the input to a list-reversal program that uses destructive-update operations is an acyclic list, the output is an acyclic list.

Such analysis problems are known generically as *shape-analysis* problems. In programs that manipulate linked data structures, storage cells can be dynamically allocated and freed, and structure fields can be destructively updated. Data structures can thus grow and shrink, with no fixed upper bound on their size or number. In the case of thread-based languages, such as Java, the number of threads can also grow and shrink dynamically [84]. The challenge in shape analysis is to find a way to create finite-sized descriptors of memory configurations that (i) abstract away certain details, but (ii) retain enough key information so that an analyzer can identify interesting node-linkage properties that hold.

A *logical structure* is a set of *individuals* together with a certain collection of relations over the individuals. (In shape analysis, individuals represent entities such as memory locations, threads, locks, etc.; unary and binary relations encode the contents of variables, pointer-valued structure fields, and other aspects of memory states; and first-order formulas with transitive closure are used to specify properties such as sharing, cyclicity, reachability, etc.) Because canonical abstraction is a general method for abstracting logical structures, it actually has much broader applicability for analyzing systems than just shape-analysis problems. It is relevant to the analysis of any system that can be modeled as an evolving logical structure [42, 12, 34, 11].

The concrete semantics of a system—such as the concrete semantics of programs written in a given programming language—is defined using a fixed set of *core relation symbols* $\mathcal{C}$. (Different kinds of systems, such as different programming languages, are defined by varying the symbols in $\mathcal{C}$.) The concrete semantics expresses how a program statement *st* causes the core relations to change. The semantics of *st* is specified with formulas in first-order logic plus transitive closure over the client-defined core relations in $\mathcal{C}$.

Different abstract domains are defined using canonical abstraction by

- Defining a set of *instrumentation relations* $\mathcal{I}$ (also known as *derived relations* or *views*). Each instrumentation relation $p(v)$ is defined by a formula $\psi_p(v)$ over the core relations.
- Choosing a set of unary *abstraction relations* $\mathcal{A}$ from among the unary relations in the vocabulary $\mathcal{R} \stackrel{\text{def}}{=} (\mathcal{C} \uplus \mathcal{I})$.

$\mathcal{I}$ controls what information is maintained (in addition to the core relations); $\mathcal{A}$ controls what individuals are indistinguishable. The two mechanisms are connected because it is possible to declare unary instrumentation relations as abstraction relations. An *abstract logical structure* is the quotient of a concrete logical structure with respect to the sets of indistinguishable individuals.

The TVLA (Three-Valued-Logic Analyzer) system [42, 12] automates some of the more difficult aspects of working with canonical-abstraction domains. However, the initial version of TVLA failed to meet our goal of automating abstract interpretation because not all aspects of abstract transformers were derived automatically from the specification of a given abstraction. The analysis designer had to supply a key portion of every abstract transformer manually.

The introduction of instrumentation relations causes auxiliary information to be recorded in a program state, such as whether an individual memory location possesses (or does not possess) a certain property. The concrete semantics expresses how a program statement *st* causes the core relations to change; the challenge is how one should go about updating the instrumentation relations. Canonical-abstraction domains are based on 3-valued logic, where the third truth value (1/2) arises when it is not known whether a property holds or not. Suppose that $p(v) \in \mathcal{I}$ is defined by $\psi_p(v)$. Reevaluating $\psi_p(v)$ almost always yields 1/2, and thus completely defeats the purpose of having augmented logical structures with instrumentation relation p.

Table 1: Core relations for shape analysis of programs that manipulate linked lists.

Relation	Intended Meaning
$eq(v_1, v_2)$	Do v_1 and v_2 denote the same memory cell?
$x(v)$	Does pointer variable `x` point to memory cell v?
$n(v_1, v_2)$	Does the n-field of v_1 point to v_2?

To overcome this effect, the initial version of TVLA required an analysis designer to specify a *relation-maintenance formula* for each instrumentation relation, for each kind of statement in the language being analyzed. This approach could obtain more precise results than that of reevaluating $\psi_p(v)$, but placed the onus on the analysis designer to supply a key part of every abstract transformer, which was both burdensome and a source of errors.

In 2002, we developed a way to create relation-maintenance formulas—and thereby abstract transformers—fully automatically [63]. Our solution to the problem is based on a finite-differencing transformation. Finite-differencing turns out to be a natural way to identify the "footprint" of statement *st* on an instrumentation relation p, which reduces the number of tuples in p that have to be reevaluated (compared to reevaluating *all* of p's tuples using $\psi_p(v)$).

2-Valued Logical Structures. A concrete state is a 2-*valued logical structure*, which provides an interpretation of a vocabulary $\mathcal{R} = \{eq, p_1, \ldots, p_n\}$ of relation symbols (with given arities). $\mathcal{R}_k$ denotes the set of k-ary symbols.

Definition 1. *A* 2-***valued logical structure*** S *over* $\mathcal{R}$ *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* $p \in \mathcal{R}_i$*,* $\iota(p)\colon U^i \to \mathbb{B}$*. We assume that* $eq \in \mathcal{R}_2$ *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* $\mathcal{R}$ *is denoted by* $\mathcal{S}_2[\mathcal{R}]$*.*

A concrete state is modeled by a 2-valued logical structure over a fixed vocabulary $\mathcal{C} \subseteq \mathcal{R}$ of *core relations*. Tab. 1 lists the core relations that are used to represent a program state made up of linked lists. The set of unary core relations, $\mathcal{C}_1$, contains relations that encode the pointer variables of the program: a unary relation of the form $x(v) \in \mathcal{C}_1$ encodes pointer variable $\mathtt{x} \in \mathit{Var}$. The binary relation $n(v_1, v_2) \in \mathcal{C}_2$ encodes list-node linkages.

$\mathcal{R}$ does not include constant or function symbols. Constant symbols are encoded via unary relations, and k-ary functions via $k + 1$-ary relations. In both cases, we use *integrity rules*—i.e., global constraints that restrict the set of structures considered to ones that we intend. The following integrity rules restrict each unary relation x, for $\mathtt{x} \in \mathit{Var}$, to serve as a constant, and restrict binary relation n to encode a partial function:

$$\text{for each } \mathtt{x} \in \mathit{Var}, \forall v_1, v_2 : x(v_1) \wedge x(v_2) \Rightarrow eq(v_1, v_2)$$
$$\forall v_1, v_2, v_3 : n(v_3, v_1) \wedge n(v_3, v_2) \Rightarrow eq(v_1, v_2)$$

3-Valued Structures, Embedding, and Canonical Abstraction. A 3-

valued logical structure provides a finite over-approximation of a possibly infinite set of 2-valued structures. The set $\mathbb{T} \stackrel{\text{def}}{=} \{0, 1, 1/2\}$ of 3-valued truth values is partially ordered under the *information order*: $l \sqsubset 1/2$ for $l \in \{0, 1\}$. 0 and 1 are *definite* values; 1/2, which denotes uncertainty, is an *indefinite* value. The symbol $\sqcup$ denotes the least-upper-bound operation with respect to $\sqsubseteq$.

Definition 2. *A* 3-***valued logical structure*** $S = \langle U, \iota \rangle$ *is almost identical to a* 2*-valued structure, except that* ι *maps each* $p \in \mathcal{R}_i$ *to a* 3*-valued function* $\iota(p) \colon U^i \to \mathbb{T}$*. In addition, (i) for all* $u \in U$*,* $\iota(eq)(u, u) \sqsupseteq 1$*, and (ii) for all* $u_1, u_2 \in U$ *such that* u_1 *and* u_2 *are distinct individuals,* $\iota(eq)(u_1, u_2) = 0$*. (An individual* u *for which* $\iota(eq)(u, u) = 1/2$ *is called a* ***summary individual****.)*

The set of 3*-valued logical structures over* $\mathcal{R}$ *is denoted by* $\mathcal{S}_3[\mathcal{R}] \supsetneq \mathcal{S}_2[\mathcal{R}]$*. Given* $S = \langle U, \iota \rangle, S' = \langle U', \iota' \rangle \in \mathcal{S}_3[\mathcal{R}]$*, and surjective function* $f \colon U \to U'$*,* f ***embeds*** S ***in*** S'*, denoted by* $S \sqsubseteq^f S'$*, if for all* $p \in \mathcal{R}$ *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, for all* $u'_1, \ldots, u'_k \in U'$*,*

$$\iota'(p)(u'_1, \ldots, u'_k) = \bigsqcup_{u_1, \ldots, u_k \in U, s.t. f(u_i) = u'_i, 1 \le i \le k} \iota(p)(u_1, \ldots, u_k)$$

then S' *is the* ***tight embedding of*** S ***with respect to*** f*, denoted by* $S' = f(S)$*.*

The relation $\sqsubseteq^{\text{id}}$, abbreviated as $\sqsubseteq$, reflects the tuple-wise information order between structures with the same universe. We have $S \sqsubseteq^f S' \Leftrightarrow f(S) \sqsubseteq S'$.

The Embedding Theorem [70, Thm. 4.9] says that if $S \sqsubseteq^f S'$, then every piece of information extracted from S' via a formula φ is a conservative approximation of the information extracted from S via φ:

Theorem 1. (Embedding Theorem [simplified]). *If* $S = \langle U, \iota \rangle, S' = \langle U', \iota' \rangle \in \mathcal{S}_3[\mathcal{R}]$ *such that* $S \sqsubseteq^f S'$*, then for every formula* φ*,* $[\![\varphi]\!]_3^S \sqsubseteq [\![\varphi]\!]_3^{S'}$*.*

However, embedding alone is not enough. The universe U of 2-valued structure $S = \langle U, \iota \rangle \in \mathcal{S}_2[\mathcal{R}]$ is of *a priori* unbounded size; consequently, we need a method that maps U to an abstract universe $U^\sharp$ of bounded size. The idea behind canonical abstraction is to choose a subset $\mathcal{A} \subseteq \mathcal{R}_1$ of *abstraction relations*, and to define an equivalence relation $\simeq_{\mathcal{A}^S}$ on U that is parameterized by S itself:

$$u_1 \simeq_{\mathcal{A}^S} u_2 \;\Leftrightarrow\; \forall p \in \mathcal{A} : \iota(p)(u_1) = \iota(p)(u_2).$$

This equivalence relation defines the surjective function $f_{\mathcal{A}}^S \colon U \to (U/\simeq_{\mathcal{A}^S})$, which maps an individual to its equivalence class. We have the Galois connection

$$\wp(\mathcal{S}_2[\mathcal{R}]) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \wp(\mathcal{S}_3[\mathcal{R}])$$
$$\alpha(X) = \{f_{\mathcal{A}}^S(S) \mid S \in X\} \qquad \gamma(Y) = \{S \mid S^\sharp \in Y \wedge S \sqsubseteq^f S^\sharp\},$$

where $f_{\mathcal{A}}^S$ in the definition of α denotes the tight-embedding function for logical structures induced by the node-embedding function $f_{\mathcal{A}}^S \colon U \to (U/\simeq_{\mathcal{A}^S})$. The abstraction function α is referred to as *canonical abstraction*. Note that there is an upper bound on the size of each structure $\langle U^\sharp, \iota^\sharp \rangle \in \mathcal{S}_3[\mathcal{R}]$ that is in the image of α: $|U^\sharp| \le 2^{|\mathcal{A}|}$—and thus the power-set of the image of α is a finite sublattice of $\wp(\mathcal{S}_3[\mathcal{R}])$. The ordering on $\wp(\mathcal{S}_3[\mathcal{R}])$ is the Hoare ordering: $SS_1 \sqsubseteq SS_2$ if for all $S_1 \in SS_1$ there exists $S_2 \in SS_2$ such that $S_1 \sqsubseteq^f S_2$.

