A Method for Symbolic Computation of Precise Abstract Transformers

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 best transformers. In fact, algorithms are known for only a few abstract domains.

This paper presents a parametric framework that, for some abstract domains, is capable of obtaining best transformers in the limit. Because the method approaches best transformers from "above", if the computation takes too much time it can be stopped to yield a sound abstract transformer. Thus, the framework provides a tunable algorithm that offers a performance-versus-precision trade-off.

We describe instantiations of the framework for well-known abstract domains, such as intervals, polyhedra, and Cartesian predicate abstraction. We also show that the framework applies to several new variants of predicate-abstraction domains that we define in the paper.

1 Introduction

Abstract interpretation [8] 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, in the limit. While at one level the work concerns fundamental limits, the algorithms that we present are also likely to perform well in practice. While we do not yet have experimental results to report, the algorithms adopt (and adapt) principles that are known to perform well in a somewhat different context [35].

Cousot and Cousot [10] gave a *specification* of the most-precise abstract interpretation of a concrete operation that is possible in a given abstract domain. Their conditions are quite general, and apply to most program-analysis situations. Suppose that $\mathcal{G} = \mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$ is a Galois connection defined by abstraction function $\alpha : \mathcal{C} \rightarrow \mathcal{A}$ and concretization function $\gamma : \mathcal{A} \rightarrow \mathcal{C}$. Given $\mathcal{G}$, the "best transformer", or best abstract post operator for transition τ, denoted by $\widehat{\text{Post}}[\tau] : \mathcal{A} \rightarrow \mathcal{A}$, is the most precise abstract operator possible for the concrete post operator for τ, $\text{Post}[\tau] : \mathcal{C} \rightarrow \mathcal{C}$. $\widehat{\text{Post}}[\tau]$ can be expressed in terms of α, γ, and $\text{Post}[\tau]$, as follows [10]:

$$\widehat{\text{Post}}[\tau] = \alpha \circ \text{Post}[\tau] \circ \gamma. \tag{1}$$

Eqn. (1) defines the limit of precision obtainable using abstraction $\mathcal{A}$. However, Eqn. (1) 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. $\underline{\widehat{\alpha}(\varphi)}$: Given a formula φ in some 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 $[\![\varphi]\!]_{\mathcal{L}}$ [30]. In particular, $\widehat{\alpha}(\varphi) = \alpha([\![\varphi]\!]_{\mathcal{L}})$.
2. $\underline{\widehat{\text{Assume}}[\varphi](A)}$: 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, $\widehat{\text{Assume}}[\varphi](A)$ is the best value that over-approximates $[\![\varphi]\!] \cap \gamma(A)$.
3. Creation of a representation of $\widehat{\text{Post}}[\tau]$: Some intraprocedural dataflow-analysis algorithms [16], and many interprocedural dataflow-analysis algorithms [9, 34, 23, 29, 5, 31], 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 expressed in terms of $\widehat{\text{Assume}}$ (item 2) as follows:

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

To describe how item 3 is related to items 1 and 2, we need the concept of a *vocabulary*. A vocabulary consists of the set of program variables, constant symbols, predicate symbols, and function symbols that are used to describe program states. A *one-vocabulary* formula denotes a set of states. A *two-vocabulary* formula denotes a state transformer—i.e., a relation between states described by the *pre-state* vocabulary and states described by the *post-state* vocabulary.

The concrete post operator $\text{Post}[\tau]$ is a two-vocabulary formula. For item 3, the instances of abstract datatype $\mathcal{T}$ represent elements of a two-vocabulary abstract domain. In this case, we can create the best instance in $\mathcal{T}$ that over-approximates $\text{Post}[\tau]$ by performing $\widehat{\alpha}(\text{Post}[\tau])$.

Because of these relationships, even though the paper's title refers to "precise abstract transformers", we primarily concentrate on $\widehat{\text{Assume}}[\varphi](A)$ in the next few sections. We return to the subject of precise abstract transformers in §5.

Heretofore, algorithms for performing best abstract operations have been known for only a few abstract domains [15, 30, 37, 21, 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) [21, 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 ∼325x 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 sound result—i.e., an over-approximation to the best abstract operation. 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] Thus, the framework provides a tunable algorithm that offers a performance-versus-precision trade-off.

Key Insight. In [36], we showed how Stålmarck's method [35], an algorithm for validity checking of propositional formulas,[4] can be explained using abstract-interpretation terminology—in particular, as an instantiation of a more general algorithm, Stålmarck[$\mathcal{A}$], which 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 particular instantiation of Stålmarck[$\mathcal{A}$] with a certain abstract domain.

The key insight of the present paper is that there is a connection between Stålmarck[$\mathcal{A}$] on the one hand and $\widehat{\alpha}_{\mathcal{A}}$ and $\widehat{\mathrm{Assume}}_{\mathcal{A}}$ on the other. In particular, to check whether a formula φ is valid, Stålmarck[$\mathcal{A}$] performs a sound test of whether φ is falsifiable by checking whether $\widehat{\alpha}_{\mathcal{A}}(\neg\varphi)$ equals $\bot_{\mathcal{A}}$. If the test succeeds, that 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, and assumes that we have a more expressive logic $\mathcal{L}$, such as quantifier-free linear arithmetic or quantifier-free bit-vector arithmetic. Because we are working with more expressive logics, our algorithm uses several ideas that go beyond what is used in Stålmarck's method [35] as well as what is used in Stålmarck[$\mathcal{A}$] [36].

Stålmarck's method is structured as a semi-decision procedure for validity checking. The actions of the algorithm are parameterized by a certain parameter k that is fixed by the user. For a given tautology, if k is large enough Stålmarck's method can prove validity, but if k is too small the answer returned is "unknown". In the latter case, one can increment k and try again; however, the bigger the k, the more expensive it is to run Stålmarck's method.

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

[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.

[4] A validity checker (also known as a tautology checker) determines whether a given formula φ over the propositional variables $\{p_i\}$ is true for all assignments of truth values to $\{p_i\}$. Validity checking is dual to satisfiability checking; validity of φ can be checked using a SAT solver by checking whether $\neg\varphi$ is satisfiable and complementing the answer: VALIDITY(φ) = ¬SAT(¬φ).

on parameter k), and the operation performed at each recursive level is the slightly more general operation $\widehat{\text{Assume}}[\psi](A)$. Consequently, the remainder of the paper mainly concerns the operation $\widehat{\text{Assume}}[\psi](A)$.

The method presented in this paper holds the promise of providing improved technology for performing fundamental operations in abstract interpretation.

- At each step, the algorithm holds an over-approximation to the most-precise answer. The advantage of this design is that the algorithm can be stopped at any point.
- In practice, Stålmarck's method succeeds in proving most propositional-logic tautologies using only $k = 1$. The hope is that this property carries over to the $\widetilde{\alpha}$, $\widetilde{\text{Assume}}$, and $\widetilde{\text{Post}}[\tau]$ algorithms developed in this paper. If it does, the procedures for $\widetilde{\alpha}$, $\widetilde{\text{Assume}}$, and $\widetilde{\text{Post}}[\tau]$ would nearly always attain the precision of $\widehat{\alpha}$, $\widehat{\text{Assume}}$, and $\widehat{\text{Post}}[\tau]$, respectively.

Contributions. The contributions of the paper can be summarized as follows:

- We present a new parametric framework that, for some abstract domains, is capable of performing most-precise abstract operations in the limit, including $\widehat{\alpha}(\varphi)$ and $\widehat{\text{Assume}}[\varphi](A)$, as well as creating a representation of $\widehat{\text{Post}}[\tau]$. Because the method approaches most-precise values from "above", if the computation takes too much time it can be stopped to yield a sound result.
- We present new algorithms for well-known domains, such as Cartesian predicate abstraction, intervals, and polyhedra [13].
- We show how our framework applies to several new domains that are "not much harder" than standard predicate abstraction. That is, they are all based on a fixed collection of closed state predicates and use mostly the same machinery as standard predicate abstraction. These domains are able to represent richer relationships between the state predicates by using 2-variable Boolean affine relations (equivalences), k-variable Boolean affine relations, and 2-variable Boolean inequalities.

In [36], we showed how abstract interpretation can provide insight on decision procedures for propositional logic. In the present paper, we describe new ways in which ideas from decision procedures 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 various instantiations of the framework. §5 presents several additional aspects of our work. §6 discusses related work. Proofs can be found in App. A.

2 Overview

In this section, we illustrate our algorithm for computing $\widetilde{\text{Assume}}[\varphi](A)$, where φ is a formula and A is a value in abstract domain $\mathcal{A}$ of intervals.

Example 1. Consider $\varphi = (x \geq 4 \wedge (x = 2 \vee y < 2x+5))$ and the abstract domain $\mathcal{A}$ in which each element A is a pair (A_x, A_y), where A_x and A_y are intervals that over-approximate the values of x and y, respectively. Let the initial abstract value $A_0 \in \mathcal{A}$ be $(\top, \top)$. Suppose that we want to compute $\widetilde{\text{Assume}}[\varphi](A_0)$. The first step is to associate a fresh Boolean variable with each subformula of φ. This step gives us a set of *integrity constraints* $\mathcal{I} = \{u_1 \Leftrightarrow (u_2 \wedge u_3), u_2 \Leftrightarrow (x \geq 4), u_3 \Leftrightarrow (u_4 \vee u_5), u_4 \Leftrightarrow (x = 2), u_5 \Leftrightarrow (y < 2x + 5)\}$. 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 φ via the set of Boolean variables $\mathcal{U} = \{u_1, u_2, \ldots, u_5\}$. We now introduce an abstraction over $\mathcal{U}$; in particular, we use the Cartesian domain $\mathcal{P}$ in which each element represents a set of assignments in $\mathbb{P}(\mathcal{U} \to \{0,1\})$. We will 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.

