A Method for Symbolic Computation of Abstract Operations*

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

[1] University of Wisconsin; Madison, WI, USA
[2] GrammaTech, Inc.; Ithaca, NY, USA

Abstract. In 1979, Cousot and Cousot gave a specification of the best (most-precise) abstract transformer possible for a given concrete transformer and a given abstract domain. Unfortunately, their specification does not lead to an algorithm for obtaining the best transformer. In fact, algorithms are known for only a few abstract domains.

This paper presents a parametric framework that, for a given abstract domain $\mathcal{A}$ and logic $\mathcal{L}$, computes successively better $\mathcal{A}$ values that over-approximate the set of states defined by an arbitrary formula in $\mathcal{L}$. Because the method approaches the most-precise $\mathcal{A}$ value from "above", if it is taking too much time, a safe answer can be returned at any time. For certain combinations of $\mathcal{A}$ and $\mathcal{L}$, the framework is complete—i.e., with enough resources, it computes the most-precise abstract value possible.

1 Introduction

Abstract interpretation [7] is a well-known technique for automatically proving certain kinds of program properties. The work described in this paper addresses the question of how to create abstract interpreters that, for some abstract domains, enjoy the best possible precision for a given abstraction, given sufficient resources. In addition to providing insight on fundamental limits, the algorithm presented in the paper also has good performance: our experiments showed a 3.5-fold speedup over a competing method [36, 23, 14] while finding dataflow facts that are equally precise or more precise at 96.8% of a program's basic blocks.

Cousot and Cousot [8] gave a *specification* of the most-precise abstract interpretation of a concrete operation that is possible in a given abstract domain. Suppose that $\mathcal{G} = \mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$ is a Galois connection defined by abstraction function $\alpha : \mathcal{C} \to \mathcal{A}$ and concretization function $\gamma : \mathcal{A} \to \mathcal{C}$. Given $\mathcal{G}$, the "best transformer", or best abstract post operator for transition τ, denoted by

* This paper is a substantial revision of [42]. This research is supported, in part, by NSF under grants CCF-{0810053, 0904371}, by ONR under grants N00014-{09-1-0510, 10-M-0251}, by ARL under grant W911NF-09-1-0413, and by AFRL under grants FA9550-09-1-0279 and FA8650-10-C-7088. 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 agencies. T. Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology reported in this publication.

$\widehat{\text{Post}}[\tau] : \mathcal{A} \to \mathcal{A}$, is the most precise abstract operator possible for the concrete post operator for τ, $\text{Post}[\tau] : \mathcal{C} \to \mathcal{C}$. $\widehat{\text{Post}}[\tau]$ can be expressed in terms of α, γ, and $\text{Post}[\tau]$, as follows [8]: $\widehat{\text{Post}}[\tau] = \alpha \circ \text{Post}[\tau] \circ \gamma$.

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

Symbolic Abstract Operations. $\widehat{\text{Post}}[\tau]$ is one of three inter-related symbolic operations that form the core repertoire at the heart of an abstract interpreter.

1. $\widehat{\alpha}(\varphi)$: Given a formula φ in logic $\mathcal{L}$, let $[\![\varphi]\!]_{\mathcal{L}}$ denote the meaning of φ in $\mathcal{L}$—i.e., the set of concrete states that satisfy φ. Given φ, $\widehat{\alpha}(\varphi)$ returns the best value in $\mathcal{A}$ that over-approximates $[\![\varphi]\!]_{\mathcal{L}}$ [36]. In particular, $\widehat{\alpha}(\varphi) = \alpha([\![\varphi]\!]_{\mathcal{L}})$.
2. $\widehat{\text{Assume}}[\varphi](A)$: Given $\varphi \in \mathcal{L}$ and $A \in \mathcal{A}$, $\widehat{\text{Assume}}[\varphi](A)$ returns the best value in $\mathcal{A}$ that over-approximates the meaning of φ in concrete states described by A. That is, $\widehat{\text{Assume}}[\varphi](A)$ equals $\alpha([\![\varphi]\!] \cap \gamma(A))$.
3. Creation of a representation of $\widehat{\text{Post}}[\tau]$: Some intraprocedural [16] and many interprocedural [39, 24] dataflow-analysis algorithms operate on instances of an abstract datatype $\mathcal{T}$ that (i) represents a family of abstract functions (or relations), and (ii) is closed under composition and join. By "creation of a representation of $\widehat{\text{Post}}[\tau]$", we mean finding the best instance in $\mathcal{T}$ that over-approximates $\text{Post}[\tau]$.

$\widehat{\alpha}$ (item 1) can be reduced to $\widehat{\text{Assume}}$ (item 2) as follows: $\widehat{\alpha}(\varphi) = \widehat{\text{Assume}}[\varphi](\top)$. Item 3 can be reduced to item 1. The concrete post operator $\text{Post}[\tau]$ is a transition relation between input states and output states. For item 3, the instances of abstract datatype $\mathcal{T}$ represent abstract-domain elements that denote sets of transition relations. In this case, we can create the best instance in $\mathcal{T}$ that over-approximates $\text{Post}[\tau]$ by performing $\widehat{\alpha}(\text{Post}[\tau])$.

Heretofore, algorithms for performing best abstract operations have been known for only a few abstract domains [15, 36, 43, 38, 23, 30, 14]. Moreover, there is a gap in current technology for performing best abstract operations: an algorithm is known for performing $\widehat{\alpha}$ for affine-relation analysis (ARA) [23, 14], and can be used to compute best ARA transformers. However, the algorithm makes repeated calls to an SMT (Satisfiability Modulo Theories) solver, and is ∼185x slower [14] than a compositional, syntax-directed method for creating sound, but not necessarily best, ARA transformers.

This paper presents a parametric framework that, for some abstract domains, is capable of performing most-precise abstract operations in the limit. Because the method approaches its result from "above", if the computation takes too much time it can be stopped to yield a safe result—i.e., an over-approximation to the best abstract operation—at any stage. Thus, the framework provides a tunable algorithm that offers a performance-versus-precision trade-off. In contrast, some other existing methods approach the result from "below" by maintaining

an under-approximation to the best abstract operation at any stage [36, 23, 14]. We replace "$\widehat{\ }$" with "$\widetilde{\ }$" to denote over-approximating operators—e.g., $\widetilde{\alpha}(\varphi)$, $\widetilde{\mathrm{Assume}}[\varphi](A)$, and $\widetilde{\mathrm{Post}}[\tau](A)$.[3]

In [41], we showed how abstract interpretation can provide insight on Stålmarck's method [40], an algorithm for validity checking of propositional formulas. In the present paper, we describe new ways in which ideas from Stålmarck's method can be adopted for use in program analysis. However, it is important to understand that the methods described in this paper are quite different from the huge amount of recent work that uses decision procedures in program analysis. It has become standard to reduce program paths to formulas by encoding a program's actions in logic (e.g., by symbolic execution) and calling a decision procedure to determine whether a given path through the program is feasible. In contrast, the techniques described in this paper do *not* reduce the problem of computing best abstract operations to Stålmarck's method; instead, the key ideas from Stålmarck's method are adopted—and adapted—to create new algorithms for key program-analysis operations.

Organization. §2 presents our framework at a semi-formal level. §3 presents the details of the framework. §4 describes two instantiations of it. §5 presents experimental results. §6 discusses applications to several other symbolic abstract operations. §7 discusses related work. Proofs can be found in [42].

2 Overview

Key Insight. In [41], we showed how Stålmarck's method [40] can be explained using abstract-interpretation terminology—in particular, as an instantiation of a more general algorithm, Stålmarck[$\mathcal{A}$], that is parameterized by an abstract domain $\mathcal{A}$ and operations on $\mathcal{A}$. The algorithm that goes by the name "Stålmarck's method" is one instantiation of Stålmarck[$\mathcal{A}$] with a certain abstract domain.