Maintaining Instrumentation Relations. The technique used to create abstract transformers for canonical-abstraction domains works as follows. The post-state structures for statement *st* are determined using four primitives: (i) partial concretization (or partial model enumeration) via the *focus* operation [70, §6.3]; (ii) formula evaluation, using (a) for a core relation $c \in \mathcal{C}$, the relation-update formula $\tau_{c,st}$ from the concrete semantics, evaluated in 3-valued logic: $[\![\tau_{c,st}]\!]_3$, and (b) for an instrumentation relation $p \in \mathcal{I}$, a finite-differencing-based relation-maintenance formula $\mu_{p,st}$ created by the technique described below [63, §5 & §6]; (iii) lightweight logical reasoning via the *coerce* operation [70, §6.4], which repeatedly performs semantic-reduction steps [19] on the post-state structure to increase the precision of the result; and (iv) a final application of canonical abstraction with respect to abstraction relations $\mathcal{A}$. Due to space limitations, we will only discuss step (ii).[4] Step (ii) transforms a 3-valued pre-state structure $S_1^\#$ that arises just before step (ii), into post-state structure $S_2^\#$ just after step (ii). The structure that consists of just the core relations of $S_2^\#$ is called a *proto-structure*, denoted by $S_{proto}^\#$. The creation of core relation c in $S_{proto}^\#$ from $S_1^\#$ can be expressed as follows:

$$\text{for each } u_1, \ldots, u_k \in U^{S_1^\#},\ \iota^{S_{proto}^\#}(c)(u_1, \ldots, u_k) := [\![\tau_{c,st}(u_1, \ldots, u_k)]\!]_3^{S_1^\#} \quad (2)$$

We now come to the crux of the matter: Suppose that instrumentation relation p is defined by formula ψ_p; how should the analysis engine obtain the value of relation p in $S_2^\#$? From the standpoint of the concrete semantics, p is just cached information that could always be recomputed by reevaluating the defining formula ψ_p, and thus the Embedding Theorem tells us that it is sound to perform

$$\text{for each } u_1, \ldots, u_k \in U^{S_{proto}^\#},\ \iota^{S_2^\#}(p)(u_1, \ldots, u_k) := [\![\psi_p(u_1, \ldots, u_k)]\!]_3^{S_{proto}^\#}. \quad (3)$$

In practice, however, this approach loses too much precision.

An alternative approach is to create a relation-maintenance formula for p with respect to *st* via a weakest-liberal-precondition (WLP) transformation,

$$\mu_{p,st} \stackrel{\text{def}}{=} \psi_p[c \hookleftarrow \tau_{c,st} \mid c \in \mathcal{C}], \quad (4)$$

where $\varphi[q \hookleftarrow \theta]$ denotes the formula obtained from φ by replacing each occurrence of relation symbol q by formula θ. Formula $\mu_{p,st}$ is evaluated in $S_1^\#$:

$$\text{for each } u_1, \ldots, u_k \in U^{S_1^\#},\ \iota^{S_2^\#}(p)(u_1, \ldots, u_k) := [\![\mu_{p,st}(u_1, \ldots, u_k)]\!]_3^{S_1^\#}. \quad (5)$$

However, Eqns. (3) and (5) turn out to be equivalent—and hence equivalently imprecise—because the steps of creating $S_{proto}^\#$ and evaluating $[\![\psi_p]\!]_3^{S_{proto}^\#}$ *mimic exactly* those of evaluating $[\![\psi_p[c \hookleftarrow \tau_{c,st} \mid c \in \mathcal{C}]]\!]_3^{S_1^\#}$.

Relation Maintenance via Finite Differencing. The algorithm for creating a relation-maintenance formula $\mu_{p,st}$, for $p \in \mathcal{I}$, uses an incremental-computation

[4] It is interesting to note that the roles of steps (i), (iii), and (iv) are close to the steps of splitting, propagation, and join, respectively, in our generalization of Stålmarck's algorithm to perform symbolic abstraction [82]. See §5.

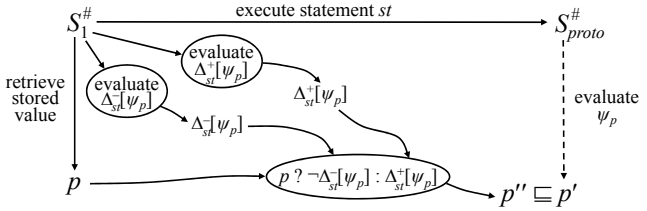

Fig. 2: How to maintain the value of ψ_p in 3-valued logic in response to changes in the values of core relations caused by the execution of structure transformer st.

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

Fig. 3: Finite-difference formulas for first-order formulas.

strategy: $\mu_{p,st}$ is defined in terms of the stored (pre-state) value of p, along with two finite-differencing operators, denoted by $\Delta_{st}^-[\cdot]$ and $\Delta_{st}^+[\cdot]$.

$$\mu_{p,st} \stackrel{\text{def}}{=} p\,?\,\neg\Delta_{st}^-[\psi_p] : \Delta_{st}^+[\psi_p]. \tag{6}$$

In this approach to the relation-maintenance problem, the two finite-differencing operators characterize the tuples of relation p that are *subtracted* and *added* in response to structure transformation st. $\Delta_{st}^-[\cdot]$ has value 1 for tuples that st changes from 1 to 0; $\Delta_{st}^+[\cdot]$ has value 1 for tuples that st changes from 0 to 1. Eqn. (6) means that if the old value of a p tuple is 1, then its new value is 1 unless there is a negative change; if the old value of a p tuple is 0, then its new value is 0 unless there is a positive change. Fig. 2 depicts how the static-analysis engine evaluates $\Delta_{st}^-[\psi_p]$ and $\Delta_{st}^+[\psi_p]$ in $S_1^\#$ and combines these values with the value of the p tuple from $S_1^\#$ to obtain the value of the p'' tuple.

The operators $\Delta^{-}_{st}[\cdot]$ and $\Delta^{+}_{st}[\cdot]$ are defined recursively, as shown in Fig. 3. The definitions in Fig. 3 make use of the operator $\mathbf{F}_{st}[\varphi]$ (standing for "Future"), defined as follows:

$$\mathbf{F}_{st}[\varphi] \stackrel{\text{def}}{=} \varphi \,?\, \neg\Delta^{-}_{st}[\varphi] : \Delta^{+}_{st}[\varphi]. \tag{7}$$

Thus, maintenance formula $\mu_{p,st}$ can also be expressed as $\mu_{p,st} \stackrel{\text{def}}{=} \mathbf{F}_{st}[p]$. Eqn. (7) and Fig. 3 define a syntax-directed translation scheme that can be implemented via a recursive walk over a formula φ. The operators $\Delta^{-}_{st}[\cdot]$ and $\Delta^{+}_{st}[\cdot]$ are mutually recursive. For instance, $\Delta^{+}_{st}[\neg\varphi_1] = \Delta^{-}_{st}[\varphi_1]$ and $\Delta^{-}_{st}[\neg\varphi_1] = \Delta^{+}_{st}[\varphi_1]$. Moreover, each occurrence of $\mathbf{F}_{st}[\varphi_i]$ contains additional occurrences of $\Delta^{-}_{st}[\varphi_i]$ and $\Delta^{+}_{st}[\varphi_i]$.

Note how $\Delta^{-}_{st}[\cdot]$ and $\Delta^{+}_{st}[\cdot]$ for $\varphi_1 \vee \varphi_2$ and $\varphi_1 \wedge \varphi_2$ resemble the product rule of differentiation. Continuing the analogy, it helps to bear in mind that the "independent variables" are the core relations, whose values are changed via the $\tau_{c,st}$ formulas; the "dependent variable" is the relation defined by formula φ.

The relation-maintenance formula defined in Eqn. (6) is, in essence, a non-standard approach to WLP based on *finite differencing*, rather than *substitution*. To see the relationship with WLP, consider the substitution-based relation-maintenance formula $\psi_p[c \leftarrow \tau_{c,st} \mid c \in \mathcal{C}]$ defined in Eqn. (4), which computes the WLP of post-state instrumentation relation p with respect to statement *st*. In the concrete semantics, this formula is equivalent to the finite-differencing-based relation-maintenance formula, $\mathbf{F}_{st}[p] = p \,?\, \neg\Delta^{-}_{st}[p] : \Delta^{+}_{st}[p]$ [63, Thm. 5.3]. In effect, $\mathbf{F}_{st}[p]$ is a "footprint-based" version of WLP.

Answers to The Four Questions.

Q1. The concrete semantics is specified by (i) declaring a suitable set of core relations $\mathcal{C}$ that define a system's concrete states, and (ii) writing—using first-order logic plus transitive closure over $\mathcal{C}$—the $\tau_{c,st}$ formulas that define the concrete transformers.

Q2. A canonical-abstraction domain is specified by (i) defining instrumentation relations $\mathcal{I}$ (again, using first-order logic plus transitive closure), and (ii) selecting which unary relations in $\mathcal{C}_1 \uplus \mathcal{I}_1$ to use as abstraction relations $\mathcal{A}$. $\mathcal{I}$ controls what information is maintained (in addition to the core relations); $\mathcal{A}$ controls what individuals are indistinguishable. The two mechanisms are connected because one can declare unary instrumentation relations to be abstraction relations.

Q3.

(a) Abstract transformers are constructed automatically by means of the four-part construction sketched in the section "Maintaining Instrumentation Relations" above. In particular, an instrumentation relation $p \in \mathcal{I}$ is evaluated using the relation-maintenance formula $\mu_{p,st}$, created by applying a finite-differencing transformation to p's defining formula ψ_p (Eqn. (6)).

(b) Representations of abstract transformers can be created by means of a principle of "pairing and then abstracting" [35, §6]. In particular, one uses (sets of) logical structures over a duplicated vocabulary $\mathcal{R} \uplus \mathcal{R}'$ to represent relations between logical structures over vocabulary $\mathcal{R}$. The relation-composition operation needed for interprocedural analysis [74], can be performed in the usual way, i.e., $R_3[\mathcal{R} \uplus \mathcal{R}''] = \exists\mathcal{R}' : R_1[\mathcal{R} \uplus \mathcal{R}'] \wedge R_2[\mathcal{R}' \uplus \mathcal{R}'']$,

using three vocabularies of relation symbols, a meet operation on 3-valued structures [4], and implementing $\exists\mathcal{R}'$ by dropping all $\mathcal{R}'$ relations [35, §6.5].