Let $P_0 \in \mathcal{P}$ be the abstract value in which the value of u_1 is 1, i.e., $P_0 = \top_{\mathcal{P}}[u_1 \mapsto 1]$, which corresponds to φ being satisfiable. In fact, the models of φ are closely related to the concrete values in $[\![\mathcal{I}]\!] \cap \gamma(P_0)$. For every concrete value in $[\![\mathcal{I}]\!] \cap \gamma(P_0)$, its projection onto x and y gives us a model of φ; that is, $[\![\varphi]\!] = ([\![\mathcal{I}]\!] \cap \gamma(P_0)) \downarrow_{\{x,y\}}$. By this means, we reduce the problem of computing $\widetilde{\text{Assume}}[\varphi](A_0)$ to that of computing $\widetilde{\text{Assume}}[\mathcal{I}]((P_0, A_0))$, where (P_0, A_0) is an element of the cross product of $\mathcal{P}$ and $\mathcal{A}$.

Given an abstract value (P, A) and integrity constraints $\mathcal{I}$, we use $\gamma_{\mathcal{I}}((P, A))$ to mean $[\![I]\!] \cap \gamma((P, A))$. An abstract value (P', A') is said be a *semantic reduction* [10] *of* (P, A) *with respect to* $\mathcal{I}$ if (i) $\gamma_{\mathcal{I}}((P', A')) = \gamma_{\mathcal{I}}((P, A))$, and (ii) $(P', A') \sqsubseteq (P, A)$.

The basic means of computing $\widetilde{\text{Assume}}$ is by means of *propagation rules*, which produce a semantic reduction of the current abstract value (P, A) with respect to the integrity constraints $\mathcal{I}$. The main feature of the propagation rules is that they are "local"; that is, they make use of only a single integrity constraint in $\mathcal{I}$, and a small constraint on the current abstract value (P, A). Given integrity constraints $\mathcal{I}$ and abstract value (P, A), a propagation rule has the form

$$\textsc{PropRule}\ \frac{J \in \mathcal{I} \qquad (P_1, A_1) \sqsupseteq (P, A) \qquad \ldots}{(P_2, A_2)}$$

A propagation rule must have the property that if the premise holds, then (P_2, A_2) is a semantic reduction of (P_1, A_1) with respect to J. A propagation rule enables information to be exchanged between the component abstract values P_1 and A_1 when computing (P_2, A_2). After deriving (P_2, A_2), we perform

the assignment $(P, A) := (P_2, A_2) \sqcap (P, A)$, which gives us a semantic reduction of (P, A).

For our example, the propagation rules work as follows:

1. The initial abstract value is $(P, A) = (P_0, A_0) = ((1, *, *, *, *), (\top, \top))$.
2. Because $u_1 \Leftrightarrow (u_2 \wedge u_3) \in \mathcal{I}$ and the value of u_1 is 1 in P, both u_2 and u_3 must be 1. This propagation rule can be stated as

$$\frac{\begin{array}{c} J = (u_1 \Leftrightarrow (u_2 \wedge u_3)) \in \mathcal{I} \\ (P_1, A_1) = ((1, *, *, *, *), (\top, \top)) \sqsupseteq ((1, *, *, *, *), (\top, \top)) \qquad P_1(u_1) = 1 \end{array}}{(P_2, A_2) = ((1, 1, 1, *, *), (\top, \top))}$$

   Thus, (P, A) becomes $((1, 1, 1, *, *), (\top, \top)) \sqcap ((1, *, *, *, *), (\top, \top)) = ((1, 1, 1, *, *), (\top, \top))$.
3. Because $u_2 \Leftrightarrow (x \geq 4) \in \mathcal{I}$ and the value of u_2 is 1 in P, $x \geq 4$ must be true in A. Let A' be the result of a "micro-$\widetilde{\text{Assume}}$" that over-approximates the inequation $x \geq 4$ in $(\top, \top)$, denoted by $A' = \mu\widetilde{\text{Assume}}[x \geq 4]((\top, \top))$. In this case, A' is $([4, \infty], \top)$. This propagation rule can be stated as

$$\frac{\begin{array}{c} J = (u_2 \Leftrightarrow (x \geq 4)) \in \mathcal{I} \\ (P_1, A_1) = ((*, 1, *, *, *), (\top, \top)) \sqsupseteq ((1, 1, 1, *, *), (\top, \top)) \\ P_1(u_2) = 1 \qquad ([4, \infty], \top) = \mu\widetilde{\text{Assume}}[x \geq 4]((\top, \top)) \end{array}}{(P_2, A_2) = ((*, 1, *, *, *), ([4, \infty], \top))}$$

   Thus, (P, A) becomes $((*, 1, *, *, *), ([4, \infty], \top)) \sqcap ((1, 1, 1, *, *), (\top, \top)) = ((1, 1, 1, *, *), ([4, \infty], \top))$.
4. Because $x \in [4, \infty]$ in A implies $x \neq 2$, and we have $u_4 \Leftrightarrow (x = 2)$ in $\mathcal{I}$, the value of u_4 must be 0. This propagation rule can be stated as

$$\frac{\begin{array}{c} J = (u_4 \Leftrightarrow (x = 2)) \in \mathcal{I} \\ (P_1, A_1) = ((*, *, *, *, *), ([4, \infty], \top)) \sqsupseteq ((1, 1, 1, *, *), ([4, \infty], \top)) \\ (x \mapsto [4, \infty]) \Rightarrow (x \neq 2) \end{array}}{(P_2, A_2) = ((*, *, *, 0, *), ([4, \infty], \top))}$$

   Thus, (P, A) becomes $((*, *, *, 0, *), ([4, \infty], \top)) \sqcap ((1, 1, 1, *, *), ([4, \infty], \top)) = ((1, 1, 1, 0, *), ([4, \infty], \top))$.
5. Because $u_3 \Leftrightarrow (u_4 \vee u_5) \in \mathcal{I}$ and the values of u_3 and u_4 are 1 and 0, respectively, the value of u_5 must be 1. This propagation rule can be stated as

$$\frac{\begin{array}{c} J = (u_3 \Leftrightarrow (u_4 \vee u_5)) \in \mathcal{I} \\ (P_1, A_1) = ((*, *, 1, 0, *), (\top, \top)) \sqsupseteq ((1, 1, 1, 0, *), ([4, \infty], \top)) \\ P_1(u_3) = 1 \qquad P_1(u_4) = 0 \end{array}}{(P_2, A_2) = ((*, *, 1, 0, 1), (\top, \top))}$$

   Thus, (P, A) becomes $((*, *, 1, 0, 1), (\top, \top)) \sqcap ((1, 1, 1, 0, *), ([4, \infty], \top)) = ((1, 1, 1, 0, 1), ([4, \infty], \top))$.

$$
\begin{aligned}
&((1,*,*,*,*),(\top,\top))\\
\sqsubseteq\ &((1,1,1,*,*),(\top,\top))\\
\sqsubseteq\ &((1,1,1,*,*),([4,\infty],\top))\\
\sqsubseteq\ &((1,1,1,0,*),([4,\infty],\top))\\
\sqsubseteq\ &((1,1,1,0,1),([4,\infty],\top))\\
\sqsubseteq\ &((1,1,1,0,1),([4,\infty],[-\infty,12]))
\end{aligned}
$$

Fig. 1. The sequence of semantic reductions carried out by the propagation rules to compute $\widetilde{\text{Assume}}$ for Ex. 1.

6. Because $u_5 \Leftrightarrow (y < 2x+5) \in \mathcal{I}$ and u_5 is 1 in P, $y < 2x+5$ must be true. Moreover, $A_x = [4,\infty]$. Let A' be the result of $\mu\widetilde{\text{Assume}}[y < 2x+5]((A_x,\top))$, which over-approximates the inequation $y < 2x+5$ in $(A_x,\top)$. In this case, A' is $([4,\infty],[-\infty,12])$. This propagation rule can be stated as

$$
\frac{\begin{array}{c} J = (u_5 \Leftrightarrow (y < 2x+5)) \in \mathcal{I} \\ (P_1,A_1) = ((*,*,*,*,1),([4,\infty],\top)) \sqsupseteq ((1,1,1,0,1),([4,\infty],\top)) \\ P_1(u_5) = 1 \qquad ([4,\infty],[-\infty,12]) = \mu\widetilde{\text{Assume}}[y < 2x+5](([4,\infty],\top)) \end{array}}{(P_2,A_2) = ((*,*,*,*,1),([4,\infty],[-\infty,12]))}
$$

Thus, (P,A) becomes $((*,*,*,*,1),([4,\infty],[-\infty,12])) \sqcap ((1,1,1,0,1),([4,\infty],\top)) = ((1,1,1,0,1),([4,\infty],[-\infty,12]))$.

The process of repeatedly applying propagation rules to compute $\widetilde{\text{Assume}}$ is called 0-assume. As summarized in Fig. 1, we obtain $\widetilde{\text{Assume}}[\mathcal{I}](((1,*,*,*,*),(\top,\top))) = ((1,1,1,0,1),([4,\infty],[-\infty,12]))$ by 0-assume, and hence $\widetilde{\text{Assume}}[\varphi]((\top,\top)) = ([4,\infty],[-\infty,12])$. □

Example 2. Unfortunately, 0-assume is not always sufficient to obtain a good answer for $\widetilde{\text{Assume}}$. Consider $\psi = z = y + x \wedge ((x = w \wedge y = 3) \vee (x = 3 \wedge y = 2))$. The abstract domain $\mathcal{A}$ in this example is a tuple of intervals representing w, x, y, and z respectively. Let $A_0 = ([2,2],\top,\top,\top)$; that is, the abstract value that represents all concrete values in which w is 2. Suppose that we wish to compute $\widetilde{\text{Assume}}[\psi](A_0)$.