Abstract value A' is a *semantic reduction* [8] *of A with respect to φ* if (i) $\gamma(A') \cap [\![\varphi]\!] = \gamma(A) \cap [\![\varphi]\!]$, and (ii) $A' \sqsubseteq A$. At each step, Stålmarck[$\mathcal{A}$] holds some $A \in \mathcal{A}$; each of the so-called "propagation rules" employed in Stålmarck's method improves A by finding a semantic reduction of A with respect to φ.

The key insight of the present paper is that there is a connection between Stålmarck[$\mathcal{A}$] on the one hand and $\widehat{\alpha}_{\mathcal{A}}$, $\widehat{\mathrm{Assume}}_{\mathcal{A}}$, and $\widehat{\mathrm{Post}}[\tau]_{\mathcal{A}}$ on the other. In particular, to check whether a formula φ is valid, Stålmarck[$\mathcal{A}$] performs a one-sided test of whether φ is falsifiable by checking whether $\widehat{\alpha}_{\mathcal{A}}(\neg\varphi)$ equals $\bot_{\mathcal{A}}$. If the test succeeds, it establishes that $[\![\neg\varphi]\!] \subseteq \gamma(\bot_{\mathcal{A}}) = \emptyset$, and hence that φ is valid. In this paper, we adopt the same algorithmic structure used in Stålmarck[$\mathcal{A}$], and employ it in an algorithm that implements the three symbolic abstract operations $\widehat{\alpha}$, $\widehat{\mathrm{Assume}}$, and $\widehat{\mathrm{Post}}[\tau]$. Moreover, this paper goes beyond the classical setting of Stålmarck's method, which works with propositional logic,

[3] $\widetilde{\mathrm{Post}}[\tau]$ is used by Graf and Saïdi [15] to mean a different state transformer from the one that $\widetilde{\mathrm{Post}}[\tau]$ denotes in this paper. Throughout the paper, we use $\widetilde{\mathrm{Post}}[\tau]$ solely to mean an over-approximation of $\widehat{\mathrm{Post}}[\tau]$; thus, our notation is not ambiguous.

and assumes that we have a more expressive logic $\mathcal{L}$, such as quantifier-free linear rational arithmetic (QF_LRA) or quantifier-free bit-vector arithmetic (QF_BV). Because we are working with more expressive logics, our algorithm uses several ideas that go beyond what is used in Stålmarck's method [40] as well as what is used in Stålmarck[$\mathcal{A}$] [41].

Two Examples. We now illustrate the key points of our technique using two simple examples. The first shows how our technique applies to computing abstract transformers; the second describes its application to satisfiability checking.

Example 1. Consider the following x86 assembly code

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

The instruction at `L1` sets the zero flag (`zf`) to true if the value of register `eax` equals 2. At instruction `L2`, if `zf` is true the program jumps to location `L4` (not seen in the code snippet) by updating the value of the program counter (`pc`) to `L4`; otherwise, control falls through to program location `L3`. The transition formula that expresses the state transformation from the beginning of `L1` to the beginning of `L4` is thus $\varphi = (\texttt{zf} \Leftrightarrow (\texttt{eax} = 2)) \wedge (\texttt{pc}' = \mathit{ITE}(\texttt{zf}, \texttt{L4}, \texttt{L3})) \wedge (\texttt{pc}' = \texttt{L4}) \wedge (\texttt{eax}' = \texttt{eax})$. ($\varphi$ is a QF_BV formula.)

Let $\mathcal{A}$ be the abstract domain of affine relations over the x86 registers. Let $A_0 = \top_{\mathcal{A}}$, the empty set of affine constraints over input-state and output-state variables. We now describe how we compute $\widetilde{\mathrm{Post}}[\varphi] = \widetilde{\mathrm{Assume}}[\varphi](A_0)$, which represents a sound abstract transformer for use in affine-relation analysis (ARA) [32, 23, 14]. First, the ITE term in φ is rewritten as $(\texttt{zf} \Rightarrow (\texttt{pc}' = \texttt{L4})) \wedge (\neg\texttt{zf} \Rightarrow (\texttt{pc}' = \texttt{L3}))$. Thus, the transition formula becomes $\varphi = (\texttt{zf} \Leftrightarrow (\texttt{eax} = 2)) \wedge (\texttt{zf} \Rightarrow (\texttt{pc}' = \texttt{L4})) \wedge (\neg\texttt{zf} \Rightarrow (\texttt{pc}' = \texttt{L3})) \wedge (\texttt{pc}' = \texttt{L4}) \wedge (\texttt{eax}' = \texttt{eax})$.

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

1. Because φ has to be true, we can conclude that each of the conjuncts of φ are also true; that is, $\texttt{zf} \Leftrightarrow (\texttt{eax} = 2)$, $\texttt{zf} \Rightarrow (\texttt{pc}' = \texttt{L4})$, $\neg\texttt{zf} \Rightarrow (\texttt{pc}' = \texttt{L3})$, $\texttt{pc}' = \texttt{L4}$, and $\texttt{eax}' = \texttt{eax}$ are all true.
2. Suppose that we have a function $\mu\widetilde{\alpha}_{\mathcal{A}}$ such that for a literal $l \in \mathcal{L}$, $A' = \mu\widetilde{\alpha}_{\mathcal{A}}(l)$ is a sound overapproximation of $\widehat{\alpha}(l)$. Because the literal $\texttt{pc}' = \texttt{L4}$ is true, we conclude that $A' = \mu\widetilde{\alpha}_{\mathcal{A}}(\texttt{pc}' = \texttt{L4}) = \{\texttt{pc}' - \texttt{L4} = 0\}$ holds, and thus $A_1 = A_0 \sqcap A' = \{\texttt{pc}' - \texttt{L4} = 0\}$, which is a semantic reduction of A_0.
3. Similarly, because the literal $\texttt{eax}' = \texttt{eax}$ is true, we obtain $A_2 = A_1 \sqcap \mu\widetilde{\alpha}_{\mathcal{A}}(\texttt{eax}' = \texttt{eax}) = \{\texttt{pc}' - \texttt{L4} = 0, \texttt{eax}' - \texttt{eax} = 0\}$.
4. We know that $\neg\texttt{zf} \Rightarrow (\texttt{pc}' = \texttt{L3})$. Furthermore, $\mu\widetilde{\alpha}_{\mathcal{A}}(\texttt{pc}' = \texttt{L3}) = \{\texttt{pc}' - \texttt{L3} = 0\}$. Now $\{\texttt{pc}' - \texttt{L3} = 0\} \sqcap A_2$ is $\bot$, which implies that $[\![\texttt{pc}' = \texttt{L3}]\!] \cap \gamma(\{\texttt{pc}' - \texttt{L4} = 0, \texttt{eax}' - \texttt{eax} = 0\}) = \emptyset$. Thus, we can conclude that $\neg\texttt{zf}$ is false, and hence `zf` is true. This value of `zf`, along with the fact that $\texttt{zf} \Leftrightarrow (\texttt{eax} = 2)$ is true, enables us to determine that $A'' = \mu\widetilde{\alpha}_{\mathcal{A}}(\texttt{eax} = 2) = \{\texttt{eax} - 2 = 0\}$ must hold. Thus, our final semantic-reduction step produces $A_3 = A_2 \sqcap A'' = \{\texttt{pc}' - \texttt{L4} = 0, \texttt{eax}' - \texttt{eax} = 0, \texttt{eax} - 2 = 0\}$.

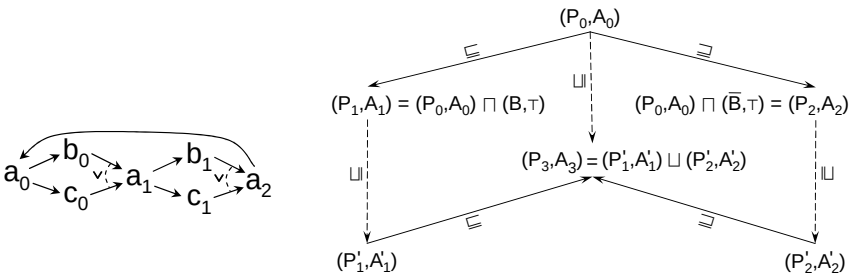


Fig. 1. (a) Inconsistent inequalities in the (unsatisfiable) formula used in Ex. 2. (b) Application of the Dilemma Rule to abstract value (P_0, A_0). The dashed arrows from (P_i, A_i) to (P'_i, A'_i) indicate that (P'_i, A'_i) is a semantic reduction of (P_i, A_i).