Q4. For statement st, the relation-maintenance formula $\mu_{p,st}$ for instrumentation relation p is $p?\neg\Delta^{-}_{st}[\psi_p] : \Delta^{+}_{st}[\psi_p]$ (evaluated in the pre-state structure), rather than ψ_p (evaluated in the post-state structure) or $\psi_p[c \hookleftarrow \tau_{c,st} \mid c \in \mathcal{C}]$ (evaluated in the pre-state structure). Finite-differencing addresses the non-compositionality issue because $\mu_{p,st}$ identifies the "footprint" of statement st on p, which reduces the number of tuples in p that have to be reevaluated.

4 TSL: Transformer Specification Language

In 2008, Lim and Reps created the TSL system [45], a meta-tool to help in the creation of tools for analyzing machine code. From a single specification of the concrete semantics of a machine-code instruction set, TSL automatically generates correct-by-construction implementations of the state-transformation functions needed in state-space-exploration tools that use static analysis, dynamic analysis, symbolic analysis, or any combination of the three [45, 44, 80].

The TSL meta-language is a strongly typed, first-order functional language with a datatype-definition mechanism for defining recursive datatypes, plus deconstruction by means of pattern matching. Writing a TSL specification for an instruction set is similar to writing an interpreter in first-order ML: the specification of an instruction set's concrete semantics is written as a TSL function

```
state interpInstr(instruction I, state S) ...;
```

where `instruction` and `state` are user-defined datatypes that represent the instructions and the semantic states, respectively. TSL's meta-language provides a fixed set of basetypes; a fixed set of arithmetic, bitwise, relational, and logical operators; and a facility for defining map-types.

TSL's most basic mechanism for creating abstract transformers is similar to the syntax-directed-replacement method described in §2.2. From the specification of interpInstr for a given instruction set, the TSL compiler creates a C++ template that serves as a common intermediate representation (CIR). The CIR template is parameterized on an abstract-domain class, $\mathbb{A}$, and a fixed set of $\mathbb{A}$ primitive operations that mainly correspond to the primitive operations of the TSL meta-language. A C++ class that can be used to instantiate the CIR is called a *semantic reinterpretation* [56–58, 46]; it must implement an interface that consists of 42 basetype operators, most of which have four variants, for 8-, 16-, 32-, and 64-bit integers, as well as 12 map access/update operations and a few additional operations, such as join, meet, and widen.

The CIR can be used to create multiple abstract interpreters for a given instruction set. Each analyzer is specified by supplying a semantic reinterpretation (for the TSL primitives), which—by extension to TSL expressions and user-defined functions—provides the reinterpretation of the function interpInstr, which is essentially the desired function $I_{s,\mathbb{A}}(\cdot)$ discussed in §2.3. Each reinterpretation instantiates the same CIR template, which in turn comes directly from the specification of the instruction set's concrete semantics. By this means, the abstract transformers generated for different abstract domains are guaranteed

to be mutually consistent (and also to be consistent with an instruction-set emulator that is generated from the same specification of the concrete semantics).

Although the syntax-directed-replacement method has its drawbacks, it works well for machine-code instruction sets. Using a corpus of 19,066 Intel x86 instructions, Lim and Reps found, for one abstract domain, that 96.8% of the transformers created via semantic reinterpretation reached the limit of precision attainable with that abstract domain [45, §5.4.1]. Evidently, the semantic specifications of x86 instructions do not usually suffer from the kinds of missed-correlation effects discussed in §2.2.

Answers to The Four Questions.

Q1. The semantics of machine-code instructions are specified by writing an interpreter in the TSL meta-language.

Q2. To define an abstract domain and its operations, one needs to supply a C++ class that implements a semantic reinterpretation.

Q3.

(a) The common intermediate representation (CIR) generated for a given TSL instruction-set specification is a C++ template that can be instantiated with multiple semantic-reinterpretation classes to create multiple reinterpretations of the function interpInstr.

(b) Representations of abstract transformers can be created via the approach discussed below in the section "Relational Abstract Domains."

Q4. One predefined reinterpretation is for quantifier-free formulas over the theory of bitvectors and bitvector arrays (QF_ABV). One can avoid the myopia of operator-by-operator reinterpretation illustrated in §2.2 by using the QF_ABV reinterpretation on basic blocks and loop-free fragments. The formula so obtained has a "long-range view" of the fragment's semantics. One can then employ the symbolic-abstraction techniques described in §5.

Relational Abstract Domains. An interesting problem that we encountered with TSL was how to perform reinterpretation for relational abstract domains, such as polyhedra [21], weakly relational domains [49], and affine equalities [55, 40, 27]. With such domains, the goal is to create a *representation* of an abstract transformer that over-approximates the concrete transformer for an instruction or basic block. Clearly state should be redefined as a relational-abstract-domain class whose values represent a relation between input states and output states; however, it was not immediately obvious how the TSL basetypes should be redefined, nor how operations such as Plus32, And32, Xor32, etc. should be handled.

The literature on relational numeric abstract domains did not provide much assistance. Most papers on such domains focus on some modeling language—typically affine programs ([21, §4], [55, §2], [49, §4])—involving only assignments and tests written in some restricted form—and describe how to create abstract transformers only for concrete transformers written in that form. For instance, for an assignment statement "x := e"

– If e is a linear expression, the coefficients for the variables in e are used to create an abstract-domain value that encodes a linear transformation.

– If e is a non-linear expression, it is modeled as "x := ?" or, equivalently, "havoc(x)." (That is, after "x := e" executes, x can hold any value.)

In contrast, with TSL each abstract-domain value must be constructed by evaluating an expression in the TSL meta-language. Moreover, the concrete semantics of an instruction set often makes use of non-linear operators, such as bitwise-and and bitwise-or. There could be an unacceptable loss of precision if *every* use of a non-linear operator in an instruction's semantic definition caused a havoc. Fortunately, we were able to devise a generic method for creating abstract transformers, usable with multiple relational abstract domains, that can retain some degree of precision for some occurrences of non-linear operators [27, §6.6.4].

For relational abstract domains, the usually straightforward syntax-directed-replacement method is somewhat subtle. For a set of variables V, a value in type Rel[V] denotes a set of *assignments* $V \to \mathrm{Val}$ (for some value space Val). When V and V' are disjoint sets of variables, the type Rel[$V; V'$] denotes the set of Rel values over variables $V \uplus V'$. We extend this notation to cover singletons: if i is a single variable not in V, then the type Rel[$V; i$] denotes the set of Rel values over the variables $V \uplus \{i\}$. (Operations sometimes introduce additional temporary variables, in which case we have types like Rel[$V; i, i'$] and Rel[$V; i, i', i''$].)

In a reinterpretation that yields abstractions of concrete transition-relations, the type state represents a relation on pre-states to post-states. For example, suppose that the goal is to track relationships among the values of the processor's registers. The abstraction of state would be Rel[$R; R'$], where R is the set of register names (e.g., for Intel x86, $R \stackrel{\text{def}}{=} \{\texttt{eax}, \texttt{ebx}, \dots\}$), and R' is the same set of names, distinguished by primes ($R' \stackrel{\text{def}}{=} \{\texttt{eax'}, \texttt{ebx'}, \dots\}$).

In contrast, the abstraction of a machine-integer type, such as INT32, becomes a relation on pre-states to machine integers. Thus, for machine-integer types, we introduce a fresh variable i to hold the "current value" of a reinterpreted machine integer. Because R still refers to the pre-state registers, we write the type of a Rel-reinterpreted machine integer as Rel[$R; i$]. Although technically we are working with relations, for a Rel[$R; i$] value it is often useful to think of R as a set of *independent variables* and i as the *dependent variable*.

Constants. The Rel reinterpretation of a constant c is the Rel[$V; i$] value that encodes the constraint $i = c$.

Variable-Access Expressions. The Rel reinterpretation of a variable-access expression access(S, v), where S's value is a Rel state-transformer of type Rel[$V; V'$] and $v \in V$, is the Rel[$V; i$] value obtained as follows:

1. Extend S to be a Rel[$V; V'; i$] value, leaving i unconstrained.
2. Assume the constraint $i = v'$ on the extended S value (to retrieve v from the "current state").
3. Project away V', leaving a Rel[$V; i$] value that holds in i constraints on v's value in terms of the pre-state vocabulary V.

Update Operations. Suppose that $S \in \mathrm{Rel}[V; V']$, and the reinterpretation of expression e with respect to S has produced the reinterpreted value $J \in \mathrm{Rel}[V; i]$. We want to create $S'' \in \mathrm{Rel}[V; V']$ that acts like S, except that post-state

variable $v' \in V'$ satisfies the constraints on i in $J \in \mathrm{Rel}[V; i]$. The operation $\mathrm{update}(S, v, J)$ is carried out as follows:

1. Let S' be the result of havocking v' from S.
2. Let K be the result of starting with J, renaming i to v', and then extending it to be a $\mathrm{Rel}[V; V']$ value by adding unconstrained variables in the set $V' - \{v'\}$.
3. Return $S'' \stackrel{\text{def}}{=} S' \sqcap K$.

S' captures the state in which we "forget" the previous value of v', and K asserts that v' satisfies the constraints (in terms of the pre-state vocabulary V) that were obtained from evaluating e.

Addition. Suppose that we have two $\mathrm{Rel}[V; i]$ values x and y, and wish to compute the $\mathrm{Rel}[V; i]$ value for the expression $x + y$. We proceed as follows:

1. Rename y's i variable to i'; this makes y a $\mathrm{Rel}[V; i']$ value.
2. Extend both x and y to be $\mathrm{Rel}[V; i, i', i'']$ values, leaving i' and i'' unconstrained in x, and i and i'' unconstrained in y.
3. Compute $x \sqcap y$.
4. Assume the constraint $i'' = i + i'$ on the value computed in step (3).
5. Project away i and i', leaving a $\mathrm{Rel}[V; i'']$ value.
6. In the value computed in step (5), rename i'' to i, yielding a $\mathrm{Rel}[V; i]$ value.

5 Symbolic Abstraction

Since 2002, the first author has been interested in connections between abstract interpretation and logic—in particular, how to harness decision procedures to obtain algorithms for several fundamental primitives used in abstract interpretation [64, 85, 82, 78, 79]. The work aims to bridge the gap between (i) the use of logic for specifying program semantics and performing program analysis, and (ii) abstract interpretation. In 1997, Graf and Saïdi [31] showed how to use theorem provers to generate best abstract transformers for predicate-abstraction domains (fixed, finite collections of Boolean predicates). In 2004, Reps et al. [64] gave a method that makes such a connection for a much broader class of abstract domains. That paper also introduced the following problem, which we (now) call *symbolic abstraction*:

> Given formula φ in logic $\mathcal{L}$, and abstract domain $\mathbb{A}$, find the most-precise descriptor $a^\sharp$ in $\mathbb{A}$ that over-approximates the meaning of φ (i.e., $[\![\varphi]\!] \subseteq \gamma(a^\sharp)$).