The integrity constraints $\mathcal{I}$ corresponding to ψ are

$$
\begin{array}{lll}
u_1 \Leftrightarrow (u_2 \wedge u_3) & u_2 \Leftrightarrow (z = y + x) & u_3 \Leftrightarrow (u_4 \vee u_5) \\
u_4 \Leftrightarrow (u_6 \wedge u_7) & u_5 \Leftrightarrow (u_8 \wedge u_9) & u_6 \Leftrightarrow (x = w) \\
u_7 \Leftrightarrow (y = 3) & u_8 \Leftrightarrow (x = 3) & u_9 \Leftrightarrow (y = 2)
\end{array}
$$

As in the previous example, the variables $\mathcal{U} = \{u_1, u_2, \ldots, u_9\}$ are introduced during the conversion to $\mathcal{I}$, with u_1 as the root variable for ψ. We use the Cartesian domain $\mathcal{P}$ over $\mathcal{U}$ to represent assignments to variables in $\mathcal{U}$. Let $P_0 = (1,*,*,*,*,*,*,*,*)$ represent the abstract value corresponding to ψ being satisfiable. We compute $\widetilde{\text{Assume}}[\psi](A_0)$ by computing $\widetilde{\text{Assume}}[\mathcal{I}]((P_0,A_0))$. Let us perform 0-assume with respect to the abstract value (P_0,A_0). Because $u_1 \Leftrightarrow (u_2 \wedge u_3) \in \mathcal{I}$ and u_1 is 1 in p, u_2 and u_3 must be 1. This gives us the new abstract value $(P_1,A_1) = ((1,1,1,*,*,*,*,*,*),([2,2],\top,\top,\top))$; however, no more propagation rules apply and 0-assume terminates.

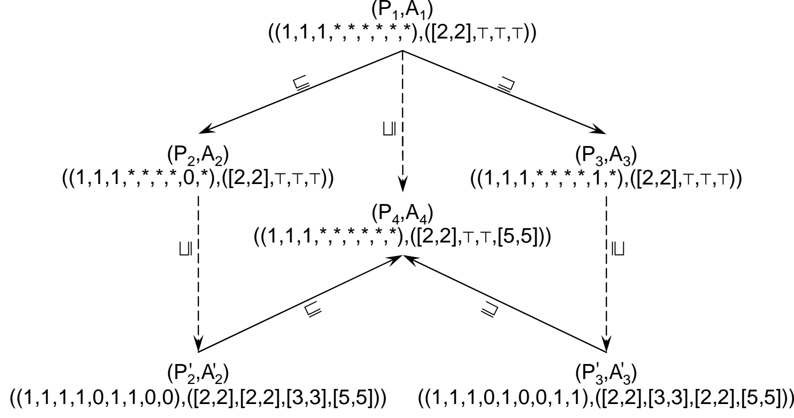

Fig. 2. The application of the Dilemma Rule to abstract value (P_1, A_1) of Ex. 2. We indicate that (P_i, A_i) is a semantic reduction of (P_j, A_j) by a dashed arrow from (P_j, A_j) to (P_i, A_i). Overall, (P_4, A_4) is a semantic reduction of (P_1, A_1).

To increase precision, we need to use the Dilemma Rule, which is a special type of branching and merging rule, shown schematically in Fig. 2. Let $B \in \mathcal{P}$ be $\top_{\mathcal{P}}[u_8 \mapsto 0]$; then $\overline{B}$, the abstract complement of B, is $\top_{\mathcal{P}}[u_8 \mapsto 1]$. Note that $\gamma(B) \cap \gamma(\overline{B}) = \emptyset$, and $\gamma(B) \cup \gamma(\overline{B}) = \gamma(\top)$. We split the current abstract value (P_1, A_1) into two values:

1. $(P_2, A_2) = (P_1, A_1) \sqcap (B, \top) = ((1, 1, 1, *, *, *, *, 0, *), ([2, 2], \top, \top, \top))$, and
2. $(P_3, A_3) = (P_1, A_1) \sqcap (\overline{B}, \top) = ((1, 1, 1, *, *, *, *, 1, *), ([2, 2], \top, \top, \top))$.

Performing 0-assume on (P_2, A_2) gives us the abstract value $(P'_2, A'_2) = ((1, 1, 1, 1, 0, 1, 1, 0, 0), ([2, 2], [2, 2], [3, 3], [5, 5]))$, as shown in Fig. 3, and performing 0-assume on (P_3, A_3) gives us the abstract value $(P'_3, A'_3) = ((1, 1, 1, 0, 1, 0, 0, 1, 1), ([2, 2], [3, 3], [2, 2], [5, 5]))$, as shown in Fig. 4. The abstract values (P'_2, A'_2) and (P'_3, A'_3) are merged by performing a join to give the abstract value $(P_4, A_4) = (P'_2, A'_2) \sqcup (P'_3, A'_3) = ((1, 1, 1, *, *, *, *, *, *), ([2, 2], \top, \top, [5, 5]))$. Note that $(P_4, A_4) \sqsubset (P_1, A_1)$.

$\widetilde{\text{Assume}}[\mathcal{I}]((P_0, A_0)) = (P_4, A_4) = ((1, 1, 1, *, *, *, *, *, *), ([2, 2], \top, \top, [5, 5]))$, and $\widetilde{\text{Assume}}[\psi](A_0) = A_4 = ([2, 2], \top, \top, [5, 5])$. □

The process of repeatedly applying the Dilemma Rule is called 1-assume; that is, for every variable $u \in \mathcal{U}$, the current abstract value is split into two

$(P_2, A_2) =$	
$((1, 1, 1, *, *, *, *, 0, *), ([2, 2], \top, \top, \top))$	$(P_0, A_0) \sqcap (B, \top)$
$((1, 1, 1, *, 0, *, *, 0, *), ([2, 2], \top, \top, \top))$	$u_5 \Leftrightarrow (u_8 \wedge u_9), u_8 = 0$ implies $u_5 = 0$
$((1, 1, 1, 1, 0, *, *, 0, *), ([2, 2], \top, \top, \top))$	$u_3 \Leftrightarrow (u_4 \vee u_5), u_3 = 1, u_5 = 0,$ implies $u_4 = 1$
$((1, 1, 1, 1, 0, 1, 1, 0, *), ([2, 2], \top, \top, \top))$	$u_4 \Leftrightarrow (u_6 \wedge u_7), u_4 = 1$ implies $u_6 = 1, u_7 = 1$
$((1, 1, 1, 1, 0, 1, 1, 0, *), ([2, 2], \top, [3, 3], \top))$	$u_7 \Leftrightarrow (y = 3), u_7 = 1,$ implies $y \mapsto [3, 3]$
$((1, 1, 1, 1, 0, 1, 1, 0, 0), ([2, 2], \top, [3, 3], \top))$	$u_9 \Leftrightarrow (y = 2), y \mapsto [3, 3]$ implies $u_9 = 0$
$((1, 1, 1, 1, 0, 1, 1, 0, 0), ([2, 2], [2, 2], [3, 3], \top))$	$u_6 \Leftrightarrow (x = w), u_6 = 1, w \mapsto [2, 2]$ implies $x \mapsto [2, 2]$
$((1, 1, 1, 1, 0, 1, 1, 0, 0), ([2, 2], [2, 2], [3, 3], [5, 5]))$	$u_2 \Leftrightarrow (z = y + x), u_2 = 1, y \mapsto [3, 3], x \mapsto [2, 2]$
$= (P'_2, A'_2)$	implies $z \mapsto [5, 5]$

Fig. 3. Propagation rules applied to abstract value (P_2, A_2) of Ex. 2.

$(P_3, A_3) =$	
$(1,1,1,*,*,*,*,1,*), ([2,2],\top,\top,\top)$	$(P_0, A_0) \sqcap (\overline{B}, \top)$
$(1,1,1,*,*,*,*,1,*), ([2,2],[3,3],\top,\top)$	$u_8 \Leftrightarrow (x = 3), u_8 = 1,$ implies $x \mapsto [3,3]$
$(1,1,1,*,*,0,*,1,*), ([2,2],[3,3],\top,\top)$	$u_6 \Leftrightarrow (x = w), w \mapsto [2,2], x \mapsto [3,3]$ implies $u_6 = 0$
$(1,1,1,0,*,0,*,1,*), ([2,2],[3,3],\top,\top)$	$u_4 \Leftrightarrow (u_6 \wedge u_7), u_6 = 0$ implies $u_4 = 0$
$(1,1,1,0,1,0,*,1,*), ([2,2],[3,3],\top,\top)$	$u_3 \Leftrightarrow (u_4 \vee u_5), u_3 = 1, u_4 = 0$ implies $u_5 = 1$
$(1,1,1,0,1,0,*,1,1), ([2,2],[3,3],\top,\top)$	$u_5 \Leftrightarrow (u_8 \wedge u_9), u_5 = 1$ implies $u_9 = 1$
$(1,1,1,0,1,0,*,1,1), ([2,2],[3,3],[2,2],\top)$	$u_9 \Leftrightarrow (y = 2), u_9 = 1,$ implies $y \mapsto [2,2]$
$(1,1,1,0,1,0,0,1,1), ([2,2],[3,3],[2,2],\top)$	$u_7 \Leftrightarrow (y = 3), y \mapsto [2,2]$ implies $u_7 = 0$
$(1,1,1,0,1,0,0,1,1), ([2,2],[3,3],[2,2],[5,5])$	$u_2 \Leftrightarrow (y + z), y \mapsto [2,2], x \mapsto [3,3]$
$= (P'_3, A'_3)$	implies $z \mapsto [5,5]$

Fig. 4. Propagation rules applied to abstract value (P_3, A_3) of Ex. 2.

using $B = \top_{\mathcal{P}}[u \mapsto 0]$, 0-assume is applied to each of these values, and the resulting abstract values are merged via a join. The generalization of the 1-assume algorithm is called k-assume: for every variable $u \in \mathcal{U}$, the current abstract value is split into two using $B = \top_{\mathcal{P}}[u \mapsto 0]$; (k–1)-assume is applied to each of these values; and the resulting values are merged via join. There is a trade-off involved when increasing k: higher values of k give greater precision, but are also computationally more expensive.