Abstract value A_3 is a set of affine constraints over the registers at `L1` (input-state variables) and those at `L4` (output-state variables), and can be used for affine-relation analysis using standard techniques (e.g., see [21] or [14, §5]). □

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

Example 2. Let $\mathcal{L}$ be QF_LRA, and let $\mathcal{A}$ be the polyhedral abstract domain [11]. Consider the formula $\psi = (a_0 < b_0) \wedge (a_0 < c_0) \wedge (b_0 < a_1 \vee c_0 < a_1) \wedge (a_1 < b_1) \wedge (a_1 < c_1) \wedge (b_1 < a_2 \vee c_2 < a_2) \wedge (a_2 < a_0) \in \mathcal{L}$ (see Fig. 1(a)). Suppose that we want to compute $\widetilde{\text{Assume}}[\psi](\top_{\mathcal{A}})$.

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

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

Consider the single-point partial assignment in which the value of u_1 is 1, i.e., $\top_{\mathcal{P}}[u_1 \mapsto 1]$. Because u_1 represents the root of ψ, $\top_{\mathcal{P}}[u_1 \mapsto 1]$ corresponds to the assertion that ψ is satisfiable. In fact, the models of ψ are closely related to the concrete values in $[\![\mathcal{I}]\!] \cap \gamma(\top_{\mathcal{P}}[u_1 \mapsto 1])$. For every concrete value in $[\![\mathcal{I}]\!] \cap \gamma(\top_{\mathcal{P}}[u_1 \mapsto 1])$, its projection onto $\{a_i, b_i, c_i \mid 0 \leq i \leq 1\} \cup \{a_2\}$ gives

us a model of ψ; that is, $[\![\psi]\!] = ([\![\mathcal{I}]\!] \cap \gamma(\top_{\mathcal{P}}[u_1 \mapsto 1]))|_{(\{a_i,b_i,c_i|0\leq i\leq 1\}\cup\{a_2\})}$. By this means, the problem of computing $\widetilde{\text{Assume}}[\psi](\top_{\mathcal{A}})$ is reduced to that of computing $\widetilde{\text{Assume}}[\mathcal{I}]((\top_{\mathcal{P}}[u_1 \mapsto 1], \top_{\mathcal{A}}))$, where $(\top_{\mathcal{P}}[u_1 \mapsto 1], \top_{\mathcal{A}})$ is an element of the cross product of $\mathcal{P}$ and $\mathcal{A}$.

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

To increase precision, we need to use the Dilemma Rule, a branch-and-merge rule, in which the current abstract state is split into two (disjoint) abstract states, 0-assume is applied to both abstract values, and the resulting abstract values are merged by performing a join. The steps of the Dilemma Rule are shown schematically in Fig. 1(b) and described below.

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

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

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

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

The process of repeatedly applying the Dilemma Rule is called 1-assume. That is, repeatedly some variable $u \in \mathcal{U}$ is selected whose truth value is unknown, the current abstract value is split using $B = \top_{\mathcal{P}}[u \mapsto 0]$ and $\overline{B} = \top_{\mathcal{P}}[u \mapsto 1]$, 0-assume is applied to each of these values, and the resulting abstract values are merged via join (Fig. 1(b)). Different policies for selecting the next variable on which to split can affect how quickly an answer is found; however, any fair

selection policy will return the same answer. The efficacy of the Dilemma Rule is partially due to case-splitting; however, the real power of the Dilemma Rule is due to the fact that it *preserves information learned in both branches when a case-split is "abandoned" at a join point.*

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

For certain abstract domains and logics, $\widetilde{\text{Assume}}[\psi](\top_{\mathcal{A}})$ is complete—i.e., with a high-enough value of k for k-assume, $\widetilde{\text{Assume}}[\psi](\top_{\mathcal{A}})$ always computes the most-precise $\mathcal{A}$ value possible for ψ. However, our experiments show that $\widetilde{\text{Assume}}[\psi](\top_{\mathcal{A}})$ has very good precision with $k = 1$ (see §5)—which jibes with the observation that, in practice, with Stålmarck's method for propositional validity (tautology) checking "a formula is either [provable with $k = 1$] or not a tautology at all!" [20, p. 227].

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

As discussed in §2, the top-level, overall goal of Stålmarck's method can be understood in terms of the operation $\widehat{\alpha}(\psi)$. However, Stålmarck's method is recursive (counting down on parameter k), and the operation performed at each recursive level is the slightly more general operation $\widehat{\text{Assume}}[\psi](A)$. Consequently, this section presents our algorithm for computing $\widetilde{\text{Assume}}[\varphi](A)$ in abstract domain $\mathcal{A}$, for φ in logic $\mathcal{L}$. The assumptions of our framework are as follows:

1. There is a Galois connection $\mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$ between $\mathcal{A}$ and concrete domain $\mathcal{C}$.
2. There is an algorithm to perform the join of arbitrary elements of $\mathcal{A}$; that is, for all $A_1, A_2 \in \mathcal{A}$, there is an algorithm that produces $A_1 \sqcup A_2$.
3. There is an algorithm to perform the meet of arbitrary elements of $\mathcal{A}$; that is, for all $A_1, A_2 \in \mathcal{A}$, there is an algorithm that produces $A_1 \sqcap A_2$.
4. Given a literal $l \in \mathcal{L}$, there is an algorithm $\mu\widetilde{\alpha}$ to compute a safe (overapproximating) "micro-$\widetilde{\alpha}$"—i.e., $A' = \mu\widetilde{\alpha}(l)$ such that $\gamma(A') \supseteq [\![l]\!]$.

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

Alg. 1 presents the algorithm that computes $\widetilde{\text{Assume}}[\varphi](A)$ for $\varphi \in \mathcal{L}$ and $A \in \mathcal{A}$. Line (1) calls the function integrity, which converts φ into integrity constraints $\mathcal{I}$ by assigning a fresh Boolean variable to each subformula of φ, using the rules described in Fig. 2. The variable u_φ corresponds to formula φ. We use $\mathcal{U}$ to denote the set of Boolean variables created when converting φ to $\mathcal{I}$. Alg. 1 also uses a second abstract domain, the Cartesian domain $\mathcal{P} = \mathcal{U} \to \{0, 1, *\}$, each of whose elements represents a set of Boolean assignments in $\mathbb{P}(\mathcal{U} \to \{0, 1\})$.

On line (2) of Alg. 1, an element of $\mathcal{P}$ is created in which u_φ is assigned the value 1, which asserts that φ is true. Alg. 1 is parameterized by the value of k (where $k \geq 0$). Let $\gamma_{\mathcal{I}}((P, A))$ denote $\gamma((P, A)) \cap [\![\mathcal{I}]\!]$. The call to k-assume on

Algorithm 1: $\widetilde{\text{Assume}}[\varphi](A)$

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

Algorithm 2: 0-assume$[\mathcal{I}]((P, A))$

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

Algorithm 3: k-assume$[\mathcal{I}]((P, A))$

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

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

Fig. 2. Rules used to convert a formula $\varphi \in \mathcal{L}$ into a set of integrity constraints $\mathcal{I}$. op represents any binary connective in $\mathcal{L}$, and literal$(\mathcal{L})$ is the set of atomic formulas and their negations.

line (3) returns $(\widetilde{P}, \widetilde{A})$, which is a semantic reduction of (P, A) with respect to $\mathcal{I}$; that is, $\gamma_{\mathcal{I}}((\widetilde{P}, \widetilde{A})) = \gamma_{\mathcal{I}}((P, A))$ and $(\widetilde{P}, \widetilde{A}) \sqsubseteq (P, A)$. In general, the greater the value of k, the more precise is the result computed by Alg. 1. The next theorem states that Alg. 1 computes an over-approximation of Assume$[\varphi](A)$.