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

The connection between logic and abstract interpretation becomes clearer if we view an abstract domain $\mathbb{A}$ as a logic fragment $\mathcal{L}_{\mathbb{A}}$ of some general-purpose logic $\mathcal{L}$, and each abstract value as a formula in $\mathcal{L}_{\mathbb{A}}$. We say that $\widehat{\gamma}$ is a *symbolic-concretization operation* for $\mathbb{A}$ if it maps each $a^\sharp \in \mathbb{A}$ to $\varphi_{a^\sharp} \in \mathcal{L}_{\mathbb{A}}$ such that the meaning of $\varphi_{a^\sharp}$ equals the concretization of $a^\sharp$; i.e., $[\![\varphi_{a^\sharp}]\!] = \gamma(a^\sharp)$. $\mathcal{L}_{\mathbb{A}}$ is often defined by a syntactic restriction on the formulas of $\mathcal{L}$.

Example 1. If $\mathbb{A}$ is the set of environments over intervals, $\mathcal{L}_{\mathbb{A}}$ is the set of conjunctions of one-variable inequalities over the program variables. It is generally

easy to implement $\widehat{\gamma}$ for an abstract domain. For example, given $a^\sharp \in \mathbb{A}$, it is straightforward to read off the appropriate $\varphi_{a^\sharp} \in \mathcal{L}_{\mathbb{A}}$: each entry $x \mapsto [c_{low}, c_{high}]$ contributes the conjuncts "$c_{low} \leq x$" and "$x \leq c_{high}$." □

Thus, symbolic abstraction addresses a fundamental approximation problem:

Given formula $\varphi \in \mathcal{L}$, find the strongest consequence of φ that is expressible in a different logic $\mathcal{L}'$.

Since 2011, we (Thakur and Reps) pursued several new insights on this question. One insight was that generalized versions of an old, and not widely used, method for validity checking of propositional-logic formulas, called Stålmarck's method, provide new ways to implement $\widehat{\alpha}$. The methods that we subsequently developed [82, 78, 81, 79] offer much promise for building more powerful program-analysis tools. They (i) allow more precise implementations of abstract-interpretation primitives to be created—including ones that attain the fundamental limits on precision that abstract-interpretation theory establishes—and (ii) drastically reduce the time needed to implement such primitives while ensuring correctness by construction. In [79], we described a method that, for a certain class of abstract domains, uses $\widehat{\alpha}$ to solve the following problem:

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

5.1 Abstract Transformers via Symbolic Abstraction

We now illustrate how $\widehat{\alpha}$ can be used both to apply an abstract transformer and to construct a representation of an abstract transformer.

Example 2. 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 the logic QF_ABV as the formula φ_τ:

$$\varphi_\tau \stackrel{\text{def}}{=} \mathtt{ebx'} = \begin{pmatrix} (\mathtt{ebx}\ \&\ \mathtt{0xFFFF00FF}) \\ |\ ((\mathtt{ebx} + 256 * (\mathtt{eax}\ \&\ \mathtt{0xFF}))\ \&\ \mathtt{0xFF00}) \end{pmatrix} \wedge \mathtt{eax'} = \mathtt{eax} \\ \wedge\ \mathtt{ecx'} = \mathtt{ecx}, \tag{8}$$

where "&" and "|" denote the non-linear bit-masking operations bitwise-and and bitwise-or, respectively.

Suppose that the abstract domain is $\mathcal{E}_{2^{32}}$, the domain of affine equalities over the 32-bit registers `eax`, `ebx`, and `ecx`, and that we would like to apply the abstract transformer for τ when the input abstract value in $\mathcal{E}_{2^{32}}$ is $\mathtt{ebx} = \mathtt{ecx}$. This task corresponds to finding the strongest consequence of the formula $\psi \equiv (\mathtt{ebx} = \mathtt{ecx} \wedge \varphi_\tau)$ that can be expressed as an affine relation among $\mathtt{eax'}$, $\mathtt{ebx'}$, and $\mathtt{ecx'}$, which turns out to be $\widehat{\alpha}(\psi) \equiv (2^{16}\mathtt{ebx'} = 2^{16}\mathtt{ecx'} + 2^{24}\mathtt{eax'}) \wedge (2^{24}\mathtt{ebx'} = 2^{24}\mathtt{ecx'})$. Multiplying by a power of 2 shifts bits to the left; because we are using arithmetic mod 2^{32}, bits shifted off the left end are unconstrained. Thus,

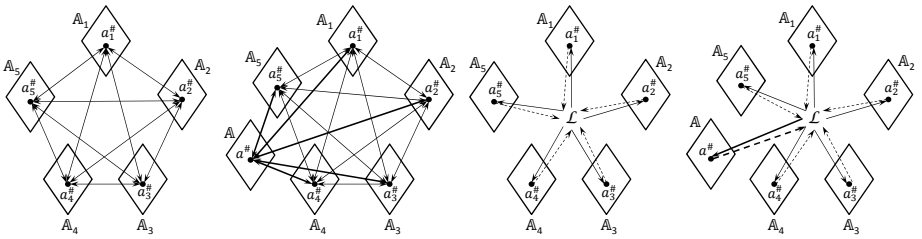

Fig. 4: Conversion between abstract domains with the clique approach ((a) and (b)) versus the symbolic-abstraction approach ((c) and (d)).

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 $\texttt{eax}'$. This example illustrates that the result of applying an abstract transformer can be non-obvious—even for a single machine-code instruction—which serves to motivate the desire for automation.

Now suppose that we would like to compute a representation of the best abstract transformer for τ in abstract domain $\mathcal{E}_{2^{32}}$. This task corresponds to finding the strongest consequence of φ_τ that can be expressed as an affine relation among $\texttt{eax}$, $\texttt{ebx}$, $\texttt{ecx}$, $\texttt{eax}'$, $\texttt{ebx}'$, and $\texttt{ecx}'$, which turns out to be $\widehat{\alpha}(\varphi_\tau) \equiv (2^{16}\texttt{ebx}' = 2^{16}\texttt{ebx} + 2^{24}\texttt{eax}) \wedge (\texttt{eax}' = \texttt{eax}) \wedge (\texttt{ecx}' = \texttt{ecx})$. □

5.2 Communication of Information Between Abstract Domains

We now show how symbolic abstraction 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).

Fig. 4(a) and Fig. 4(b) show what happens if we want to communicate information between abstract domains *without* symbolic abstraction. Because it is necessary to create explicit conversion routines for each pair of abstract domains, we call this approach the "clique approach." As shown in Fig. 4(b), when a new abstract domain $\mathbb{A}$ is introduced, the clique approach requires that a conversion method be developed for each prior domain $\mathbb{A}_i$. In contrast, as shown in Fig. 4(d), the symbolic-abstraction approach only requires that we have $\widehat{\alpha}$ and $\widehat{\gamma}$ methods that relate $\mathbb{A}$ and $\mathcal{L}$.

If each analysis i is sound, each result $a_i^\sharp$ represents an over-approximation of the actual set of concrete states. Consequently, the collection of analysis results $\{a_i^\sharp\}$ implicitly tells us that only the states in $\bigcap_i \gamma(a_i^\sharp)$ 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 [19] of each of the $a_i^\sharp$ with respect to the set of abstract values $\{a_j^\sharp \mid i \neq j\}$. 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 Fig. 5 for the case of two abstract domains, $\mathcal{P} = \mathit{Env}[\mathit{Parity}]$ and $\mathcal{I} = \mathit{Env}[\mathit{Interval}]$. Given $a_1^\sharp \in \mathcal{P}$ and $a_2^\sharp \in \mathcal{I}$, we

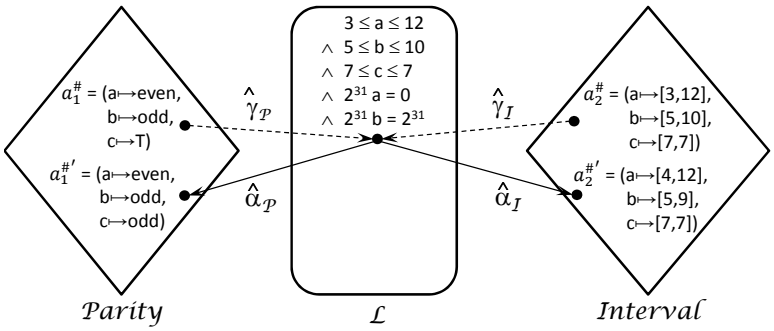

Fig. 5: Improving values from two abstract domains via symbolic abstraction.

can improve the pair $\langle a_1^\sharp, a_2^\sharp \rangle$ by first creating the formula $\varphi \stackrel{\text{def}}{=} \widehat{\gamma}_{\mathcal{P}}(a_1^\sharp) \wedge \widehat{\gamma}_{\mathcal{I}}(a_2^\sharp)$, and then applying $\widehat{\alpha}_{\mathcal{P}}$ and $\widehat{\alpha}_{\mathcal{I}}$ to φ to obtain $a_1^{\sharp\prime} = \widehat{\alpha}_{\mathcal{P}}(\varphi)$ and $a_2^{\sharp\prime} = \widehat{\alpha}_{\mathcal{I}}(\varphi)$, respectively. $a_1^{\sharp\prime}$ and $a_2^{\sharp\prime}$ can be smaller than the original values $a_1^\sharp$ and $a_2^\sharp$, respectively. We then use the pair $\langle a_1^{\sharp\prime}, a_2^{\sharp\prime} \rangle$ instead of $\langle a_1^\sharp, a_2^\sharp \rangle$. Fig. 5 shows a specific example of how this approach to semantic reduction improves both the *Env*[*Parity*] value and the *Env*[*Interval*] value. When there are more than two abstract domains, we form the conjunction $\varphi \stackrel{\text{def}}{=} \bigwedge_i \widehat{\gamma}_i(a_i^\sharp)$, and then apply each $\widehat{\alpha}_i$ to obtain $a_i^{\sharp\prime} = \widehat{\alpha}_i(\varphi)$.

5.3 Algorithms for Symbolic Abstraction

The various algorithms for computing symbolic abstraction can be seen as relying on the following two properties:

Theorem 2. *[76, Thm. 3.14]* $\widehat{\alpha}(\varphi) = \bigsqcup \{ \beta(S) \mid S \models \varphi \}$ □

Theorem 3. *[76, Thm. 3.15]* $\widehat{\alpha}(\varphi) = \bigsqcap \{ a \mid \varphi \Rightarrow \widehat{\gamma}(a) \}$ □

The *representation function* β returns the abstraction of a singleton concrete state; i.e., $\beta(\sigma) = \alpha(\{\sigma\})$.