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

In this section, we present the algorithm for computing $\widetilde{\mathrm{Assume}}[\varphi](A)$ in abstract domain $\mathcal{A}$. 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. There is a decidable logical language $\mathcal{L}$ in which all formulas of interest can be stated. Moreover, $\mathcal{A}$ supports *symbolic concretization* with respect to $\mathcal{L}$: The operation of symbolic concretization, denoted by $\widehat{\gamma}$, maps an abstract value $A \in \mathcal{A}$ to a formula $\widehat{\gamma}(A) \in \mathcal{L}$ such that A and $\widehat{\gamma}(A)$ represent the same set of concrete states (i.e., $\gamma(A) = [\![\widehat{\gamma}(A)]\!]$).
5. Given $A \in \mathcal{A}$ and an (in)equation of the form $x_1 \bowtie F(x_1, \ldots, x_n)$, where $\bowtie$ stands for a single relational symbol of $\mathcal{L}$, such as $=$, $\neq$, $<$, $\leq$, $>$, $\geq$, or the arithmetic-congruence symbol $\equiv$, there is an algorithm to compute the "micro-$\widetilde{\mathrm{Assume}}$" $A' = \mu\widetilde{\mathrm{Assume}}[x_1 \bowtie F(x_1, \ldots, x_n)](A)$, which over-approximates the (in)equation $x_1 \bowtie F(x_1, \ldots, x_n)$ in A.

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

Alg. 1 presents the algorithm that computes $\widetilde{\mathrm{Assume}}[\varphi](A)$ 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. 5. u_φ is assigned to formula φ. We use $\mathcal{U}$ to

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

Fig. 5. 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.

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

1 $\langle \mathcal{I}, u_\varphi \rangle \leftarrow \text{integrity}(\varphi)$
2 $P \leftarrow \top_{\mathcal{P}}[u_\varphi \mapsto 1]$
3 $(\widetilde{P}, \widetilde{A}) \leftarrow \text{k-assume}[\mathcal{I}]((P, A))$
4 **return** $\widetilde{A}$

denote the set of fresh variables created when converting φ to $\mathcal{I}$. Alg. 1 also uses a second abstract domain, $\mathcal{P}$, each of whose elements represents a set of Boolean assignments in $\mathbb{P}(\mathcal{U} \to \{0, 1\})$. The requirements on $\mathcal{P}$ are:

1. $\top_{\mathcal{P}}[u_\varphi \mapsto 1]$ is an element of $\mathcal{P}$.
2. $\mathcal{P}$ possesses a set of *complementable* "basis" elements $\mathcal{B}_{\mathcal{P}}$. That is, (1) for each element $B \in \mathcal{B}_{\mathcal{P}}$, there exists $B' \in \mathcal{B}_{\mathcal{P}}$ such that $\gamma(B) \cap \gamma(B') = \emptyset$ and $\gamma(B) \cup \gamma(B') = \gamma(\top)$, and (2) for each element $P \in \mathcal{P}$, there exists $B \subseteq \mathcal{B}_{\mathcal{P}}$ such that $P = \bigsqcap B$.
3. There is an algorithm to perform meet with a basis element; that is, for all $P \in \mathcal{P}$ and each basis element $B \in \mathcal{P}$, there is an algorithm that produces $P \sqcap B$.

The Cartesian domain, which was used in §2, satisfies these requirements. In §5 we describe other domains that can also be used.

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 $k \geq 0$). The call to k-assume on line (3) returns $(\widetilde{P}, \widetilde{A})$, which is a semantic reduction of (P, A) with respect to $\mathcal{I}$; that is, $\gamma_{\mathcal{I}}((\widetilde{P}, \widetilde{A})) = \gamma_{\mathcal{I}}((P, A))$ and $(\widetilde{P}, \widetilde{A}) \sqsubseteq (P, A)$. In general, the greater the value of k, the more precise is the result computed by Alg. 1. The next theorem proves that Alg. 1 does indeed compute an over-approximation of Assume$[\varphi](A)$.

Theorem 1. *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. 2 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 in the inner loop body, lines (4)–(8), which implements an analog of the Dilemma Rule from Stålmarck's method [35].

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

```
1  repeat
2      (P', A') ← (P, A)
3      foreach basis element B ∈ B_P such that B ⋣ P and B̄ ⋣ P do
4          (P_1, A_1) ← (P, A) ⊓ (B, ⊤)                          // split
5          (P_2, A_2) ← (P, A) ⊓ (B̄, ⊤)                          // split
6          (P'_1, A'_1) ← (k–1)-assume[I]((P_1, A_1))             // recurse
7          (P'_2, A'_2) ← (k–1)-assume[I]((P_2, A_2))             // recurse
8          (P, A) ← (P'_1, A'_1) ⊔ (P'_2, A'_2)                   // merge
9  until ((P, A) = (P', A')) ∥ resource bound reached
10 return (P, A)
```

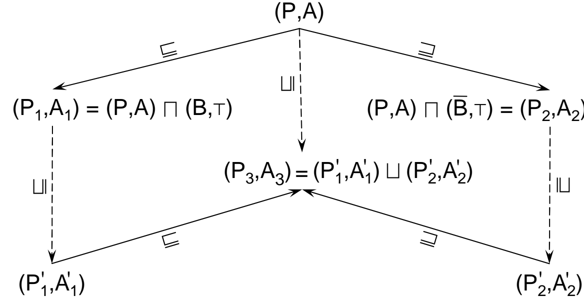

Fig. 6. The application of the Dilemma Rule to abstract value (P, A) using basis element $B \in \mathcal{B}_\mathcal{P}$. We indicate that (P'_i, A'_i) is a semantic reduction of (P_i, A_i) by a dashed arrow from (P_i, A_i) to (P'_i, A'_i).

The steps of the Dilemma Rule are shown schematically in Fig. 6: the abstract value (P, A) is split into two disjoint, refined abstract values, (P_1, A_1) and (P_2, A_2); semantic reductions (P'_1, A'_1) and (P'_2, A'_2) of (P_1, A_1) and (P_2, A_2) are computed; and finally (P'_1, A'_1) and (P'_2, A'_2) are merged by performing a join to give (P_3, A_3).

In particular, at line (3) of Alg. 2, a basis element $B \in \mathcal{B}_\mathcal{P}$ is chosen. B and its complement $\overline{B}$ are used to split the current abstract value (P, A) 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 (4) and (5). Because B is chosen in line (3) so that $B \not\sqsupseteq P$ and $\overline{B} \not\sqsupseteq P$, we are ensured that $(P_1, A_1) \neq \bot$ and $(P_2, A_2) \neq \bot$. By assumptions (2) and (3) about $\mathcal{P}$, we have $\gamma((P_1, A_1)) \cap \gamma((P_2, A_2)) = \emptyset$, and $\gamma((P_1, A_1)) \cup \gamma((P_2, A_2)) = \gamma(\top)$. Furthermore, $(P_1, A_1) \sqsubseteq (P, A)$, and $(P_2, A_2) \sqsubseteq (P, A)$.

The calls to (k–1)-assume at lines (6) and (7) 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 (8) (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. 6.)

The steps of the Dilemma Rule (Fig. 6) are repeated until a fixpoint is reached, or some resource bound is exceeded. The correctness of Alg. 2 is proved in the following theorem:

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

Algorithm 3: 0-assume$[\mathcal{I}]((P, A))$

```
1  repeat
2      (P', A') ← (P, A)
3      for some J ∈ I, (P1, A1) ⊒ (P, A) such that voc(J) ∩ voc((P1, A1)) ≠ ∅
       and |voc(J) ∪ voc((P1, A1))| < ε do
4          switch J do
5              case u ⇔ ℓ
6                  (P2, A2) ← ApplyLeafRule(J, (P1, A1))
7              otherwise
8                  (P2, A2) ← ApplyInternalRule(J, (P1, A1))
9          (P, A) = (P2, A2) ⊓ (P, A)
10 until ((P, A) = (P', A')) || resource bound reached
11 return (P, A)
```

$$\frac{J = (u \Leftrightarrow \ell) \in \mathcal{I} \qquad (P_1, A_1) \qquad \widehat{\gamma}(A_1) \Rightarrow \ell}{(P_1 \sqcap \top[u \mapsto 1], A_1)} \text{ AtoP-1}$$

$$\frac{J = (u \Leftrightarrow \ell) \in \mathcal{I} \qquad (P_1, A_1) \qquad \widehat{\gamma}(A_1) \Rightarrow \neg\ell}{(P_1 \sqcap \top[u \mapsto 0], A_1)} \text{ AtoP-0}$$

$$\frac{\begin{array}{c} J = (u \Leftrightarrow (x_1 \bowtie F(x_1, \ldots, x_n))) \in \mathcal{I} \\ (P_1, A_1) \qquad P_1(u) = 1 \qquad A_1' = \mu\widetilde{\text{Assume}}[x_1 \bowtie F(x_1, \ldots, x_n)](A_1) \end{array}}{(P_1, A_1')} \text{ PtoA-1}$$

$$\frac{\begin{array}{c} J = (u \Leftrightarrow (x_1 \bowtie F(x_1, \ldots, x_n))) \in \mathcal{I} \\ (P_1, A_1) \qquad P_1(u) = 0 \qquad A_1' = \mu\widetilde{\text{Assume}}[x_1 \overline{\bowtie} F(x_1, \ldots, x_n)](A_1) \end{array}}{(P_1, A_1')} \text{ PtoA-0}$$

Fig. 7. Rules used by Alg. 3 in the call ApplyLeafRule$(J, (P_1, A_1))$. $\bowtie$ is a relational symbol in $\mathcal{L}$, and $\overline{\bowtie}$ is the logical negation of the relational symbol $\bowtie$.