Theorem 1 ([42]). *For all $\varphi \in \mathcal{L}, A \in \mathcal{A}$, if $\widetilde{A} = \widetilde{Assume}[\varphi](A)$, then $\gamma(\widetilde{A}) \supseteq [\![\varphi]\!] \cap \gamma(A)$, and $\widetilde{A} \sqsubseteq A$.* □

Alg. 3 presents the algorithm to compute k-assume, for $k \geq 1$. Given the integrity constraints $\mathcal{I}$, and the current abstract value (P, A), k-assume$[\mathcal{I}]((P, A))$ returns an abstract value that is a semantic reduction of (P, A) with respect to $\mathcal{I}$. The crux of the computation is the inner loop body, lines (4)–(10), which implements an analog of the Dilemma Rule from Stålmarck's method [40].

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

The calls to (k–1)-assume at lines (8) and (9) compute semantic reductions of (P_1, A_1) and (P_2, A_2) with respect to $\mathcal{I}$, which creates (P'_1, A'_1) and (P'_2, A'_2), respectively. Finally, at line (10) (P'_1, A'_1) and (P'_2, A'_2) are merged by performing a join. (The result is labeled (P_3, A_3) in Fig. 1(b).)

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

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

Fig. 3. Boolean rules used by Alg. 2 in the call InternalRule$(J, (P, A))$.

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

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

Fig. 4. Rules used by Alg. 2 in the call LeafRule$(J, (P, A))$.

The steps of the Dilemma Rule (Fig. 1(b)) are repeated until a fixpoint is reached, or some resource bound is exceeded. The correctness condition for Alg. 3 is stated as follows:

Theorem 2 ([42]). *For all* $P \in \mathcal{P}$ *and* $A \in \mathcal{A}$, *if* $(P', A') =$ k-assume$[\mathcal{I}]((P, A))$, *then* $\gamma_{\mathcal{I}}((P', A')) = \gamma_{\mathcal{I}}((P, A))$ *and* $(P', A') \sqsubseteq (P, A)$. □

Alg. 2 describes the algorithm to compute 0-assume: given the integrity constraints $\mathcal{I}$, and an abstract value (P, A), 0-assume$[\mathcal{I}]((P, A))$ returns an abstract value (P', A') that is a semantic reduction of (P, A) with respect to $\mathcal{I}$. It is in this algorithm that information is passed between the component abstract values $P \in \mathcal{P}$ and $A \in \mathcal{A}$ via *propagation rules*, like the ones shown in Figs. 3 and 4. In lines (4)–(7) of Alg. 2, these rules are applied by using a single integrity constraint in $\mathcal{I}$ and the current abstract value (P, A).

Given $J \in \mathcal{I}$ and (P, A), the net effect of applying any of the propagation rules is to compute a semantic reduction of (P, A) with respect to $J \in \mathcal{I}$. The propagation rules used in Alg. 2 can be classified into two categories:

1. Rules that apply on line (7) when J is of the form $p \Leftrightarrow (q \text{ op } r)$, shown in Fig. 3. Such an integrity constraint is generated from each internal subformula of formula φ. These rules compute a non-trivial semantic reduction from P with respect to J by only using information from P. For instance, rule And1 says that if J is of the form $p \Leftrightarrow (q \wedge r)$, and p is 1 in P, then we can infer that both q and r must be 1. Thus, $P \sqcap \top[q \mapsto 1, r \mapsto 1]$ is a semantic reduction of P with respect to J. (See Ex. 1, step 1.)
2. Rules that apply on line (5) when J is of the form $u \Leftrightarrow \ell$, shown in Fig. 4. Such an integrity constraint is generated from each leaf of the original formula φ. This category of rules can be further subdivided into
   (a) Rules that propagate information from abstract value P to abstract value A; viz., rules PtoA-0 and PtoA-1. For instance, rule PtoA-1 states

that given $J = u \Leftrightarrow l$, and $P(u) = 1$, then $A \sqcap \mu\widetilde{\alpha}(l)$ is a semantic reduction of A with respect to J. (See Ex. 1, steps 2 and 3.)

(b) Rule AtoP-0, which propagates information from abstract value A to abstract value P. Rule AtoP-0 states that if $J = (u \Leftrightarrow \ell)$ and $A \sqcap \mu\widetilde{\alpha}(l) = \bot_{\mathcal{A}}$, then we can infer that u is false. Thus, the value of $P \sqcap \top[u \mapsto 0]$ is a semantic reduction of P with respect to J. (See Ex. 1, step 4.)

Alg. 2 repeatedly applies the propagation rules until a fixpoint is reached, or some resource bound is reached. The next theorem states that 0-assume computes a semantic reduction of (P, A) with respect to $\mathcal{I}$.

Theorem 3 ([42]). *For all $P \in \mathcal{P}, A \in \mathcal{A}$, if $(P', A') = 0\text{-assume}[\mathcal{I}]((P, A))$, then $\gamma_{\mathcal{I}}((P', A')) = \gamma_{\mathcal{I}}((P, A))$ and $(P', A') \sqsubseteq (P, A)$.* □

4 Instantiations

In this section, we describe instantiations of our framework for two logical-language/abstract-domain pairs: QF_BV/KS and QF_LRA/Polyhedra. For each instantiation, we describe the $\mu\widetilde{\alpha}$ operation.

We say that the $\widetilde{\text{Assume}}$ algorithm is *complete* for an abstract domain $\mathcal{A}$ and logic $\mathcal{L}$ if it is guaranteed to compute the best value $\widehat{\text{Assume}}[\varphi](A)$ given $A \in \mathcal{A}, \varphi \in \mathcal{L}$, and a sufficiently large value of k for k-assume (where $k \leq |\varphi|$). We state completeness results for the two instantiations.

Bitvector Affine-Relation Domain (QF_BV/KS). King and Søndergaard [23] gave an algorithm for $\widehat{\alpha}$ for an abstract domain of Boolean affine relations. Elder et al. [14] extended the algorithm to affine relations in arithmetic modulo 2^w (i.e., for some bit-width w of bounded integers). Both algorithms work from below, making repeated calls on a SAT solver (King and Søndergaard) or an SMT solver (Elder et al.) and performing joins to incorporate more and more of the concrete state space into the current approximation of the final answer. Because the generalized technique is essentially the same as the Boolean case, we call both kinds of domains KS, and call the algorithm $\widehat{\alpha}^{\uparrow}_{\text{KS}}$.

Logic $\mathcal{L}$ in this case is QF_BV. Given a literal $l \in$ QF_BV, we compute $\mu\widetilde{\alpha}_{\text{KS}}(l)$ by invoking $\widehat{\alpha}^{\uparrow}_{\text{KS}}(l)$. That is, for QF_BV/KS we harness $\widehat{\alpha}^{\uparrow}_{\text{KS}}$ in service of $\widetilde{\text{Assume}}_{\text{KS}}$, but only for $\mu\widetilde{\alpha}_{\text{KS}}$, which means that $\widehat{\alpha}^{\uparrow}_{\text{KS}}$ is only applied to literals (i.e., small formulas). In practice, if the invocation of $\widehat{\alpha}^{\uparrow}_{\text{KS}}$ does not return an answer within a specified time limit, we use $\top_{\text{KS}}$.