RSY Algorithm. Reps et al. [64] presented a framework for computing $\widehat{\alpha}$—which we call the RSY algorithm—that applies to any logic $\mathcal{L}$ and abstract domain $\mathbb{A}$ that satisfy certain conditions. The key insight of the algorithm is the use of an SMT solver for $\mathcal{L}$ as a black-box to query for models of φ and then make use of Thm. 2. Unfortunately, Thm. 2 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 queries the SMT solver to compute a *finite* sequence $\sigma_1, \sigma_2, \ldots, \sigma_k$ of models of φ. This sequence is used to compute the sequence of abstract values $a_0^\sharp, a_1^\sharp, a_2^\sharp, \ldots, a_k^\sharp$ as follows:

$$\begin{aligned} a_0^\sharp &= \bot \\ a_i^\sharp &= a_{i-1}^\sharp \sqcup \beta(\sigma_i), \quad \sigma_i \models \varphi, \quad 1 \leq i \leq k \end{aligned} \tag{9}$$

Merely sampling k arbitrary models of φ would not work. In particular, it is possible that $a_{i-1}^\sharp = a_i^\sharp$, in which case step i has not made progress. To ensure

progress, we require σ_i to be a model of φ such that $\sigma_i \notin \gamma(a^\sharp_{i-1})$. In other words, σ_i should be a model that satisfies $\varphi \wedge \neg\widehat{\gamma}(a^\sharp_{i-1})$. Eqn. (9) can be restated as

$$\begin{array}{l} a^\sharp_0 = \bot \\ a^\sharp_i = a^\sharp_{i-1} \sqcup \beta(\sigma_i), \quad \sigma_i \models \varphi \wedge \neg\widehat{\gamma}(a^\sharp_{i-1}), \quad 1 \leq i \end{array} \tag{10}$$

Obtaining σ_i as a model of $\varphi \wedge \neg\widehat{\gamma}(a^\sharp_{i-1})$ ensures that either $a^\sharp_{i-1} \sqsubset a^\sharp_i$ or else $a^\sharp_{i-1} = a^\sharp_i = \widehat{\alpha}(\varphi)$. Thus, if $\mathbb{A}$ has no infinite ascending chains, the sequence constructed by Eqn. (10) forms a finite ascending chain that converges to $\widehat{\alpha}(\varphi)$:

$$\bot = a^\sharp_0 \sqsubset a^\sharp_1 \sqsubset a^\sharp_2 \sqsubset \ldots \sqsubset a^\sharp_{k-1} \sqsubset a^\sharp_k = \widehat{\alpha}(\varphi). \tag{11}$$

From Eqn. (10), we can identify the requirements on $\mathcal{L}$ and $\mathbb{A}$:

1. There is a Galois connection $\mathbb{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{A}$ between $\mathbb{A}$ and concrete domain $\mathbb{C}$, and an implementation of the corresponding representation function β.
2. There is an algorithm to evaluate $a^\sharp \sqcup \beta(\sigma)$ for all $a^\sharp \in \mathbb{A}$.
3. There is a symbolic-concretization operation $\widehat{\gamma}$ that maps an abstract value $a^\sharp \in \mathbb{A}$ to a formula $\widehat{\gamma}(a^\sharp)$ in $\mathcal{L}$.
4. $\mathbb{A}$ has no infinite ascending chains.
5. There is a decision procedure for logic $\mathcal{L}$ that is also capable of returning a model satisfying a given formula in $\mathcal{L}$.
6. Logic $\mathcal{L}$ is closed under conjunction and negation.

Pseudo-code for the RSY algorithm can be found in [64].

Bilateral Algorithm. The bilateral algorithm [78] is a framework for computing $\widehat{\alpha}$ that is similar to the RSY algorithm in that it queries an SMT solver. However, the nature of the queries differ in the two algorithms. Furthermore, the bilateral algorithm makes use of both Thm. 2 and Thm. 3. While the RSY algorithm converges to the final answer by moving up the lattice, the bilateral algorithm converges to the final answer by both moving up the lattice starting from $\bot$ and moving down the lattice starting from $\top$. That is, the bilateral algorithm computes a finite sequence of pairs of abstract values $(l^\sharp_i, u^\sharp_i)$ such that

$$\bot = l^\sharp_0 \sqsubseteq l^\sharp_1 \sqsubseteq \ldots \sqsubseteq l^\sharp_k = \widehat{\alpha}(\varphi) = u^\sharp_k \sqsubseteq \ldots \sqsubseteq u^\sharp_1 \sqsubseteq u^\sharp_0 = \top. \tag{12}$$

The progress guarantee for the RSY algorithm is that $a^\sharp_i \sqsubset a^\sharp_{i+1}$: on each iteration, the algorithm moves up the lattice. The progress guarantee for the bilateral algorithm is slightly different: on each iteration, the algorithm either moves up the lattice or moves down the lattice: *either* $l^\sharp_i \sqsubset l^\sharp_{i+1}$ *or* $u^\sharp_{i+1} \sqsubset u^\sharp_i$.

A key concept in the bilateral algorithm is the notion of an *abstract-consequence* operation:

Definition 3. *An operation $AC(\cdot,\cdot)$ is an* **acceptable abstract-consequence operation** *iff for all $l^\sharp, u^\sharp \in \mathbb{A}$ such that $l^\sharp \sqsubset u^\sharp$, $a^\sharp = AC(l^\sharp, u^\sharp)$ implies that $l^\sharp \sqsubseteq a^\sharp$ and $a^\sharp \not\sqsupseteq u^\sharp$.* □

In particular, $\gamma(a^\sharp)$ does not encompass $\gamma(u^\sharp)$, and whenever $a^\sharp \neq \bot$, $\gamma(a^\sharp)$ overlaps $\gamma(u^\sharp)$.

Readers familiar with the concept of interpolation [23] might see similarities between interpolation and abstract consequence. However, as discussed in [78, §3], there are significant differences between these two notions.

The sequence $(l_i^\sharp, u_i^\sharp)$ is computed using the following rules:

$$(l_0^\sharp, u_0^\sharp) = (\bot, \top) \tag{13}$$

$$(l_i^\sharp, u_i^\sharp) = (l_{i-1}^\sharp, u_{i-1}^\sharp \sqcap \mathrm{AC}(l_{i-1}^\sharp, u_{i-1}^\sharp),\ \varphi \Rightarrow \widehat{\gamma}\big(\mathrm{AC}(l_{i-1}^\sharp, u_{i-1}^\sharp)\big),\ l_{i-1}^\sharp \sqsubset u_{i-1}^\sharp \tag{14}$$

$$(l_i^\sharp, u_i^\sharp) = (l_{i-1}^\sharp \sqcup \beta(\sigma_i), u_{i-1}^\sharp),\ \sigma_i \models \varphi \wedge \neg\widehat{\gamma}\big(\mathrm{AC}(l_{i-1}^\sharp, u_{i-1}^\sharp)\big),\ l_{i-1}^\sharp \sqsubset u_{i-1}^\sharp \tag{15}$$

The invariant that is maintained is that $l_i^\sharp \sqsubseteq \widehat{\alpha}(\varphi) \sqsubseteq u_i^\sharp$. $l_0^\sharp$ is initialized to $\bot$, and $u_0^\sharp$ is initialized to $\top$. Let $a_{i-1}^\sharp = \mathrm{AC}(l_{i-1}^\sharp, u_{i-1}^\sharp)$. There are two cases: either $\varphi \Rightarrow \widehat{\gamma}(a_{i-1}^\sharp)$ or it does not. If $\varphi \Rightarrow \widehat{\gamma}(a_{i-1}^\sharp)$, then $u_i^\sharp$ can be defined as $u_{i-1}^\sharp \sqcap a_{i-1}^\sharp$, and $l_i^\sharp = l_{i-1}^\sharp$ (Eqn. (14)). This step makes progress because $a_{i-1}^\sharp \not\sqsupseteq u_{i-1}^\sharp$ implies that $u_i^\sharp \sqsubset u_{i-1}^\sharp \sqcap a_{i-1}^\sharp$. Otherwise, there must exist a model σ_i such that $\sigma_i \models \varphi \wedge \neg\widehat{\gamma}(a_{i-1}^\sharp)$. In this case, $l_i^\sharp$ can be defined as $l_{i-1}^\sharp \sqcup \beta(\sigma_i)$ (Eqn. (15)). This step makes progress for reasons similar to the RSY algorithm. Thus, on each iteration either $l_i^\sharp$ or $u_i^\sharp$ is updated. The values $l_i^\sharp$ and $u_i^\sharp$ are guaranteed to converge to $\widehat{\alpha}(\varphi)$ provided $\mathbb{A}$ has neither infinite ascending chains nor infinite descending chains.[5]

There can be multiple ways of defining the abstract-consequence operation. In fact, the bilateral algorithm reduces to the RSY algorithm if we define $\mathrm{AC}(l_{i-1}^\sharp, u_{i-1}^\sharp) \stackrel{\text{def}}{=} l_{i-1}^\sharp$. Other algorithms for computing abstract consequence for a large class of abstract domains are described in [78]. The choice of abstract consequence determines the cost of each query of the SMT solver as well as the rate of convergence of the bilateral algorithm.

The key advantage of the bilateral algorithm over the RSY algorithm is that the bilateral algorithm is an *anytime algorithm*, because the algorithm can return a sound over-approximation ($u_i^\sharp$) of the final answer if it is stopped at any point. This property makes the bilateral algorithm resilient to SMT-solver timeouts.

Pseudo-code for the bilateral algorithm can be found in [78] and [76, Ch. 5].

Generalizations of Stålmarck's Algorithm. In [81], we showed how Stålmarck's method [75], 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[$\mathbb{A}$], that is parameterized on an abstract domain $\mathbb{A}$ and operations on $\mathbb{A}$. The algorithm that goes by the name "Stålmarck's method" is one instantiation of Stålmarck[$\mathbb{A}$] with a certain Boolean abstract domain. At each step, Stålmarck[$\mathbb{A}$] holds some $a^\sharp \in \mathbb{A}$; each of the proof rules employed in Stålmarck's method improves $a^\sharp$ by finding a semantic reduction of $a^\sharp$ with respect to φ.

The abstraction-interpretation-based view enables us to lift Stålmarck's method from propositional logic to richer logics by instantiating Stålmarck[$\mathbb{A}$] with richer abstract domains [82]. Moreover, it brings out a new connection between Stålmarck's method and $\widehat{\alpha}$. To check whether a formula φ is unsatisfiable, Stålmarck[$\mathbb{A}$] computes $\widehat{\alpha}_{\mathbb{A}}(\varphi)$ and performs the test "$\widehat{\alpha}_{\mathbb{A}}(\varphi) = \bot_{\mathbb{A}}$?" If the test succeeds, it establishes that $[\![\varphi]\!] \subseteq \gamma(\bot_{\mathbb{A}}) = \emptyset$, and hence that φ is unsatisfiable.