Alg. 3 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. 7

$$\frac{J = (u_1 \Leftrightarrow (u_2 \vee u_3)) \in \mathcal{I} \qquad (P_1, A_1) \qquad P_1(u_1) = 0}{(P_1 \sqcap \top[u_2 \mapsto 0, u_3 \mapsto 0], A_1)} \textsc{Or1}$$

$$\frac{J = (u_1 \Leftrightarrow (u_2 \wedge u_3)) \in \mathcal{I} \qquad (P_1, A_1) \qquad P_1(u_1) = 1}{(P_1 \sqcap \top[u_2 \mapsto 1, u_3 \mapsto 1], A_1)} \textsc{And1}$$

Fig. 8. Boolean rules used by Alg. 3 in the call ApplyInternalRule$(J, (P_1, A_1))$.

and 8. In lines (4)–(9), these rules are applied by using a single integrity constraint in $\mathcal{I}$ and part of the current abstract value (P, A); that is, on line (3) $J \in \mathcal{I}$ and $(P_1, A_1) \sqsupseteq (P, A)$ are chosen. Furthermore, the additional constraint that $|\text{voc}(J) \cup \text{voc}((P_1, A_1))| < \epsilon$, ensures that the application of the rules is not computationally expensive.

Given $J \in \mathcal{I}, (P_1, A_1) \sqsupseteq (P, A)$, the net effect of applying any of the propagation rules is to compute a semantic reduction of (P_1, A_1) with respect to $J \in \mathcal{I}$. The propagation rules used in Alg. 3 can be classified into two categories:

1. Rules that apply when J is of the form $u \Leftrightarrow \ell$, shown in Fig. 7. Note that the integrity constraint $u \Leftrightarrow \ell$ is generated from the leaves of the original formula φ. This category of rules can be further subdivided into:
   (a) Rules that compute a semantic reduction of P_1 with respect to J by propagating information from abstract value A_1 to abstract value P_1; viz., Rules AtoP-0 and AtoP-1. For instance, Rule AtoP-1 states that if $J = (u \Leftrightarrow \ell)$ and $\widehat{\gamma}(A_1) \Rightarrow \ell$, then we can infer that the Boolean variable u is true. Thus, the value of $P_1 \sqcap \top[u \mapsto 1]$ is a semantic reduction of P_1 with respect to J.
   (b) Rules that compute a semantic reduction of A_1 with respect to J by propagating information from abstract value P_1 to abstract value A_1; viz., Rules PtoA-0 and PtoA-1. For instance, the premise of Rule PtoA-1 requires J to be a literal of the form $u \Leftrightarrow (x_1 \bowtie F(x_1, \ldots, x_n))$ and Boolean variable u to have the value 1 in P_1. Furthermore, suppose that $A_1' \in \mathcal{A}$ is the the result of a "micro-$\widetilde{\text{Assume}}$", $A_1' = \mu\widetilde{\text{Assume}}[x_1 \bowtie F(x_1, \ldots, x_n)](A_1)$. Then A_1' is a semantic reduction of A_1 with respect to J.
2. Rules that apply when J is of the form $p \Leftrightarrow (q \text{ op } r)$, shown in Fig. 8. Such an integrity constraint is generated from an internal subformula of formula φ. These rules compute a non-trivial semantic reduction from P_1 with respect to J by only using information from P_1. For instance, rule Or1 says that if J is of the form $p \Leftrightarrow (q \text{ op } r)$, and p is 0 in P_1, then we can infer that both q and r must be 0. Thus, $P_1 \sqcap \top[q \mapsto 0, r \mapsto 0]$ is a semantic reduction of P_1.

Alg. 3 repeatedly applies the propagation rules until a fixpoint is reached, or some resource bound is reached. The next theorem proves that 0-assume does indeed compute a semantic reduction of (P, A) with respect to $\mathcal{I}$.

Theorem 3. *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 show how various well-known abstract domains, such as intervals [9], polyhedra [13], and Cartesian predicate abstraction [2], satisfy the requirements of our framework. All of these domains satisfy assumptions (1)–(3) about $\mathcal{A}$ from the beginning of §3. The decidable logic $\mathcal{L}$ needed for assumption (4) is that of quantifier-free first-order logic over linear arithmetic.

In the rest of this section, we describe for each of the domains the symbolic-concretization operation and the algorithm for computing "micro-$\widetilde{\text{Assumes}}$" of the form $\mu\widetilde{\text{Assume}}[x_1 \bowtie F(x_1, \ldots, x_n)](A)$, as required in assumption (5).

Interval Domain. The *interval domain* [9] over a set of variables $\mathcal{V} = \{v_i \mid 1 \leq i \leq n\}$ maintains an interval of integers $A_i = [l_i, h_i]$ for each variable v_i. Each element of the interval domain is a tuple $(A_1, A_2, \ldots, A_n)$. Let $\top[v \mapsto [l, h]]$ be the abstract value in which each variable $u \neq v$ has the value $[-\infty, \infty]$, and v has the value $[l, h]$. $\widehat{\gamma}(([l_1, h_1], \ldots, [l_n, h_n])) = \bigwedge_i (l_i \leq v_i \wedge v_i \leq h_i)$

Given an (in)equation $v_1 \bowtie F(v_1, \ldots, v_n)$ and an element of the interval domain A_1, Granger [17] describes a method to find an interval A_{v_1} that over-approximates the value of v_1. Then $\mu\widetilde{\text{Assume}}[v_1 \bowtie F(v_1, \ldots, v_n)](A_1) = A_1 \sqcap \top[v_1 \mapsto A_{v_1}]$.

Polyhedral Domain. Each element of the polyhedral domain can be represented as a set of linear inequality constraints [13]. Thus, $\widehat{\gamma}$ for any abstract value is the conjunction of the constraints corresponding to that abstract value.

Given an (in)equation $v_1 \bowtie F(v_1, \ldots, v_n)$, $\widehat{\alpha}(v_1 \bowtie F(v_1, \ldots, v_n))$ is a primitive provided by the polyhedral domain, as long as F is a linear function. Thus, given polyhedral-domain element A_1, $\mu\widetilde{\text{Assume}}[v_1 \bowtie F(v_1, \ldots, v_n)](A_1) = A_1 \sqcap \widehat{\alpha}(v_1 \bowtie F(v_1, \ldots, v_n))$.

Cartesian Predicate Abstraction Domain. The Cartesian predicate abstraction domain [2] is defined as a Cartesian domain over Boolean variables $\mathcal{W} = \{a_i\}$, in which each variable a_i is associated with a predicate $\varphi_i \in \mathcal{L}$. An element A of the Cartesian predicate abstraction domain represents a set of assignments in $\mathbb{P}(\mathcal{W} \to \{0, 1\})$. As we did for the Cartesian domain, we use $\top[a \mapsto b]$ to denote the abstract value in which all variables except a are mapped to $*$, and a has the value b. Then $\widehat{\gamma}(A) = \bigwedge_i (\varphi_i \Leftrightarrow a_i)$.

Given an abstract value A_1, let a_j be a variable such that $A_1(a_j) = *$. Given an (in)equation $v_1 \bowtie F(v_1, \ldots, v_n)$, we can check whether $(\widehat{\gamma}(A_1) \wedge (v_1 \bowtie F(v_1, \ldots, v_n))) \Rightarrow \varphi_j$; in which case $\mu\widetilde{\text{Assume}}[v_1 \bowtie F(v_1, \ldots, v_n)](A_1) = A_1 \sqcap \top[a_j \mapsto 1]$. Similarly, if $(\widehat{\gamma}(A_1) \wedge (v_1 \bowtie F(v_1, \ldots, v_n))) \Rightarrow \neg\varphi_j$, then $\mu\widetilde{\text{Assume}}[v_1 \bowtie F(v_1, \ldots, v_n)](A_1) = A_1 \sqcap \top[a_j \mapsto 0]$.

Enhanced Predicate Abstraction Domains. Cartesian predicate abstraction does not track any relations between the various predicates. We introduce variants of predicate-abstraction domains that are "not much harder" than

Cartesian predicate abstraction, but which track certain relations among the predicate variables $\mathcal{W} = \{a_i\}$.

- An element of the **2-BAR predicate abstraction domain** is a set of two-variable Boolean affine relations, each of the form $a_i \oplus a_j = \mathbf{0}$, $a_i \oplus a_j \oplus \mathbf{1} = \mathbf{0}$, $a_i = \mathbf{0}$, or $a_i \oplus \mathbf{1} = \mathbf{0}$, where $a_i, a_j \in \mathcal{W}$. In effect, this domain tracks equivalences between the Boolean variables. For instance, the relation $a_i \oplus a_j = \mathbf{0}$ corresponds to a_i being equivalent to a_j.
  $\widehat{\gamma}(A) = \bigwedge\{\varphi_i \Leftrightarrow \varphi_j \mid a_i \oplus a_j = \mathbf{0} \in A\} \wedge \bigwedge\{\varphi_i \Leftrightarrow \neg\varphi_j \mid a_i \oplus a_j = \mathbf{1} \in A\} \wedge \bigwedge\{\varphi_i \mid a_i = \mathbf{1} \in A\} \wedge \bigwedge\{\neg\varphi_i \mid a_i = \mathbf{0} \in A\}$.
  We can similarly define the **k-BAR predicate abstraction domain** as a set of k-variable Boolean affine relations.
- An element of the **2-BIR predicate abstraction domain** is a set of Boolean inequality relations of the form $a_i \leq a_j$, $a_j \leq b$, or $b \leq a_j$, where $a_i, a_j \in \mathcal{W}$ and $b \in \{\mathbf{0}, \mathbf{1}\}$.
  $\widehat{\gamma}(A) = \bigwedge\{\varphi_i \Rightarrow \varphi_j \mid a_i \leq a_j \in A\} \wedge \bigwedge\{\neg\varphi_i \mid a_i \leq \mathbf{0} \in A\} \wedge \bigwedge\{\varphi_j \mid \mathbf{1} \leq a_j \in A\}$.