The $\widetilde{\text{Assume}}$ algorithm is not complete for QF_BV/KS. Let x be a 2-bit-wide bitvector, and φ be the formula $(x \neq 0 \wedge x \neq 1 \wedge x \neq 2)$. Thus, $\widehat{\text{Assume}}[\varphi](\top_{\text{KS}}) = \{x - 3 = 0\}$. The KS abstract domain is not expressive enough to represent disequalities. For instance, if c is a constant, $\mu\widetilde{\alpha}(x \neq c)$ equals $\top_{\text{KS}}$. Thus, because the $\widetilde{\text{Assume}}$ algorithm considers only a single integrity constraint at a time, we get $\widetilde{\text{Assume}}[\varphi](\top_{\text{KS}}) = \mu\widetilde{\alpha}(x \neq 0) \sqcap \mu\widetilde{\alpha}(x \neq 1) \sqcap \mu\widetilde{\alpha}(x \neq 2) = \top_{KS}$.

The current approach can be made complete for QF_BV/KS by either (i) making 0-assume consider multiple integrity constraints during propagation (in the limit, having to call $\mu\widetilde{\alpha}(\varphi)$), or (ii) performing a 2^w-way split on the current

KS abstract value, where w is the bitvector length, each time a disequality is encountered; effectively rewriting $x \neq 0$ to $(x = 1 \vee x = 2 \vee x = 3)$. Both of these approaches would be prohibitively expensive. Our current approach, though theoretically not complete, works very well in practice (see §5).

Polyhedral Domain (QF_LRA/Polyhedra). The second instantiation that we implemented is for the logic QF_LRA and the polyhedral domain [11]. Because a QF_LRA disequality $t \neq 0$ can be normalized to $(t < 0 \vee t > 0)$, every literal l in a normalized QF_LRA formula is merely a half-space in the polyhedral domain. Consequently, $\mu\widetilde{\alpha}_{\text{Polyhedra}}(l)$ is exact, and easy to compute. Furthermore, because of this precision, the $\widetilde{\text{Assume}}$ algorithm is complete for QF_LRA/Polyhedra. In particular, if $k = |\varphi|$, then k-assume is sufficient to guarantee that $\widetilde{\text{Assume}}[\varphi](A)$ returns $\widehat{\text{Assume}}[\varphi](A)$. The QF_LRA/Polyhedra instantiation uses the Parma Polyhedra Library [34].

5 Experiments

Bitvector Affine-Relation Analysis. In this section, we compare two methods for computing the abstract transformers for the KS domain for affine relation analysis (ARA) [23]:

- the $\widehat{\alpha}^{\uparrow}$-based procedure described in Elder et al. [14].
- the $\widetilde{\alpha}$-based procedure described in this paper ("$\widetilde{\alpha}^{\downarrow}$"), instantiated for KS.

Our experiments were designed to answer the following questions:

1. How does the speed of $\widetilde{\alpha}^{\downarrow}$ compare with that of $\widehat{\alpha}^{\uparrow}$?
2. How does the precision of $\widetilde{\alpha}^{\downarrow}$ compare with that of $\widehat{\alpha}^{\uparrow}$?

To address these questions, we performed ARA on x86 machine code, computing affine relations over the x86 registers. Our experiments were run on a single core of a quad-core 3.0 GHz Xeon computer running 64-bit Windows XP (SP2), configured so that a user process has 4GB of memory. We analyzed a corpus of Windows utilities using the WALi [22] system for weighted pushdown systems (WPDSs). For the baseline $\widehat{\alpha}^{\uparrow}$-based analysis we used a weight domain of $\widehat{\alpha}^{\uparrow}$-generated KS transformers. The weight on each WPDS rule encodes the KS transformer for a basic block B of the program, including a jump or branch to a successor block. A formula φ_B is created that captures the concrete semantics of B, and then the KS weight for B is obtained by performing $\widehat{\alpha}^{\uparrow}(\varphi_B)$ (cf. Ex. 1). We used EWPDS merge functions [26] to preserve caller-save and callee-save registers across call sites. The post* query used the FWPDS algorithm [25].

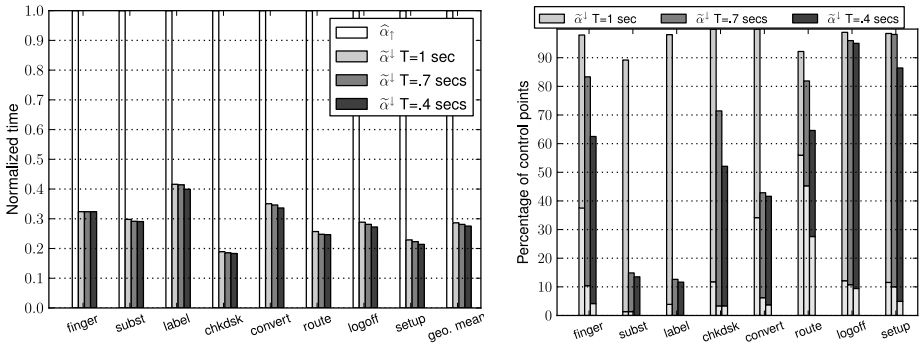
Fig. 5 lists several size parameters of the examples (number of instructions, procedures, basic blocks, and branches) along with the times for constructing abstract transformers and running post*.[4] Col. 6 of Fig. 5 shows that the calls to $\widehat{\alpha}^{\uparrow}$ during WPDS construction dominate the total time for ARA.

Each call to $\widehat{\alpha}^{\uparrow}$ involves repeated invocations of an SMT solver. Although the overall time taken by $\widehat{\alpha}^{\uparrow}$ is not limited by a timeout, we use a 3-second timeout

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

Prog.	Measures of size				$\widehat{\alpha}^{\uparrow}$ Performance			
name	instrs	CFGs	BBs	brs	WPDS	t/o	post*	query
finger	532	18	298	48	110.9	4	0.266	0.015
subst	1093	16	609	74	204.4	4	0.344	0.016
label	1167	16	573	103	148.9	2	0.344	0.032
chkdsk	1468	18	787	119	384.4	16	0.219	0.031
convert	1927	38	1013	161	289.9	9	1.047	0.062
route	1982	40	931	243	562.9	14	1.281	0.046
logoff	2470	46	1145	306	621.1	16	1.938	0.063
setup	4751	67	1862	589	1524.7	64	0.968	0.047

Fig. 5. WPDS experiments ($\widehat{\alpha}^{\uparrow}$). The columns show the number of instructions (instrs); the number of procedures (CFGs); the number of basic blocks (BBs); the number of branch instructions (brs); the times, in seconds, for WPDS construction with $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$ weights, running post*, and finding one-vocabulary affine relations at blocks that end with branch instructions (query). The number of basic blocks for which $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$-weight generation timed out is listed under "t/o".

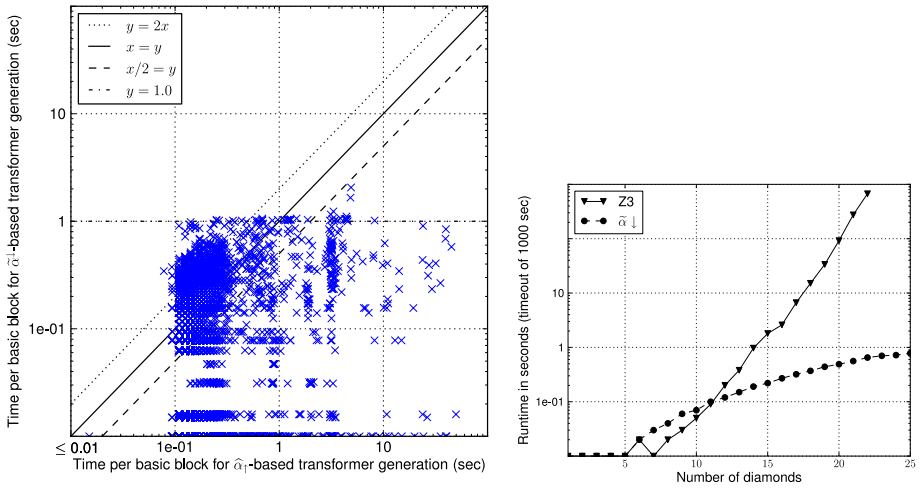


(a) (b)

Fig. 6. (a) Performance: $\widetilde{\alpha}^{\downarrow}$ vs. $\widehat{\alpha}^{\uparrow}$. (b) Precision: % of control points at which $\widetilde{\alpha}^{\downarrow}$ has as good or better precision as $\widehat{\alpha}^{\uparrow}$; the lighter-color lower portion of each bar indicates the % of control points at which the precision is strictly greater for $\widetilde{\alpha}^{\downarrow}$.

for each invocation of the SMT solver (as in Elder et al. [14]). Fig. 5 lists the number of such SMT solver timeouts for each benchmark. In case the invocation of the SMT solver times out, $\widehat{\alpha}^{\uparrow}$ is forced to return $\top_{\mathrm{KS}}$ in order to be sound. (Consequently, it is possible for $\widetilde{\alpha}^{\downarrow}$ to return a more precise answer than $\widehat{\alpha}^{\uparrow}$.)