[5] A slight modification to the bilateral algorithm can remove the requirement of having no infinite descending chains [78].

To explain the Stålmarck[$\mathbb{A}$] algorithm for $\widehat{\alpha}$, we first define the notion of $\widehat{\text{Assume}}$. Given $\varphi \in \mathcal{L}$ and $a^\sharp \in \mathbb{A}$, $\widehat{\text{Assume}}[\varphi](a^\sharp)$ returns the best value in $\mathbb{A}$ that over-approximates the meaning of φ in concrete states described by $a^\sharp$. That is, $\widehat{\text{Assume}}[\varphi](a^\sharp)$ equals $\alpha([\![\varphi]\!] \cap \gamma(a^\sharp))$.

The principles behind the Stålmarck[$\mathbb{A}$] algorithm for $\widehat{\alpha}$ can be understood via the following equations:

$$\widehat{\alpha}(\varphi) = \widehat{\text{Assume}}[\varphi](\top) \tag{16}$$

$$\widehat{\text{Assume}}[\varphi_1 \wedge \varphi_2](a^\sharp) \sqsubseteq \widehat{\text{Assume}}[\varphi_1](a^\sharp) \sqcap \widehat{\text{Assume}}[\varphi_2](a^\sharp) \tag{17}$$

$$\widehat{\text{Assume}}[\varphi](a^\sharp) \sqsubseteq \widehat{\text{Assume}}[\varphi](a^\sharp \sqcap a_1^\sharp) \sqcup \widehat{\text{Assume}}[\varphi](a^\sharp \sqcap a_2^\sharp), \quad \text{where } \gamma(a_1^\sharp) \cup \gamma(a_2^\sharp) \supseteq \gamma(a^\sharp) \tag{18}$$

$$\widehat{\text{Assume}}[\ell](a^\sharp) \sqsubseteq \mu\widehat{\alpha}(\ell) \sqcap a^\sharp, \text{where } \ell \text{ is a literal in } \mathcal{L} \tag{19}$$

Eqn. (16) follows from the definition of $\widehat{\alpha}$ and $\widehat{\text{Assume}}$. Eqn. (17) follows from the definition of $\wedge$ and $\sqcap$, and corresponds to the simple deductive rules used in Stålmarck's algorithm. Eqn. (18) is the abstract-interpretation counterpart of the Dilemma Rule used in Stålmarck's method: the current goal $a^\sharp$ is split into sub-goals using meet ($\sqcap$), and the results of the sub-goals are combined using join ($\sqcup$). The correctness of this rule relies on the condition that $\gamma(a_1^\sharp) \cup \gamma(a_2^\sharp) \supseteq \gamma(a^\sharp)$. The $\mu\widehat{\alpha}$ operation in Eqn. (19) translates a literal in $\mathcal{L}$ into an abstract value in $\mathbb{A}$; that is $\mu\widehat{\alpha}(\ell) \stackrel{\text{def}}{=} \widehat{\alpha}(\ell)$. However, for certain combinations of $\mathcal{L}$ and $\mathbb{A}$, the $\mu\widehat{\alpha}$ operation is straightforward to implement—for example, when $\mathcal{L}$ is linear rational arithmetic (LRA) and $\mathbb{A}$ is the polyhedral domain [21]. $\mu\widehat{\alpha}$ can also be implemented using the RSY or bilateral algorithms when $\mathcal{L}$ and $\mathbb{A}$ satisfy the requirements for those frameworks.

The Stålmarck-based framework is based on much different principles from the RSY and bilateral frameworks for computing symbolic abstraction. The latter frameworks use an *inductive-learning approach* to learn from examples, while the Stålmarck-based 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/Bilateral framework, which uses a decision procedure as a black box, the Stålmarck-based framework adopts (and adapts) some principles from decision procedures.

Answers to The Four Questions.

Q1. The semantics of a statement *st* are specified as a two-vocabulary formula φ_{st} in some logic $\mathcal{L}$. In our work, we have typically used quantifier-free formulas over the theory of bitvectors and bitvector arrays (QF_ABV).

Q2. The abstract domain is specified via an interface consisting of the standard operations ($\sqcup$, $\sqcap$, etc.). The RSY and bilateral frameworks for symbolic abstraction require the β operation. The Stålmarck-based framework for symbolic abstraction requires the $\mu\widehat{\alpha}$ operation.

Q3. The various algorithms for $\widehat{\alpha}$ are the engines that apply/construct abstract transformers for a concrete transformer τ.

(a) The abstract execution of τ on $a^\sharp$ is performed via $a^{\sharp\prime} = \widehat{\alpha}\big(\varphi_\tau \wedge \widehat{\gamma}(a^\sharp)\big)$.
(b) The representation of the abstract transformer for τ is obtained via $\tau^\sharp = \widehat{\alpha}(\varphi_\tau)$.

Q4. The formula used to construct an abstract transformer can express the concrete semantics of (i) a basic block or (ii) a loop-free fragment (including a finite unrolling of a loop) à la large-block encoding [9] or adjustable-block encoding [10]. In our work, we used the TSL framework to obtain such formulas.

5.4 Automated Reasoning/Decision Procedures

Our investigation of symbolic abstraction led us to a new connection between decision procedures and abstract interpretation—namely, how to exploit abstract interpretation to provide new principles for designing decision procedures [82]. This work, which we call *Satisfiability Modulo Abstraction* (SMA), has led to new principles for designing decision procedures, and provides a way to create decision procedures for new logics. At the same time, it shows great promise from a practical standpoint. In other words, the methods for symbolic abstraction are "dual-use." In addition to providing methods for building improved abstract-interpretation tools, they also provide methods for building improved logic solvers that use abstract interpretation to speed up the search that a solver carries out.

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 [82], the polyhedral abstract domain—implemented in PPL [5]—is used to implement an SMA solver for the logic of linear rational arithmetic.

More recently, we created an SMA solver for separation logic [77]. Separation logic (SL) [68] is an expressive logic for reasoning about heap structures in programs, and provides a mechanism for concisely describing program states by explicitly localizing facts that hold in separate regions of the heap. SL is undecidable in general, but by using an abstract domain of shapes [70] we were able to design an unsatisfiability checker for SL.

5.5 Symbolic Abstraction and Quantifier Elimination

Gulwani and Musuvathi [32] 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 $[\![\exists V : \varphi]\!] \subseteq [\![\overline{\varphi}]\!]$.

(We use $\mathrm{Cover}_V(\varphi)$ to denote the cover of φ with respect to variable set V.)

Both $\mathrm{Cover}_V(\varphi)$ and $\widehat{\alpha}(\varphi)$ (deliberately) lose information from 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 $\mathrm{Cover}_V(\varphi)$ 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 $\widehat{\alpha}(\varphi)$ involves finding a formula ψ in an impoverished logic $\mathcal{L}'$: 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 express the semantics of a concrete transformer τ (e.g., Eqn. (8)). 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, the cover of φ_τ 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 Ex. 2.

Example 3. In Ex. 2, the application of 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 Eqn. (8). 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 pre-state registers $\{\texttt{eax}, \texttt{ebx}, \texttt{ecx}\}$. The cover of ψ with respect to R is

$$\mathrm{Cover}_R(\psi) \equiv \texttt{ebx}' = \begin{pmatrix} (\texttt{ecx}' \ \&\ \texttt{0xFFFF00FF}) \\ |\ ((\texttt{ecx}' + 256 * (\texttt{eax}' \ \&\ \texttt{0xFF}))\ \&\ \texttt{0xFF00}) \end{pmatrix} \tag{20}$$

Eqn. (20) shows that even though the result does not contain any unprimed registers, it is not an abstract value in the domain $\mathcal{E}_{2^{32}}$. □

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 $\widehat{\alpha}_{\mathcal{L}'}(\varphi) = \mathrm{Cover}_V(\varphi)$.

6 Connections with Other Areas of Computer Science

One of the most exciting aspects of the work on symbolic abstraction and automating the creation of abstract transformers is that the problem turns out to have many connections to other areas of Computer Science. Connections with automated reasoning and decision procedures were discussed in §5.4. Other connections include concept learning (§6.1) and constraint programming (§6.2).

6.1 Concept Learning

Reps et al. [64] identified a connection between the RSY algorithm for symbolic abstraction and the problem of *concept learning* in (classical) machine learning. In machine-learning terms, an abstract domain $\mathbb{A}$ 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 $\mathbb{A}$ not being the full power set of the concrete objects. Given a formula φ, the symbolic-abstraction problem is to find the most specific concept that explains the meaning of φ.

The RSY algorithm is related to the Find-S algorithm for concept learning [51, §2.4]. 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, the RSY algorithm 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; the RSY algorithm never sees a negative example.

A similar connection exists between the Bilateral algorithm and the Candidate-Elimination (CE) algorithm for concept learning [51, §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) in 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.

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 concept exists, which imposes limitations on the types of queries that the learning algorithm can make to an oracle (or teacher); see, for instance, [2, §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 over-approximation 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)* [83, 39]. The PAC guarantee also enables a learning algorithm to be applicable to concept lattices that are not complete lattices.

The similarities and differences between symbolic abstraction and concept learning open up opportunities for a richer exchange of ideas between the two areas. In particular, one can imagine situations in which it is appropriate for the over-approximation 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 [14], or to analyze programs with a probabilistic concrete semantics [41, 52, 22].

6.2 Constraint Programming

Constraint programming [54] is a declarative programming paradigm in which problems are expressed as conjunctions of first-order-logic formulas, called constraints. A constraint-satisfaction problem is defined by (i) a set of variables $V_1, \ldots, V_n$; (ii) a search space S given by a domain D_i for each variable V_i; and (iii) a set of constraints $\varphi_1, \ldots, \varphi_p$. The objective is to enumerate all variable valuations in the search space that satisfy every constraint. Different families of constraints come with specific operators—such as choice operators and propagators—used by the solver to explore the search space of the problem and to reduce its size, respectively. A constraint solver alternates two kinds of steps:

1. *Propagation steps* exploit constraints to reduce the domains of variables by removing values that cannot participate in a solution. The goal is to achieve *consistency*, when no more values can be removed.
2. When domains cannot be reduced further, the solver performs a *splitting step*: it makes an assumption about how to split a domain, and continues searching in the smaller search spaces.

The search proceeds, alternating propagation and splitting, until the search space contains either no solution, only solutions, or is smaller than a user-specified size. Backtracking may be used to explore other splitting assumptions.

Because the solution set cannot generally be enumerated exactly, continuous solvers compute a collection of intervals with floating-point bounds that contain all solutions and over-approximate the solution set while trying—on a best-effort basis—to include as few non-solutions as possible. In our terminology, such a constraint solver approaches $\widehat{\alpha}(\varphi)$ from above, for a conjunctive formula φ; the abstract domain is the disjunctive completion of the domain of environments of intervals; and the splitting and tightening steps are semantic reductions.