For these new domains, $\mu\widetilde{\mathrm{Assume}}[v_1 \bowtie F(v_1, \ldots, v_n)](A_1)$ can be computed in a way similar to what is done for Cartesian predicate abstraction. Thus, our framework can be instantiated with these richer variants of predicate abstraction. These domains are strictly more expressive than Cartesian predicate abstraction. For instance, suppose that a_1 and a_2 are associated with the predicates $(x = 3)$ and $(y = 2)$, respectively, and φ is $(x = 4 \vee y = 2)$. With Cartesian predicate abstraction, $\widehat{\mathrm{Assume}}[\varphi](\top) = \top$. In contrast, with 2-BIR predicate abstraction, $\widehat{\mathrm{Assume}}[\varphi](\top) = \{a_1 \leq a_2\}$.

5 Discussion

Improving precision. So far, Alg. 2 performs splits in the Dilemma Rule using the basis elements of the Boolean abstraction domain $\mathcal{P}$. Suppose that the abstract domain $\mathcal{A}$ also possesses a set of complementable basis elements $\mathcal{B}_{\mathcal{A}}$. Given $\mathcal{B}_{\mathcal{A}}$, the split for the Dilemma Rule can also be performed using a basis element $B \in \mathcal{B}_{\mathcal{A}}$. This enhanced version of Alg. 2 can be more precise.

All of the abstract domains discussed in §4 have a complementable basis set.

- For any interval $[l_i, h_i]$, we have $[l_i, h_i] = [l_i, \infty] \sqcap [-\infty, h_i]$. Thus, $\mathcal{B}_{\mathcal{A}}$ is $\{\top[v_i \mapsto [-\infty, m] \mid m \in \mathbb{Z}, v_i \in \mathcal{V}]\} \cup \{\top[v_i \mapsto [m, \infty] \mid m \in \mathbb{Z}, v_i \in \mathcal{V}]\}$.
- For the polyhedral domain, the complementable basis elements are all the half-spaces of $\mathbb{R}^n$.
- For Cartesian predicate abstraction, the complementable basis elements are $\top[a_i \mapsto 0] \cup \top[a_i \mapsto 1]$.
- For the enhanced predicate-abstraction domains, the complementable basis elements are those that have a single relation.

Completeness. We say that the $\widetilde{\mathrm{Assume}}$ algorithm is complete for an abstract domain $\mathcal{A}$ if it is guaranteed to compute the best value $\widehat{\mathrm{Assume}}[\varphi](A)$. A sufficient condition for $\widetilde{\mathrm{Assume}}$ to be complete for abstract domain $\mathcal{A}$ is that $\mathcal{A}$ has a complementable basis set $\mathcal{B}_{\mathcal{A}}$ that is finite. In this case, given $\mathcal{B}_{\mathcal{A}}$, the split for

the Dilemma Rule can be performed with respect to a basis element $B \in \mathcal{B}_{\mathcal{A}}$, and Alg. 1 for $\widetilde{\text{Assume}}$ is able to compute $\widehat{\text{Assume}}$ using a sufficiently large value for k. In particular, $k = |\mathcal{B}_{\mathcal{P}} \cup \mathcal{B}_{\mathcal{A}}|$ suffices. The basis set $\mathcal{B}_{\mathcal{A}}$ is finite for the predicate-abstraction domains.

We are currently investigating completeness arguments for other abstract domains. For instance, we have been able to show that Alg. 1 is complete for polyhedra over rationals, when formulas are expressed in linear rational arithmetic and $k = |\mathcal{B}_{\mathcal{P}}|$. Interestingly, because the set of literals in the logic is closed under complementation, and both positive and negative literals are representable as polyhedra, this result holds for the version of Alg. 1 that does Dilemma-Rule splits only with respect to basis elements in $\mathcal{B}_{\mathcal{P}}$.

Enhanced Boolean Abstractions. Instead of using the Cartesian domain as the Boolean abstraction domain $\mathcal{P}$, we can also use the 2-BAR, k-BAR, and 2-BIR Boolean abstraction domains as $\mathcal{P}$. All of the domains satisfy assumptions (1)–(3) about $\mathcal{P}$ listed in §3. The variants of Alg. 1 instantiated with the enhanced Boolean abstraction domains are strictly more powerful than the version that uses the Cartesian domain. Further discussion of these domains can be found in [36].

Applying $\widehat{\text{Post}}[\tau]$. In §1, we described how $\widehat{\alpha}$ can be used to find a representation of $\widehat{\text{Post}}[\tau]$ in a two-vocabulary abstract domain by performing $\widehat{\alpha}(\text{Post}[\tau])$. However, in many intraprocedural dataflow analyses, one uses a one-vocabulary abstract domain, and the problem is how to apply $\widehat{\text{Post}}[\tau]$ to a specific abstract value A.

We typically distinguish the post-state vocabulary symbols by attaching primes ($'$); the pre-state symbols are unprimed. Similarly, we distinguish post-state abstract values by attaching primes, whereas pre-state abstract values are unprimed. The operation DropPrimes removes primes from primed symbols.

The concrete post operator $\text{Post}[\tau]$ is a two-vocabulary formula. Here we will write it as $\text{Post}[\tau(\overrightarrow{\text{Voc}}, \overrightarrow{\text{Voc}'})]$ to emphasize its two-vocabulary nature and the fact that each of the vocabularies usually consists of multiple symbols. With logics that support explicit quantification, a one-vocabulary, post-state (primed) formula ψ' that expresses the result of applying $\widehat{\text{Post}}[\tau]$ to abstract value A can be written as follows:

$$\psi' := \exists \overrightarrow{\text{Voc}}.(\widehat{\gamma}(A) \wedge \text{Post}[\tau(\overrightarrow{\text{Voc}}, \overrightarrow{\text{Voc}'})]).$$

We can then find the result of $\widehat{\text{Post}}[\tau](A)$ by performing $\text{DropPrimes}(\widehat{\alpha}(\psi'))$.

For logics that do not support explicit quantification, however, $\widehat{\text{Post}}[\tau](A)$ cannot be expressed symbolically without resorting to a more powerful logic. For instance, if sets of concrete states are represented using quantifier-free first-order logic formulas, one has to switch to quantified second-order logic to express ψ'.[5] In such situations, a variant of the procedure for $\widehat{\alpha}$ can sometimes be used

[5] For a language with pointers, a state is encoded via two functions, an *environment* η and a *store* σ. In general, $\exists \overrightarrow{\text{Voc}}$ will quantify out the unprimed functions η and σ.

to compute $\widehat{\text{Post}}[\tau](A)$ while staying within quantifier-free first-order logic [30, §5].

Other Symbolic Operations. As defined in assumption (4) (§3), 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 $\mathcal{A}_2$ [10, §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 [22] to obtain a two-vocabulary formula ψ_F, and then performing $\widehat{\alpha}(\psi_F)$.

6 Related Work

Extensions of Stålmarck's Method. As mentioned in §1, the work described in this paper was inspired by Stålmarck's method. Our take-off point was the insight that the essence of Stålmarck's method is to employ a particular abstract domain to implement a sound test of whether a formula φ is valid by checking whether $\widehat{\alpha}(\neg\varphi) = \bot$. However, in this paper we go beyond the classical setting of Stålmarck's method, which works with propositional logic, and assume that we have a more expressive logic $\mathcal{L}$, such as quantifier-free linear arithmetic or quantifier-free bit-vector arithmetic.

Because we are working with more expressive logics, our algorithm uses several ideas that go beyond what is used in Stålmarck's method. Our choices were influenced by our experience with some similar operations used in previous work on shape analysis [32]—in particular, the operations of *focus* [32, §6.3] and *coerce* [32, §6.4], which are similar to splitting and propagation, respectively. Our approach has also been 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]. In addition to (in)equations of the form $x_1 \bowtie F(x_1, \ldots, x_n)$, Granger also describes additional techniques for (in)equations of the form $(x_1 * F(x_1, \ldots, x_n)) \bowtie G(x_1, \ldots, x_n)$, which we did not discuss in this paper merely to simplify the presentation. More generally, our framework does not have to be limited to "micro-$\widetilde{\text{Assumes}}$" on literals of these forms. All that we require is that for some family of commonly occurring literals $\{l_i\}$, there is an algorithm to compute $A' = \mu\widetilde{\text{Assume}}[l_i](A)$.

Björk [4] 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's work, the work presented in this paper goes beyond the classical setting of Stålmarck's method (i.e., propositional logic) and extends the method to a more expressive logic. 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 $A' = \widetilde{\text{Assume}}[\varphi](A)$.

In [36], 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 [36] 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 $\widetilde{\text{Post}}[\tau]$.

Best Abstract Operations. A variety of approaches to the problem of performing best abstract operations have appeared in the literature [15, 30, 37, 21, 14]. Graf and Saïdi [15] showed that decision procedures can be used to generate best abstract transformers for predicate-abstraction domains. Predicate abstraction is employed in many software model checkers [1, 19], and its adoption has inspired several investigations of more efficient methods to generate transformers that, in general, are not best transformers, but approach the precision of best transformers [2, 7].

Other work has shown how the benefits enjoyed by tools that use predicate abstraction can also be enjoyed by applications that use abstract domains

other than predicate-abstraction domains. Reps et al. [30] is similar in spirit to the present paper in that it presents a parametric framework for obtaining symbolic abstract operations in a *family* of abstract domains. In particular, they showed that symbolic abstract operations can be performed for all finite-height abstract domains that meet a small number of properties. The constant-propagation domain $(\mathit{Var} \to \mathbb{Z}^\top)_\perp$ is an example of a domain that meets the requirements. However, the technique works from *below*—performing joins to incorporate more and more of the concrete state space—which has the drawback that if it is stopped before the final answer is reached, the most-recent approximation to the final answer 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, the technique from this paper works from *above*, performing meets to eliminate more and more of the concrete state space. It can be stopped at any time and the current approximation returned as a safe answer.