The setup for the $\widetilde{\alpha}^{\downarrow}$-based analysis is the same as the baseline $\widehat{\alpha}^{\uparrow}$-based analysis, except that we call $\widetilde{\alpha}^{\downarrow}$ when calculating the KS weight for a basic block. We use 1-assume in this experiment. Each basic-block formula φ_B is rewritten to a set of integrity constraints, with ITE-terms rewritten as illustrated in Ex. 1. The priority of a Boolean variable is its postorder-traversal number, and is used to select which variable is used in the Dilemma Rule. We bound the total time taken by each call to $\widetilde{\alpha}^{\downarrow}$ to a fixed timeout $\mathtt{T}$. Note that even when the call to $\widetilde{\alpha}^{\downarrow}$ times out, it can still return a sound non-$\top_{\mathrm{KS}}$ value. We ran $\widetilde{\alpha}^{\downarrow}$ using $\mathtt{T} = 1$ sec, $\mathtt{T} = 0.7$ secs, and $\mathtt{T} = 0.4$ secs.



(a) (b)

Fig. 7. (a) Log-log scatter plot of transformer-construction time. (b) Semilog plot of Z3 vs. $\widetilde{\alpha}^{\downarrow}$ on χ_d formulas.

Fig. 6(a) shows the normalized time taken for WPDS construction when using $\widetilde{\alpha}^{\downarrow}$ with $\mathtt{T} = 1$ sec, $\mathtt{T} = 0.7$ secs, and $\mathtt{T} = 0.4$ secs. The running time is normalized to the corresponding time taken by $\widehat{\alpha}^{\uparrow}$; lower numbers are better. WPDS construction using $\widetilde{\alpha}^{\downarrow}$ with $\mathtt{T} = 1$ sec. is about 3.5 times faster than $\widehat{\alpha}^{\uparrow}$ (computed as the geometric mean), which answers question 1.

Decreasing the timeout $\mathtt{T}$ makes the $\widetilde{\alpha}^{\downarrow}$ WPDS construction only slightly faster. To understand this behavior better, we show in Fig. 7(a) a log-log scatter-plot of the times taken by $\widehat{\alpha}^{\uparrow}$ versus the times taken by $\widetilde{\alpha}^{\downarrow}$ (with $\mathtt{T} = 1$ sec.), to generate the transformers for each basic block in the benchmark suite. As shown in Fig. 7(a), the times taken by $\widetilde{\alpha}^{\downarrow}$ are bounded by 1 second. (There are a few calls that take more than 1 second; they are an artifact of the granularity of operations at which we check whether the procedure has timed out.) Most of the basic blocks take less than 0.4 seconds, which explains why the overall time for WPDS construction does not decrease much as we decrease $\mathtt{T}$ in Fig. 6(a). We also see that the $\widehat{\alpha}^{\uparrow}$ times are not bounded, and can be as high as 50 seconds.

To answer question 2 we compared the precision of the WPDS analysis when using $\widetilde{\alpha}^{\downarrow}$ with $\mathtt{T}$ equal to 1, 0.7, and 0.4 seconds with the precision obtained using $\widehat{\alpha}^{\uparrow}$. In particular, we compare the affine relations computed by the $\widetilde{\alpha}^{\downarrow}$-based and $\widehat{\alpha}^{\uparrow}$-based analyses for each *control point*—i.e., the beginning of a basic block that ends with a branch. Fig. 6(b) shows the percentage of control points for which the $\widetilde{\alpha}^{\downarrow}$-based analysis computes a better or equally good affine relation. When using $\mathtt{T}$= 1 sec, $\widetilde{\alpha}^{\downarrow}$ computes as good a result as $\widehat{\alpha}^{\uparrow}$ at 96.8% of the control points (geometric mean). Interestingly, $\widetilde{\alpha}^{\downarrow}$ often computes an answer that is more precise compared to that computed by $\widehat{\alpha}^{\uparrow}$. That is not a bug in our implementation; it happens because $\widehat{\alpha}^{\uparrow}$ has to return $\top_{\mathrm{KS}}$ when the call to the

SMT solver times out. In Fig. 6(b), the lighter-color lower portion of each bar shows the percentage of control points for which $\widetilde{\alpha}^{\downarrow}$ provides strictly more precise answers when compared to $\widehat{\alpha}^{\uparrow}$. Furthermore, as expected, when the timeout for $\widetilde{\alpha}^{\downarrow}$ is reduced, the precision decreases.

Satisfiability Checking. The formula used in Ex. 2 is just one instance of a family of unsatisfiable QF_LRA formulas [28]. Let $\chi_d = (a_d < a_0) \wedge \bigwedge_{i=0}^{d-1}((a_i < b_i) \wedge (a_i < c_i) \wedge ((b_i < a_{i+1}) \vee (c_i < a_{i+1})))$. The formula ψ in Ex. 2 is χ_2; that is, the number of "diamonds" is 2 (see Fig. 1(a)). We used the QF_LRA/polyhedra instantiation of our framework to check whether $\widetilde{\alpha}(\chi_d) = \bot$ for $d = 1 \ldots 25$ using 1-assume. We ran this experiment on a single processor of a 16-core 2.4 GHz Intel Zeon computer running 64-bit RHEL Server release 5.7. The semilog plot in Fig. 7(b) compares the running time of $\widetilde{\alpha}^{\downarrow}$ with that of Z3, version 3.2 [12]. The time taken by Z3 increases exponentially with d, exceeding the timeout threshold of 1000 seconds for $d = 23$. This corroborates the results of a similar experiment conducted by McMillan et. al [28], where the reader can also find an in-depth explanation of this behavior.

On the other hand, the running time of $\widetilde{\alpha}^{\downarrow}$ increases linearly with d taking 0.78 seconds for $d = 25$. The cross-over point is $d = 12$. In Ex. 2, we saw how two successive applications of the Dilemma Rule suffice to prove that ψ is unsatisfiable. That explanation generalizes to χ_d: d applications of the Dilemma Rule are sufficient to prove unsatisfiability of χ_d. The order in which Boolean variables with unknown truth values are selected for use in the Dilemma Rule has no bearing on this linear behavior, as long as no variable is starved from being chosen (i.e., a fair choice-schedule is used). Each application of the Dilemma Rule is able to infer that $a_i < a_{i+1}$ for some i.

We do not claim that $\widetilde{\alpha}^{\downarrow}$ is better than mature SMT solvers such as Z3. We do believe that it represents another interesting point in the design space of SMT solvers, similar in nature to the GDPLL algorithm [28] and the k-lookahead technique used in the DPLL($\sqcup$) algorithm [4].

6 Applications to Other Symbolic Operations

The operation of symbolic concretization, denoted by $\widehat{\gamma}$, maps an abstract value $A \in \mathcal{A}$ to a formula $\widehat{\gamma}(A)$ such that A and $\widehat{\gamma}(A)$ represent the same set of concrete states (i.e., $\gamma(A) = [\![\widehat{\gamma}(A)]\!]$). Experience shows that the assumption that $\mathcal{A}$ supports symbolic concretization is not a significant restriction because it is easy to write the $\widehat{\gamma}$ function for most abstract domains.