Several connections between abstract interpretation and constraint solving have been made in the past. Apt observed that applying propagators can be seen as an iterative fixpoint computation [3]. Pelleau et al. used this connection to describe a parameterized constraint solver that can be instantiated with different abstract domains [60]. Miné et al. describe a related algorithm to prove that a candidate invariant φ for a loop really is an invariant [50]. The goal is to identify a stronger invariant ψ that is both inductive and implies φ. The algorithm is parameterized on an abstract domain $\mathbb{A}$; the algorithm's actions are inspired by constraint solvers: it repeatedly splits and tightens non-overlapping elements of $\mathbb{A}$ (and therefore is searching for an inductive invariant in the disjunctive completion of $\mathbb{A}$). The algorithm works from "above" in the sense that it starts with (an under-approximation of) φ and creates descriptors of successively smaller areas of the state space as it searches for a suitable ψ.

7 Related Work

7.1 Best Abstract Transformers

In 1979, Cousot and Cousot [19] gave the specification of the best abstract transformer:

Let $\tau : \mathit{Store} \to \mathit{Store}$ be a concrete transformer and $\mathbb{C} = \mathcal{P}(\mathit{Store})$. Given a Galois connection $\mathbb{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{A}$, the *best abstract transformer*, defined by

$$\tau^{\sharp}_{best} \stackrel{\text{def}}{=} \alpha \circ \tilde{\tau} \circ \gamma, \tag{21}$$

is the most precise abstract transformer that over-approximates τ.

$\tau^{\sharp}_{best}$ establishes the limit of precision with which the actions of τ can be tracked using a given abstract domain $\mathbb{A}$. It provides a limit on what can be achieved by a system to automate the construction of abstract transformers. However, Eqn. (21) is non-constructive; it does not provide an *algorithm*, either for computing the result of applying $\tau^{\sharp}_{best}$ or for finding a representation of the function $\tau^{\sharp}_{best}$. In particular, the explicit application of γ to an abstract value would, in most cases, yield an intermediate set of concrete states that is either infinite or too large to fit into memory.

Graf and Saïdi [31] showed that theorem provers can be used to generate best abstract transformers for predicate-abstraction domains. In 2004, three papers appeared that concerned the problem of automatically constructing abstract transformers:

- Reps et al. [64] gave the method described in §5.3 for computing best transformers from below, which applies to a broader class of abstract domains than predicate-abstraction domains.
- Yorsh et al. [85] gave a method that works from above, for abstract domains based on canonical abstraction.
- Regehr and Reid [61] presented a method to construct abstract transformers for machine instructions, for interval and bitwise abstract domains. Their method is not based on logical reasoning, but instead uses a physical processor (or simulator) as a black box. To compute the abstract post-state for an abstract value $a^{\sharp}$, the approach recursively divides $a^{\sharp}$ 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.

Since then, a number of other methods for creating best abstract transformers have been devised [71, 53, 8, 40, 82, 78, 27]. (Some of them are discussed in §7.3.)

7.2 Heuristics for Good Transformers

With TVLA, a desired abstraction is specified by (i) defining the set of instrumentation relations $\mathcal{I}$ to use, and (ii) selecting which unary relations to use as abstraction relations $\mathcal{A}$. The abstract transformers are then constructed automatically by means of the four-part construction sketched in the paragraph "Maintaining Instrumentation Relations" of §3. There is no expectation that the abstract transformers constructed in this way are best transformers. However, practical experience with TVLA has shown that when the abstract domain is defined by the right sets of relations $\mathcal{I}$ and $\mathcal{A}$, TVLA produces excellent results.

Four theorems at the level of the framework—one for each part of the four-part construction—relieve the TVLA user from having to write the usual "near-commutativity" proofs of soundness that one finds in papers about one-off uses of

abstract interpretation.[6] These meta-level theorems are the key enabling factors that allow abstract transformers to be constructed automatically for canonical-abstraction domains.

The finite-differencing approach is generally able to retain an appropriate amount of precision because, for a concrete transformer τ_{st}, the application of the finite-differencing operators to an instrumentation relation p's defining formula ψ_p identifies the "footprint" of st on p. Knowledge of the footprint lets the relation-maintenance formula reuse as much information as possible from the pre-state structure, and thereby avoid performing formula-reevaluation operations for tuples whose values cannot be changed by st.

The term "footprint of a statement" also appears in work on abstract interpretation using separation logic (SL) [24, 15], but there it means a compact characterization of the concrete semantics of a statement in terms of the resources it accesses. In our terminology, footprints in the SL literature concern the core relations—i.e., the *independent variables* in the analogy with differentiation from §3. In this paper, when we refer to footprints, we mean the minimal effects of the concrete transformer on the instrumentation relations—which play the role of *dependent* variables.

The finite-differencing operators used in TVLA are most closely related to work on logic and databases: finite-difference operators for the propositional case were studied by Akers [1] and Sharir [73]. Work on (i) incrementally maintaining materialized views in databases [33], (ii) first-order incremental evaluation schemes [25], and (iii) dynamic descriptive complexity [59] have also addressed the problem of maintaining one or more auxiliary relations after new tuples are inserted into or deleted from base relations. In databases, view maintenance is solely an optimization; the correct information can always be obtained by reevaluating the defining formula. In the abstract-interpretation context, where abstraction has been performed, this is no longer true: reevaluating a formula in the local (3-valued) state can lead to a drastic loss of precision. Thus, the motivation for the work is completely different, although the techniques have strong similarities.

The method used in TVLA for finite differencing of formulas inspired some follow-on work using *numeric* finite differencing for program analysis [26]. That paper shows how to augment a numeric abstraction with numeric views, and gives a technique based on finite differencing to maintain an over-approximation of a view-variable's value in response to a transformation of the program state.

The idea of augmenting domains with instrumentation values has been used before in predicate-abstraction domains [31], which maintain the values of a given set of Boolean predicates. Graf and Saïdi [31] showed that decision procedures can be used to generate best abstract transformers for predicate-abstraction domains, but with high cost. Other work has investigated more efficient methods to

[6] (i) The correctness theorem for *focus* [70, Lems. 6.8 & 6.9]; (ii) the Embedding Theorem [70, Thm. 4.9]; (iii) the correctness theorem for the finite-differencing scheme for maintaining instrumentation relations [63, Thm. 5.3]; and (iv) the correctness theorem for *coerce* [70, Thm. 6.28].

generate approximate transformers that are not best transformers, but approach the precision of best transformers [7, 16]. Ball et al. [7] use a "focus" operation inspired by TVLA's *focus*, which as noted in footnote 4, plays a role similar to the splitting step in Stålmarck's algorithm.

Scherpelz et al. [72] developed a method for creating abstract transformers for use with parameterized predicate abstraction [17]. It performs WLP of a post-state relation with respect to transformer τ, followed by heuristics that attempt to determine combinations of pre-state relations that imply the WLP value. Generating the abstract transformer for a (nullary) instrumentation relation $p \in \mathcal{I}$, defined by the nullary formula $\psi_p()$, involves two steps:

1. Create the formula $\varphi = \mathrm{WLP}(\tau, \psi_p())$.
2. Find a Boolean combination $\nu_{p,\tau}$ of pre-state relations such that if $\nu_{p,\tau}$ holds in the pre-state, then φ must also hold in the pre-state (and hence p must hold in the post-state).

The abstract transformer is a function that sets the value of p in the post-state according to whether $\nu_{p,\tau}$ holds in the pre-state.

Because WLP performs substitution on $\psi_p()$, the formula created by step (1) is related to the substitution-based relation-maintenance formula defined in Eqn. (4). Step (2) applies several heuristics to φ to produce one or more strengthenings of φ; step (2) returns the disjunction of the strengthened variants of φ. In contrast, the finite-differencing algorithm discussed in §3 does not operate by trying to strengthen the substitution-based relation-maintenance formula; instead, it uses a systematic approach—based on finite differencing of p's defining formula $\psi_p()$—to identify how τ changes p's value. Moreover, the method is not restricted to nullary instrumentation relations: it applies to relations of arbitrary arity.

A special case of canonical abstraction occurs when no abstraction relations are used at all, in which case all individuals of a logical structure are collapsed to a single individual. When this is done, in almost all structures the only useful information remaining resides in the nullary core and instrumentation relations. Predicate abstraction can be seen as going one step further, retaining only the nullary instrumentation relations (and *no* abstracted core relations). However, to be able to evaluate a "Future" formula—as defined in Eqn. (7)—such as $\mathbf{F}_\tau[p] \stackrel{\text{def}}{=} p \,?\, \neg\Delta_\tau^-[p] : \Delta_\tau^+[p]$, one generally needs the pre-state abstract structure to hold (abstracted) core relations. From that standpoint, the finite-differencing method and that of Scherpelz et al. [72] are incomparable; they have different goals, and neither can be said to subsume the other.

Cousot et al. [20, §7] 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 [20, §7.2], and (ii) creating abstract transformers that transform input elements of a logical abstract domain to output elements of the domain [20, §7.3]. Their problems are particular cases of $\widehat{\alpha}(\varphi)$. They present heuristic methods for creating over-approximations of $\widehat{\alpha}(\varphi)$.

7.3 Symbolic Abstraction

Work on symbolic abstraction falls into three categories:

1. algorithms for specific domains [61, 47, 13, 8, 40, 27, 77, 43]
2. algorithms for parameterized abstract domains [31, 85, 71, 53]
3. abstract-domain frameworks [64, 82, 78].

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 approaches presented in §5 fall into category 3.

Some of the work mentioned above has already been discussed in §7.1.

Algorithms for specific domains. Brauer and King [13] 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 [53] (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 [48]; the update is expressed as an element of the abstract domain of rational affine equalities [38]. 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 $\widehat{\alpha}$ [40, Fig. 2], which works from below. Brauer and King show that optimal evaluation of such transfer functions requires linear programming. They give an example showing 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 [8] 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. [43] developed a symbolic-abstraction method for LRA, called SYMBA. The scenario considered by [43] is the following: 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 $\varphi \Rightarrow \big(\bigwedge_{1 \leq i \leq n} l_i \leq t_i \leq u_i\big)$. Similar to the bilat-

eral framework described in §5, the SYMBA algorithm maintains an under-approximation and an over-approximation of the final answer.

McMillan [47] 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 for which the algorithm is used is to compute don't-care conditions for logic synthesis.

Algorithms for parameterized abstract domains. Template Constraint Matrices (TCMs) are a parametrized family of linear-inequality domains for expressing invariants in linear real arithmetic. Sankaranarayanan et al. [71] gave a meet, join, and set of abstract transformers for all TCM domains. Monniaux [53] 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.