King and Søndergaard [21, Fig. 2] 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). They are both similar to the technique of Reps et al. [30] in that the 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.

Methods that work from above have also been developed. Yorsh et al. [37] developed a framework for performing symbolic abstract operations, such as $\widetilde{\text{Assume}}[\varphi](A)$, for the kind of abstract domains used in shape analysis (i.e., "canonical abstraction" of logical structures [32]). Their method has both similarities to, and differences from, the method presented in this paper. For instance, their method has a splitting step, but no analog of the merge step performed at the end of an invocation of the Dilemma Rule. In addition, the analog of propagation rules in their method is much more heavyweight. As discussed in §2, our propagation rules are local: they make use of only a *single integrity constraint* and a *small constraint* on the current abstract value. 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.

Sankaranarayanan et al. [33] described Template Constraint Matrices (TCMs), a parametrized family of linear inequality domains for expressing invariants in linear real arithmetic. They gave a parametrized meet, join, and set of abstract transformers for all domains in the family. Monniaux [26] 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. The algorithm is based on quantifier elimination over linear-arithmetic formulas, along with some

additional symbolic manipulation; however, the method suffers from the fact that no polynomial-time elimination algorithm is known for piecewise-linear systems.

Brauer and King [6] 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 [25]; the update operation is expressed as an element of the abstract domain of rational affine equalities [20]. 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 $\widehat{\alpha}$ [21, 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 [3] 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 [28] 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. [12] 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 [12, §7.1], and (ii) creating abstract transformers that transform input elements of a logical abstract domain to output elements of the domain [12, §7.2]. Their problems are particular cases of $\widehat{\alpha}$ (or $\widetilde{\alpha}$) and $\widehat{\mathrm{Post}}[\tau]$ (or $\widetilde{\mathrm{Post}}[\tau]$). They present methods for $\widetilde{\alpha}$ and $\widetilde{\mathrm{Post}}[\tau]$ 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 [27]. 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. 7 can be $=$, $\neq$, $<$, $\leq$, $>$, $\geq$, $\equiv$ (arithmetic congruence), or other relational symbols of the logic $\mathcal{L}$.

Cousot and Cousot [10] defined the reduced product, which allows abstract values in different domains to influence each other's value. In ASTRÉE, Cousot et al. [11] 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. [24] 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 $\widetilde{\mathrm{Assume}}[\varphi](A)$. This aspect resembles the "instrumentation predicates" used to control the precision of abstract domains for shape-analysis [32]. However, $\mathcal{P}$'s domain of discourse is the structure of the formula on which $\widetilde{\mathrm{Assume}}$ 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, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, 2001.
2. T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In *TACAS*, pages 268–283, 2001.
3. E. Barrett and A. King. Range and set abstraction using SAT. *Electr. Notes Theor. Comp. Sci.*, 267(1), 2010.
4. M. Björk. First order Stålmarck. *J. Autom. Reasoning*, 42(1):99–122, 2009.
5. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL*, 2003.
6. J. Brauer and A. King. Automatic abstraction for intervals using Boolean formulae. In *SAS*, 2010.
7. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *FMSD*, 25(2–3), 2004.
8. 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.
9. P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *Formal Descr. of Programming Concepts*. North-Holland, 1978.
10. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.
11. 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.
12. P. Cousot, R. Cousot, and L. Mauborgne. Logical abstract domains and interpretations. In *The Future of Software Engineering*, 2011.
13. P. Cousot and N. Halbwachs. Automatic discovery of linear constraints among variables of a program. In *POPL*, pages 84–96, 1978.
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. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.
20. M. Karr. Affine relationship among variables of a program. *Acta Inf.*, 6, 1976.
21. A. King and H. Søndergaard. Automatic abstraction for congruences. In *VMCAI*, 2010.
22. J. King. *A Program Verifier*. PhD thesis, Carnegie-Mellon Univ., Pittsburgh, PA, 1969.
23. J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *CC*, 1992.
24. B. McCloskey, T. Reps, and M. Sagiv. Statically inferring complex heap, array, and numeric invariants. In *SAS*, 2010.
25. A. Miné. The octagon abstract domain. In *WCRE*, 2001.
26. D. Monniaux. Automatic modular abstractions for template numerical constraints. *Logical Methods in Comp. Sci.*, 6(3), 2010.

27. G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *TOPLAS*, 1(2), 1979.
28. J. Regehr and A. Reid. HOIST: A system for automatically deriving static analyzers for embedded systems. In *Arch. Support for Program. Langs. and Oper. Systs.*, 2004.
29. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.
30. T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, pages 252–266, 2004.
31. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *SCP*, 58(1–2), 2005.
32. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.
33. S. Sankaranarayanan, H. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI*, 2005.
34. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.
35. M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. *FMSD*, 16(1), 2000.
36. A. Thakur and T. Reps. A generalization of Stålmarck's method. TR 1699, CS Dept., Univ. of Wisconsin, Madison, WI, Oct. 2011.
37. G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS*, pages 530–545, 2004.

A Proofs

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

Proof. Line (1) calls the function integrity, which converts φ into integrity constraints $\mathcal{I}$ by assigning a fresh Boolean variable to each subformula of φ, using the rules described in Fig. 5. u_φ is assigned to formula φ. We use $\mathcal{U}$ to denote the set of fresh variables created when converting φ to $\mathcal{I}$.

Thus, after line (1) we have

$$[\![\varphi]\!] = ([\![\mathcal{I}]\!] \cap [\![u_\varphi]\!]) \downarrow_{\mathrm{voc}(\varphi)}, \tag{2}$$

where $''M \downarrow \mathrm{voc}(\varphi)''$ denotes the operation of discarding all constants, functions, and relations of a model M that are not in the vocabulary of φ. On line (2) of Alg. 1, an element P of $\mathcal{P}$ is created in which u_φ is assigned the value 1. By assumption (1) about abstract domain $\mathcal{P}$ (see page 10) such a P has to exist. Thus, we have

$$[\![u_\varphi]\!] = \gamma((P, \top)) \tag{3}$$

Using Eqns. (2) and (3), we obtain

$$[\![\varphi]\!] = ([\![\mathcal{I}]\!] \cap \gamma((P, \top))) \downarrow_{\mathrm{voc}\,\varphi} \tag{4}$$

$$\begin{aligned}
[\![\varphi]\!] \cap \gamma(A) &= ([\![\mathcal{I}]\!] \cap \gamma((P, \top_{\mathcal{A}}))) \downarrow_{\mathrm{voc}(\varphi)} \cap \gamma(A) & (5)\\
&= ([\![\mathcal{I}]\!] \cap \gamma((P, \top_{\mathcal{A}})) \cap \gamma((\top_{\mathcal{P}}, A))) \downarrow_{\mathrm{voc}(\varphi)} & (6)\\
&= ([\![\mathcal{I}]\!] \cap \gamma((P, A))) \downarrow_{\mathrm{voc}(\varphi)} & (7)
\end{aligned}$$

After line (3), we have

$$(\widetilde{P}, \widetilde{A}) = \text{k-assume}[\mathcal{I}]((P, A)) \tag{8}$$

Using Thm. 2 on Eqn. (8), we have

$$\gamma_{\mathcal{I}}((\widetilde{P}, \widetilde{A})) = \gamma_{\mathcal{I}}((P, A)) \tag{9}$$

$$(\widetilde{P}, \widetilde{A}) \sqsubseteq (P, A) \tag{10}$$

Using Eqn. (10), we immediately have

$$\widetilde{A} \sqsubseteq A \tag{11}$$

Using Eqn. (9), we have

$$\begin{aligned}
\gamma((\widetilde{P}, \widetilde{A})) \cap [\![I]\!] &= \gamma((P, A)) \cap [\![I]\!]\\
(\gamma((\widetilde{P}, \widetilde{A})) \cap [\![I]\!]) \downarrow_{\mathrm{voc}(\varphi)} &= (\gamma((P, A)) \cap [\![I]\!]) \downarrow_{\mathrm{voc}(\varphi)}\\
(\gamma((\widetilde{P}, \widetilde{A})) \cap [\![I]\!]) \downarrow_{\mathrm{voc}(\varphi)} &= [\![\varphi]\!] \cap \gamma(A) \ \ldots \text{using Eqn. (7)}\\
\gamma((\widetilde{P}, \widetilde{A})) \downarrow_{\mathrm{voc}(\varphi)} &\supseteq [\![\varphi]\!] \cap \gamma(A)\\
\gamma(\widetilde{A}) &\supseteq [\![\varphi]\!] \cap \gamma(A)
\end{aligned}$$

□

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

Proof. We prove this via induction on k. Thm. 3 proves the base case when $k = 0$.

To prove the inductive case, assume that Alg. 2 is sound for $k - 1$, i.e., for all $P \in \mathcal{P}$ and $A \in \mathcal{A}$, if $(P', A') = \text{k-assume}[\mathcal{I}]((P, A))$, then

$$\gamma_{\mathcal{I}}((P', A')) = \gamma_{\mathcal{I}}((P, A)) \text{ and } (P', A') \sqsubseteq (P, A) \tag{12}$$

Let (P^0, A^0) be the value of (P, A) passed as input to Alg. 2, and (P^i, A^i) be the value of (P, A) computed at the end of the ith iteration of the loop body consisting of lines (4)–(8); that is, at line (8).