In contrast with $\gamma(A)$, $\widehat{\gamma}(A)$ produces a finite-sized formula that can be manipulated in computer memory. Thus, the problem raised in §1 about $\gamma(A)$ producing a result that is either infinite or too large to fit in computer memory can be side-stepped by using $\widehat{\gamma}$ and performing subsequent operations on the formulas that $\widehat{\gamma}$ produces. However, one must now work with symbolic representations of sets of states (i.e., formulas), and for each of the operations needed in abstract interpretation the challenge is to develop an algorithm that operates on formulas. §3 has shown how $\widetilde{\text{Assume}}$, and hence $\widetilde{\alpha}$, can be implemented.

The symbolic operations of $\widehat{\gamma}$ and $\widehat{\alpha}$ can be used to implement a number of other useful operations, as discussed below. In each case, over-approximations result if $\widehat{\alpha}$ is replaced by $\widetilde{\alpha}$.

- The operation of containment checking, $A_1 \sqsubseteq A_2$, which is needed by analysis algorithms to determine when a post-fixpoint is attained, can be implemented by checking whether $\widehat{\alpha}(\widehat{\gamma}(A_1) \wedge \neg\widehat{\gamma}(A_2))$ equals $\bot$.
- Suppose that there are two Galois connections $\mathcal{G}_1 = \mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}_1$ and $\mathcal{G}_2 = \mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}_2$, and one wants to work with the reduced product of $\mathcal{A}_1$ and $\mathcal{A}_2$ [8, §10.1]. The semantic reduction of a pair (A_1, A_2) can be performed by letting ψ be the formula $\widehat{\gamma}_1(A_1) \wedge \widehat{\gamma}_2(A_2)$, and creating the pair $(\widehat{\alpha}_1(\psi), \widehat{\alpha}_2(\psi))$.
- Given $A_1 \in \mathcal{A}_1$, one can find the most-precise value $A_2 \in \mathcal{A}_2$ that over-approximates A_1 in $\mathcal{A}_2$ as follows: $A_2 = \widehat{\alpha}_2(\widehat{\gamma}_1(A_1))$.
- Given a loop-free code fragment F, consisting of one or more blocks of program statements and conditions, one can obtain a representation of its best transformer by symbolically executing F to obtain a transition formula ψ_F, and then performing $\widehat{\alpha}(\psi_F)$.

7 Related Work

Extensions of Stålmarck's Method. Björk [3] describes extensions of Stålmarck's method to first-order logic. (Björk credits Stålmarck with making the first extension of the method in that direction, and mentions an unpublished manuscript of Stålmarck's.) Like Björk, our work goes beyond the classical setting of Stålmarck's method [40] (i.e., propositional logic) and extends the method to more expressive logics, such as QF_LRA or QF_BV. However, Björk is concerned solely with validity checking, and—compared with the propositional case—the role of abstraction is less clear in his method. Our algorithm not only uses an abstract domain as an explicit datatype, the goal of the algorithm is to compute an abstract value $A' = \widetilde{\text{Assume}}[\varphi](A)$.

Our approach was influenced by Granger's method of using (in)equation solving as a way to implement semantic reduction and Assume as part of his technique of *local decreasing iterations* [17]. Granger describes techniques for performing reductions with respect to (in)equations of the form $x_1 \bowtie F(x_1, \ldots, x_n)$ and $(x_1 * F(x_1, \ldots, x_n)) \bowtie G(x_1, \ldots, x_n)$, where $\bowtie$ stands for a single relational symbol of $\mathcal{L}$, such as $=$, $\neq$, $<$, $\leq$, $>$, $\geq$, or $\equiv$ (arithmetical congruence). Our framework is not limited to literals of these forms; all that we require is that for a literal $l \in \mathcal{L}$, there is an algorithm to compute an overapproximating value $\mu\widetilde{\alpha}(l)$. Moreover, Granger has no analog of the Dilemma Rule, nor does he present any completeness results (cf. §4).

A variant of the Dilemma Rule is used in DPLL($\sqcup$), and allows the theory solver in a lazy DPLL-based SMT solver to produce joins of facts deduced along different search paths. However, as pointed out by Bjørner et al. [4, §5], their system is weaker than Stålmarck's method, because Stålmarck's method can learn equivalences between literals.

Another difference between our work and Bjørner et al. is the connection presented in this paper between Stålmarck's method and the computation of best abstract operations for abstract interpretation.

In [41], we studied Stålmarck's method from the perspective of abstract interpretation, and gave an account in which we explained each of its key components in terms of well-known abstract-interpretation techniques. We then used those insights to devise a framework for propositional-logic validity-checking algorithms that is parametrized by an abstract domain and operations on the domain. The algorithm that goes by the name "Stålmarck's method" is one particular instantiation of our framework with a certain abstract domain. The work reported in [41] shows how abstract interpretation offers insight on the design of decision procedures for propositional logic. In contrast, the present paper describes ways in which ideas from Stålmarck's method can be adopted for use in symbolic abstract operations, such as $\widetilde{\alpha}(\varphi)$ and $\widetilde{\text{Assume}}[\varphi](A)$, as well as creating a representation of $\widetilde{\text{Post}}[\tau]$. On the other hand, the more expressive Boolean abstraction domains described in [41] can be used instead of the Cartesian abstraction domain in the $\widetilde{\text{Assume}}$ algorithm [42].

We recently became aware that Haller and D'Silva [19] are (jointly) engaged in research with similar goals to ours, but in a somewhat different domain. They have given an abstract-interpretation-based account of Conflict-Driven Clause Learning (CDCL) SAT solvers [31]. Our work and that of Haller and D'Silva were performed independently and contemporaneously. Recently, they have also lifted their technique from a propositional SAT solver to a floating-point decision procedure that makes use of floating-point intervals [13].

Best Abstract Operations. Several papers about best abstract operations have appeared in the literature [15, 36, 43, 23, 14]. Graf and Saïdi [15] showed that decision procedures can be used to generate best abstract transformers for predicate-abstraction domains. Other work has investigated more efficient methods to generate approximate transformers that are not best transformers, but approach the precision of best transformers [1, 6].

Several techniques work from *below* [36, 23, 14]—performing joins to incorporate more and more of the concrete state space—which has the drawback that if they are stopped before the final answer is reached, the most-recent approximation is an *under-approximation* of the desired value. The issue is all the more problematic because the technique makes repeated calls to an SMT solver, and thus if one sets a timeout threshold for the solver, there must be a backup strategy invoked whenever the timeout threshold is exceeded. In contrast, our technique works from *above*, performing meets to eliminate more and more of the concrete state space. It can stop at any time and return a safe answer.

Yorsh et al. [43] developed a method that works from above to perform $\widetilde{\text{Assume}}[\varphi](A)$ for the kind of abstract domains used in shape analysis (i.e., "canonical abstraction" of logical structures [37]). Their method has a splitting step, but no analog of the join step performed at the end of an invocation of the Dilemma Rule. In addition, their propagation rules are much more heavyweight. As discussed in §2, our propagation rules are local: they make use of only a *single*

integrity constraint. In contrast, the step that corresponds to propagation in the method of Yorsh et al. repeatedly passes the *entire formula* φ to the theorem prover.

Template Constraint Matrices (TCMs) are a parametrized family of linear-inequality domains for expressing invariants in linear real arithmetic. Sankaranarayanan et al. [38] gave a parametrized meet, join, and set of abstract transformers for all TCM domains. Monniaux [30] 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.

Brauer and King [5] developed a method that works from below to derive abstract transformers for the interval domain. Their method is based on Monniaux's general approach, 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 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 [29]; the update operation is expressed as an element of the abstract domain of rational affine equalities [21]. 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 $\widehat{\alpha}$ [23, Fig. 2], which works from below, as discussed earlier. Brauer and King show that optimal evaluation of such transfer functions requires linear programming. They give an example that demonstrates that an octagon-closure operation on a combination of the guard's octagon and the update's affine equality is sub-optimal.