8 Conclusion

The algorithms developed in our research reduce the burden on analysis designers and implementers by raising the level of automation in abstraction interpretation. The work summarized in this paper focuses on the question "Given the specification of an abstraction, how does one create an execution engine for an analyzer that performs computations in an over-approximating fashion?" We know of only four systematic ways to address this question, three of which feature in our work:

1. Semantic reinterpretation and the related technique of syntax-directed reinterpretation (§4).
2. A strategy of splitting, propagation, and join à la the work on the generalized Stålmarck procedure [82] and TVLA [70, 63].
3. The approach illustrated by our bilateral algorithm, which uses concept learning via sampling, generalization, and abstract consequence to bound the answer from below and above.
4. Heuristic methods for formula normalization, for use with abstract domains in which abstract values are formulas in some logic ([24, §5.1] and [20, §7.3]).

The availability of automated methods for creating abstract transformers provides help along the following four dimensions:

Soundness: Creation of analyzers that are correct by construction, while requiring an analysis designer to implement only a small number of operations. Consequently, one only relies on a small "trusted computing base."

Precision: In contrast to most conventional approaches to creating abstract transformers, the use of symbolic abstraction can achieve the fundamental limits of precision that abstract-interpretation theory establishes.

Resource awareness: The algorithms for applying/constructing abstract transformers that approach $\widehat{\alpha}(\varphi_\tau)$ from above can be implemented as "any-

time" algorithms—i.e., an algorithm can be equipped with a monitor, and if the algorithm exhausts some time or space budget, the monitor can stop it at any time, and a safe (over-approximating) answer can be returned.

Extensibility: If an additional abstract domain is needed in an analyzer, automation makes it easy to add. In addition, for techniques 2 and 3, information can be exchanged automatically between domains via symbolic abstraction to improve the abstract values in each domain.

In terms of future research directions, we believe that because methods 2, 3, and 4 all provide a way to avoid the myopia of reinterpretation, they are all worthy of future research. In particular, for method 2, more results on partial-concretization and semantic-reduction operations are desirable, and for method 3, more results about abstract consequence are desirable. Finally, we believe that it will be fruitful to continue to explore the connections between the problems that arise in creating abstract transformers automatically and other areas of computer science.

Acknowledgments. T. Reps would like to thank the many people with whom he collaborated on the work described in the paper (as well as work that motivated the work described): for shape analysis: M. Sagiv, R. Wilhelm, a long list of their former students, as well as his own former students A. Loginov and D. Gopan; for machine-code analysis: G. Balakrishnan, J. Lim, Z. Xu, B. Miller, D. Gopan, A. Thakur, E. Driscoll, A. Lal, M. Elder, T. Sharma, and researchers at GrammaTech, Inc.; for symbolic abstraction: M. Sagiv, G. Yorsh, A. Thakur, M. Elder, T. Sharma, J. Breck, and A. Miné.

The work has been supported for many years by grants and contracts from NSF, DARPA, ONR, ARL, AFOSR, HSARPA, and GrammaTech, Inc. Special thanks go to R. Wachter, F. Anger, T. Teitelbaum and A. White.

Current support comes from a gift from Rajiv and Ritu Batra; DARPA under cooperative agreement HR0011-12-2-0012; AFRL under DARPA MUSE award FA8750-14-2-0270 and DARPA STAC award FA8750-15-C-0082; and the UW-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from WARF. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring organizations.

References

1. S. Akers, Jr. On a theory of Boolean functions. *J. SIAM*, 7(4):487–498, Dec. 1959.
2. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.
3. K. Apt. The essence of constraint propagation. *TCS*, 221, 1999.
4. G. Arnold, R. Manevich, M. Sagiv, and R. Shaham. Combining shape analyses by intersecting abstractions. In *VMCAI*, 2006.
5. R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *SCP*, 72(1–2):3–21, 2008.
6. G. Balakrishnan and T. Reps. WYSINWYX: What You See Is Not What You eXecute. *TOPLAS*, 32(6), 2010.

7. T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In *TACAS*, 2001.
8. E. Barrett and A. King. Range and set abstraction using SAT. *ENTCS*, 267(1), 2010.
9. D. Beyer, A. Cimatti, A. Griggio, M. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *FMCAD*, 2009.
10. D. Beyer, M. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *FMCAD*, 2010.
11. E. Boerger and R. Staerk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.
12. I. Bogudlov, T. Lev-Ami, T. Reps, and M. Sagiv. Revamping TVLA: Making parametric shape analysis competitive (tool paper). In *CAV*, 2007.
13. J. Brauer and A. King. Automatic abstraction for intervals using Boolean formulae. In *SAS*, 2010.
14. W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30:775–802, 2000.
15. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Footprint analysis: A shape analysis that discovers preconditions. In *SAS*, 2007.
16. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *FMSD*, 25(2–3), 2004.
17. P. Cousot. Verification by abstract interpretation. In *Verification: Theory and Practice*, 2003.
18. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
19. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.
20. P. Cousot, R. Cousot, and L. Mauborgne. Theories, solvers and static analysis by abstract interpretation. *J. ACM*, 59(6), 2012.
21. P. Cousot and N. Halbwachs. Automatic discovery of linear constraints among variables of a program. In *POPL*, 1978.
22. P. Cousot and M. Monerau. Probabilistic abstract interpretation. In *ESOP*, 2012.
23. W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22(3), Sept. 1957.
24. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.
25. G. Dong and J. Su. Incremental and decremental evaluation of transitive closure by first-order queries. *Inf. and Comp.*, 120:101–106, 1995.
26. M. Elder, D. Gopan, and T. Reps. View-augmented abstractions. *ENTCS*, 267(1), 2010.
27. M. Elder, J. Lim, T. Sharma, T. Andersen, and T. Reps. Abstract domains of affine relations. *TOPLAS*, 36(4), Jan. 2014.
28. Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symb. Comp.*, 12(4), 1999. Reprinted from Systems · Computers · Controls 2(5), 1971.
29. D. Gopan and T. Reps. Lookahead widening. In *CAV*, 2006.
30. D. Gopan and T. Reps. Guided static analysis. In *SAS*, 2007.
31. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, 1997.
32. S. Gulwani and M. Musuvathi. Cover algorithms and their combination. In *ESOP*, 2008.

33. A. Gupta and I. Mumick, editors. *Materialized Views: Techniques, Implementations, and Applications*. The M.I.T. Press, Cambridge, MA, 1999.
34. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The M.I.T. Press, 2006.
35. B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. *TOPLAS*, 32(2), 2010.
36. S. Johnson. YACC: Yet another compiler-compiler. Technical Report Comp. Sci. Tech. Rep. 32, Bell Laboratories, 1975.
37. N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.
38. M. Karr. Affine relationship among variables of a program. *Acta Inf.*, 6:133–151, 1976.
39. M. J. Kearns and U. V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, USA, 1994.
40. A. King and H. Søndergaard. Automatic abstraction for congruences. In *VMCAI*, 2010.
41. D. Kozen. Semantics of probabilistic programs. *JCSS*, 22(3), 1981.
42. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *SAS*, 2000.
43. Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, and M. Chechik. Symbolic optimization with smt solvers. In *POPL*, 2014.
44. J. Lim, A. Lal, and T. Reps. Symbolic analysis via semantic reinterpretation. *STTT*, 13(1):61–87, 2011.
45. J. Lim and T. Reps. TSL: A system for generating abstract interpreters and its application to machine-code analysis. *TOPLAS*, 35(1), 2013. Article 4.
46. K. Malmkjær. *Abstract Interpretation of Partial-Evaluation Algorithms*. PhD thesis, Dept. of Comp. and Inf. Sci., Kansas State Univ., 1993.
47. K. McMillan. Don't-care computation using k-clause approximation. *IWLS*, 2005.
48. A. Miné. The octagon abstract domain. In *WCRE*, 2001.
49. A. Miné. A few graph-based relational numerical abstract domains. In *SAS*, 2002.
50. A. Miné, J. Breck, and T. Reps. An algorithm inspired by constraint solvers to infer inductive invariants in numeric programs. Submitted for publication, 2015.
51. T. Mitchell. *Machine Learning*. WCB/McGraw-Hill, Boston, MA, 1997.
52. D. Monniaux. Abstract interpretation of probabilistic semantics. In *SAS*, 2000.
53. D. Monniaux. Automatic modular abstractions for template numerical constraints. *LMCS*, 6(3), 2010.
54. U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7(2):95–132, 1974.
55. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL*, 2004.
56. A. Mycroft and N. Jones. A relational framework for abstract interpretation. In *Programs as Data Objects*, 1985.
57. A. Mycroft and N. Jones. Data flow analysis of applicative programs using minimal function graphs. In *POPL*, 1986.
58. F. Nielson. Two-level semantics and abstract interpretation. *TCS*, 69, 1989.
59. S. Patnaik and N. Immerman. Dyn-FO: A parallel, dynamic complexity class. *JCSS*, 55(2):199–209, 1997.
60. M. Pelleau, A. Miné, C. Truchet, and F. Benhamou. A constraint solver based on abstract domains. In *VMCAI*, 2013.
61. J. Regehr and A. Reid. HOIST: A system for automatically deriving static analyzers for embedded systems. In *ASPLOS*, 2004.

62. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.
63. T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. *TOPLAS*, 6(32), 2010.
64. T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, 2004.
65. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *SCP*, 58(1–2), Oct. 2005.
66. T. Reps and A. Thakur. Through the lens of abstraction. In *HCSS*, 2014.
67. T. Reps, E. Turetsky, and P. Prabhu. Newtonian program analysis via tensor product. In *POPL*, 2016.
68. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
69. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *TOPLAS*, 20(1):1–50, Jan. 1998.
70. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.
71. S. Sankaranarayanan, H. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI*, 2005.
72. E. Scherpelz, S. Lerner, and C. Chambers. Automatic inference of optimizer flow functions from semantic meanings. In *PLDI*, 2007.
73. M. Sharir. Some observations concerning formal differentiation of set theoretic expressions. *TOPLAS*, 4(2):196–225, April 1982.
74. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.
75. M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods in System Design*, 16(1):23–58, 2000.
76. A. Thakur. *Symbolic Abstraction: Algorithms and Applications*. PhD thesis, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, Aug. 2014. Tech. Rep. 1812.
77. A. Thakur, J. Breck, and T. Reps. Satisfiability modulo abstraction for separation logic with linked lists. In *Spin Workshop*, 2014.
78. A. Thakur, M. Elder, and T. Reps. Bilateral algorithms for symbolic abstraction. In *SAS*, 2012.
79. A. Thakur, A. Lal, J. Lim, and T. Reps. PostHat and all that: Automating abstract interpretation. *ENTCS*, 311, 2015.
80. A. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. Reps. Directed proof generation for machine code. In *CAV*, 2010.
81. A. Thakur and T. Reps. A generalization of Stålmarck's method. In *SAS*, 2012.
82. A. Thakur and T. Reps. A method for symbolic computation of abstract operations. In *CAV*, 2012.
83. L. G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, 1984.
84. E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *POPL*, 2001.
85. G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS*, 2004.