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

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

At line (3) of Alg. 2, a basis element $B \in \mathcal{B}_{\mathcal{P}}$ is chosen. $\overline{B}$ is the complement of B. By assumption (2) on domain $\mathcal{P}$, we have

$$\gamma((B, \top)) \cup \gamma((\overline{B}, \top) = \gamma((\top, \top)) \tag{13}$$

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

After line (4) of Alg. 2, we have $(P_1, A_1) = (P^0, A^0) \sqcap (B, \top)$, which implies

$$\gamma((P_1, A_1)) \supseteq \gamma((P^0, A^0)) \cap \gamma((B, \top)) \tag{14}$$

Similarly, after line (5) of Alg. 2, we have

$$\gamma((P_2, A_2)) \supseteq \gamma((P^0, A^0)) \cap \gamma((\overline{B}, \top)) \tag{15}$$

From Eqns. (14) and (15), we have

$$\begin{aligned}
\gamma((P_1, A_1)) \cup \gamma((P_2, A_2)) &\supseteq (\gamma((P^0, A^0)) \cap \gamma((B, \top))) \cup (\gamma((P^0, A^0)) \cap \gamma((\overline{B}, \top))) \\
&\supseteq \gamma((P^0, A^0)) \cap (\gamma((B, \top)) \cup \gamma((\overline{B}, \top)) \\
&\supseteq \gamma((P^0, A^0)) \cap \gamma((\top, \top)) \text{ (using Eqn. (13))} \\
&\supseteq \gamma((P^0, A^0))
\end{aligned}$$

Thus, we obtain

$$\gamma_{\mathcal{I}}((P_1, A_1)) \cup \gamma_{\mathcal{I}}((P_2, A_2)) \supseteq \gamma_{\mathcal{I}}((P^0, A^0)) \tag{16}$$

After lines (4) and (5) of Alg. 2 we also have

$$(P_1, A_1) \sqsubseteq (P^0, A^0) \tag{17}$$

$$(P_2, A_2) \sqsubseteq (P^0, A^0) \tag{18}$$

After line (6) of Alg. 2 , we have

$$(P_1', A_1') = \text{(k–1)-assume}[\mathcal{I}]((P_1, A_1)) \tag{19}$$

By using the induction hypothesis (Eqn. (12)) in Eqn. (19), we obtain

$$(P_1', A_1') \sqsubseteq (P_1, A_1) \tag{20}$$

Using Eqns. (17) and (20), we obtain

$$(P_1', A_1') \sqsubseteq (P^0, A^0) \tag{21}$$

Similary, after line (7) of Alg. 2 , we obtain

$$(P_2', A_2') \sqsubseteq (P^0, A^0) \tag{22}$$

Using Eqns. (21) and (22), we obtain

$$(P'_1, A'_1) \sqcup (P'_2, A'_2) \sqsubseteq (P^0, A^0) \tag{23}$$

Using Eqn. (23), we have

$$\gamma((P'_1, A'_1) \sqcup (P'_2, A'_2)) \subseteq \gamma((P^0, A^0))$$
$$\gamma((P'_1, A'_1) \sqcup (P'_2, A'_2)) \cap [\![\mathcal{I}]\!] \subseteq \gamma((P^0, A^0)) \cap [\![\mathcal{I}]\!]$$

This gives us

$$\gamma_{\mathcal{I}}((P'_1, A'_1) \sqcup (P'_2, A'_2)) \subseteq \gamma_{\mathcal{I}}((P^0, A^0)) \tag{24}$$

After line (8) of Alg. 2, we have $(P^1, A^1) = (P'_1, A'_1) \sqcup (P'_2, A'_2)$. Using Eqn. (23), we obtain

$$(P^1, A^1) \sqsubseteq (P^0, A^0) \tag{25}$$

Using Eqn. (24), we obtain

$$\gamma_{\mathcal{I}}(P^1, A^1) \subseteq \gamma_{\mathcal{I}}((P^0, A^0)) \tag{26}$$

Furthermore, by the induction hypothesis (Eqn. (12)), after lines (6) and (7) of Alg. 2, we also have

$$\gamma_{\mathcal{I}}((P'_1, A'_1)) = \gamma_{\mathcal{I}}((P_1, A_1)) \tag{27}$$
$$\gamma_{\mathcal{I}}((P'_2, A'_2)) = \gamma_{\mathcal{I}}((P_2, A_2)) \tag{28}$$

After line (8) of Alg. 2, we have

$$\gamma_{\mathcal{I}}((P^1, A^1)) = \gamma_{\mathcal{I}}((P'_1, A'_1) \sqcup (P'_2, A'_2))$$
$$\gamma_{\mathcal{I}}((P^1, A^1)) \supseteq \gamma_{\mathcal{I}}((P'_1, A'_1)) \cup \gamma_{\mathcal{I}}((P'_2, A'_2))$$
$$\gamma_{\mathcal{I}}((P^1, A^1)) \supseteq \gamma_{\mathcal{I}}((P_1, A_1)) \cup \gamma_{\mathcal{I}}((P_2, A_2)) \ \ldots \text{using Eqns. (27) and (28)}$$

Using Eqn. (16), we obtain

$$\gamma_{\mathcal{I}}((P^1, A^1)) \supseteq \gamma_{\mathcal{I}}((P^0, A^0)) \tag{29}$$

Using Eqns. (26) and (29), we have

$$\gamma_{\mathcal{I}}((P^1, A^1)) = \gamma_{\mathcal{I}}((P^0, A^0)) \tag{30}$$

Thus, using Eqns. (25) and (30) we have proved the base case with $i = 1$. An almost identical proof can be used to prove the inductive case for the induction on i. Thus, for each i, $\gamma_{\mathcal{I}}((P^i, A^i)) = \gamma_{\mathcal{I}}((P^0, A^0))$ and $(P^i, A^i) \sqsubseteq (P^0, A^0)$.

Because the value returned by Alg. 2 is the final value computed at line (8), we have proven the inductive case for the induction on k. Thus, by induction, we have shown that for all $P \in \mathcal{P}$ and $A \in \mathcal{A}$, if $(P', A') = \text{k-assume}[\mathcal{I}]((P, A))$, then $\gamma_{\mathcal{I}}((P', A')) = \gamma_{\mathcal{I}}((P, A))$ and $(P', A') \sqsubseteq (P, A)$. □

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

Proof. Let (P^0, A^0) be the value of (P, A) passed as input to Alg. 3, and (P^i, A^i) be the value of (P, A) computed at the end of the ith iteration of the loop body consisting of lines (4)–(9); that is, at line (9).

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

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

$$J \in \mathcal{I} \tag{31}$$

$$(P_1, A_1) \sqsupseteq (P^0, A^0) \tag{32}$$

From Eqn. (31), we have

$$[\![J]\!] \supseteq [\![\mathcal{I}]\!] \tag{33}$$

From Eqn. (32), we have

$$\gamma((P_1, A_1)) \supseteq \gamma((P^0, A^0)) \tag{34}$$

Using Eqns. (33) and (34),

$$\gamma((P_1, A_1)) \cap [\![J]\!] \supseteq \gamma((P^0, A^0)) \cap [\![\mathcal{I}]\!] \tag{35}$$

It is easy to show that, given $J \in \mathcal{I}$ and $(P_1, A_1) \sqsupseteq (P, A)$, the net effect of applying any of the propagation rules in Figs. 7 and 8 is to compute a semantic reduction of (P_1, A_1) with respect to $J \in \mathcal{I}$. Thus, after lines (6) and (8), we have

$$\gamma((P_2, A_2)) \cap [\![J]\!] = \gamma((P_1, A_1)) \cap [\![J]\!] \tag{36}$$

Using Eqns. (35) and (36),

$$\gamma((P_2, A_2)) \cap [\![J]\!] \supseteq \gamma((P^0, A^0)) \cap [\![\mathcal{I}]\!] \tag{37}$$

$$(\gamma((P_2, A_2)) \cap [\![J]\!]) \cap [\![\mathcal{I}]\!] \supseteq (\gamma((P^0, A^0)) \cap [\![\mathcal{I}]\!]) \cap [\![\mathcal{I}]\!] \tag{38}$$

Using Eqns. (33) and (38),

$$\gamma((P_2, A_2)) \cap [\![\mathcal{I}]\!] \supseteq \gamma((P^0, A^0)) \cap [\![\mathcal{I}]\!] \tag{39}$$

After line (9) of Alg. 3, we have

$$(P^1, A^1) = (P_2, A_2) \sqcap (P^0, A^0) \tag{40}$$

Using Eqn. (40), we obtain

$$(P^1, A^1) \sqsubseteq (P^0, A^0) \tag{41}$$

$$\gamma((P^1, A^1)) = \gamma((P_2, A_2)) \cap \gamma((P^0, A^0)) \tag{42}$$

Using Eqn. (42),

$$\gamma((P^1, A^1)) \cap [\![\mathcal{I}]\!] = (\gamma((P_2, A_2)) \cap \gamma((P^0, A^0))) \cap [\![\mathcal{I}]\!] \tag{43}$$

$$= (\gamma((P_2, A_2)) \cap [\![\mathcal{I}]\!]) \cap (\gamma((P^0, A^0)) \cap [\![\mathcal{I}]\!]) \tag{44}$$

Using Eqn. (39) in Eqn. (44), we have

$$\gamma((P^1, A^1)) \cap [\![\mathcal{I}]\!] = \gamma((P^0, A^0)) \cap [\![\mathcal{I}]\!] \tag{45}$$

This gives us

$$\gamma_{\mathcal{I}}((P^1, A^1)) = \gamma_{\mathcal{I}}((P^0, A^0)) \tag{46}$$

Thus, Eqns. (41) and (46) prove the base case of the induction for $i = 1$. An almost identical proof can be used to prove the inductive case for the induction on i. Thus, for each i, $\gamma_{\mathcal{I}}((P^i, A^i)) = \gamma_{\mathcal{I}}((P^0, A^0))$ and $(P^i, A^i) \sqsubseteq (P^0, A^0)$.

Because the value returned by Alg. 3 is the final value of (P, A) computed at line (9), we have shown that for all $P \in \mathcal{P}, A \in \mathcal{A}$, if $(P', A') =$ 0-assume$[\mathcal{I}]((P, A))$, then $\gamma_{\mathcal{I}}((P', A')) = \gamma_{\mathcal{I}}((P, A))$ and $(P', A') \sqsubseteq (P, A)$. □