Barrett and King [2] 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., $\widehat{\alpha}$).

Regehr and Reid [35] present a method that constructs abstract transformers for machine instructions, for interval and bitwise abstract domains. Their method does not call a SAT solver, but instead uses the physical processor (or a simulator of a processor) as a black box. To compute the abstract post-state for an abstract value A, the approach recursively subdivides A into A_1 and A_2, computes the post-states for A_1 and A_2, and joins the results. The algorithm tabulates abstract results for all combinations of abstract inputs. The algorithm keep subdividing

an abstract input 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 different post-state values are joined together as the recursion is unwound.

The approach of subdividing, with an eventual join of the results, is similar to the split-and-merge steps of the Dilemma Rule used in our approach. However, the algorithm of Regehr and Reid takes advantage of the subdivision in a limited way; that is, recursive subdivision is exploited merely to be able to cache previously computed results. Their goal is to speed up the algorithm by caching results that have been computed previously, when the process was applied to other inputs whose recursive subdivision led to the same combination of abstact values as inputs. In contrast, in our work the split performed by the Dilemma Rule enables the application of propagation rules. The algorithm of Regehr and Reid can be seen as an instance of our framework in which—for n-bit bit-vectors—they use n-assume, and the only propagation rules are for concrete evaluation of an arithmetic operation.

Cousot et al. [10] 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 [10, §7.1], and (ii) creating abstract transformers that transform input elements of a logical abstract domain to output elements of the domain [10, §7.2]. Their problems are particular cases of $\widehat{\alpha}$ (or $\widetilde{\alpha}$) and $\widehat{\text{Post}}[\tau]$ (or $\widetilde{\text{Post}}[\tau]$). They present heuristic methods for creating $\widetilde{\alpha}$ and $\widetilde{\text{Post}}[\tau]$ based on literal elimination and quantifier elimination.

Work on Combining Abstract Domains. The use of propagation rules to exchange information between the $\mathcal{P}$ and $\mathcal{A}$ components of a paired abstract value is related to previous research on combining abstract domains. It is also reminiscent of the Nelson-Oppen technique for combining decision procedures [33]. However, whereas the Nelson-Oppen technique is limited to sharing equalities, the propagation rules in our framework can share relations other than equality: the symbol $\bowtie$ in the rules in Fig. 4 can be $=$, $\neq$, $<$, $\leq$, $>$, $\geq$, $\equiv$ (arithmetic congruence), or other relational symbols of the logic $\mathcal{L}$.

Cousot and Cousot [8] defined the reduced product, which allows abstract values in different domains to influence each other's value. In ASTRÉE, Cousot et al. [9] use a hierarchical organization for communication between multiple abstract domains to implement an over-approximation of the reduced product. An analysis can employ multiple hierarchies; each hierarchy uses a fixed communication pattern among domains to implement a partially reduced product. Gulwani and Tiwari [18] gave a method for combining abstract interpreters, based on the Nelson-Oppen method. As in Nelson-Oppen, communication between domains in their framework is solely via equalities. McCloskey et al. [27] presented a framework for communication between abstract domains that goes beyond shared equalities: their technique uses a common predicate language in which shared facts can be quantified predicates expressed in first-order logic with transitive closure.

All of the work mentioned above has the common goal that the combined domain should be more than the sum of it parts—i.e., it should be able infer facts that the individual domains could not infer alone. Our work has the unique characteristic that the auxiliary domain $\mathcal{P}$ is introduced for the purpose of increasing the precision of computing $\widetilde{\text{Assume}}[\varphi](A)$. This aspect resembles the "instrumentation predicates" used to control the precision of abstract domains for shape-analysis [37]. However, $\mathcal{P}$'s domain of discourse is the structure of the formula on which $\widetilde{\text{Assume}}$ is performed, rather than another "take" on what subset of the concrete domain $\mathcal{C}$ is being represented. Our use of $\mathcal{P}$ was inspired by the similar component used in Stålmarck's method (in the simpler context of propositional-validity checking) to encode the structure of a formula.

References

1. T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In *TACAS*, pages 268–283, 2001.
2. E. Barrett and A. King. Range and set abstraction using SAT. *Electr. Notes Theor. Comp. Sci.*, 267(1), 2010.
3. M. Björk. First order Stålmarck. *J. Autom. Reasoning*, 42(1):99–122, 2009.
4. N. Bjørner and L. de Moura. Accelerated lemma learning using joins–DPLL(⊔). In *LPAR*, 2008.
5. J. Brauer and A. King. Automatic abstraction for intervals using Boolean formulae. In *SAS*, 2010.
6. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *FMSD*, 25(2–3), 2004.
7. 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.
8. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.
9. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the ASTRÉE static analyzer. In *ASIAN*, 2006.
10. P. Cousot, R. Cousot, and L. Mauborgne. Logical abstract domains and interpretations. In *The Future of Software Engineering*, 2011.
11. P. Cousot and N. Halbwachs. Automatic discovery of linear constraints among variables of a program. In *POPL*, pages 84–96, 1978.
12. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
13. V. D'Silva, L. Haller, D. Kroening, and M. Tautschnig. Numeric bounds analysis with conflict-driven learning. In *TACAS*, 2012. To appear.
14. M. Elder, J. Lim, T. Sharma, T. Andersen, and T. Reps. Abstract domains of affine relations. In *SAS*, 2011.
15. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, 1997.
16. S. Graham and M. Wegman. A fast and usually linear algorithm for data flow analysis. *J. ACM*, 23(1):172–202, 1976.
17. P. Granger. Improving the results of static analyses programs by local decreasing iteration. In *FSTTCS*, 1992.
18. S. Gulwani and A. Tiwari. Combining abstract interpreters. In *PLDI*, 2006.
19. L. Haller. Satisfiability solving as abstract interpretation, 2011. Talk presented at NSAD 2011 (Sept. 13, 2011).

20. J. Harrison. Stålmarck's algorithm as a HOL Derived Rule. In *TPHOLs*, 1996.
21. M. Karr. Affine relationship among variables of a program. *Acta Inf.*, 6, 1976.
22. N. Kidd, A. Lal, and T. Reps. WALi: The Weighted Automaton Library, 2007. www.cs.wisc.edu/wpis/wpds/download.php.
23. A. King and H. Søndergaard. Automatic abstraction for congruences. In *VMCAI*, 2010.
24. J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *CC*, 1992.
25. A. Lal and T. Reps. Improving pushdown system model checking. In *CAV*, 2006.
26. A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *CAV*, 2005.
27. B. McCloskey, T. Reps, and M. Sagiv. Statically inferring complex heap, array, and numeric invariants. In *SAS*, 2010.
28. K. McMillan, A. Kuehlmann, and M. Sagiv. Generalizing DPLL to richer logics. In *CAV*, 2009.
29. A. Miné. The octagon abstract domain. In *WCRE*, 2001.
30. D. Monniaux. Automatic modular abstractions for template numerical constraints. *LMCS*, 6(3), 2010.
31. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535, 2001.
32. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. *TOPLAS*, 2007.
33. G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *TOPLAS*, 1(2), 1979.
34. PPL: The Parma polyhedra library. www.cs.unipr.it/ppl/.
35. J. Regehr and A. Reid. HOIST: A system for automatically deriving static analyzers for embedded systems. In *Arch. Support for Program. Langs. and Oper. Syst.*, 2004.
36. T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, pages 252–266, 2004.
37. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.
38. S. Sankaranarayanan, H. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI*, 2005.
39. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.
40. M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. *FMSD*, 16(1), 2000.
41. A. Thakur and T. Reps. A generalization of Stålmarck's method. TR 1699, CS Dept., Univ. of Wisconsin, Madison, WI, Oct. 2011.
42. A. Thakur and T. Reps. A method for symbolic computation of precise abstract transformers. TR 1702, CS Dept., Univ. of Wisconsin, Madison, WI, Oct. 2011.
43. G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS*, pages 530–545, 2004.