Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 311 (2015) 15–32

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

PostHat and All That: Automating Abstract Interpretation

A. Thakur [a,1], A. Lal [b,2], J. Lim [c,3], and T. Reps [a,c,4]

[a] *Computer Sciences Department, Univ. of Wisconsin; Madison, WI; USA*

[b] *Microsoft Research India; Bangalore; India*

[c] *GrammaTech, Inc.; Ithaca, NY; USA*

Abstract

Abstract interpretation provides an elegant formalism for performing program analysis. Unfortunately, designing and implementing a sound, precise, scalable, and extensible abstract interpreter is difficult. In this paper, we describe an approach to creating correct-by-construction abstract interpreters that also *attain the fundamental limits* on precision that abstract-interpretation theory establishes. Our approach requires the analysis designer to implement only a small number of operations. In particular, we describe a systematic method for implementing an abstract interpreter that solves the following problem:

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

Keywords: abstract interpretation, invariant generation, symbolic abstraction, decision procedures

1 Introduction

Computing invariants via static program analysis is crucial when proving correctness of programs. For the analysis to be tractable, the language of invariants is restricted. In particular, an *abstract domain* is used to specify the program properties that are observable. Consequently, the program analysis works on an *abstraction* of a program, which over-approximates the original program's behavior. As long as the abstract semantics is an over-approximation of the concrete semantics of the

[1] Email: adi@cs.wisc.edu

[2] Email: akashl@microsoft.com

[3] Email: junghee@grammatech.com

[4] Email: reps@cs.wisc.edu

[5] Supported, in part, by NSF under grant CCF-0904371; by ONR under grants N00014-{09-1-0510, 11-C-0447}; by AFRL under contract FA8650-10-C-7088; and by DARPA under cooperative agreement HR0011-12-2-0012. 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.

http://dx.doi.org/10.1016/j.entcs.2015.02.003

program, the program properties inferred describe a superset of the states that can actually occur, and can be used as invariants.

The theory underlying this approach is called *abstract interpretation* [3]. Unfortunately, abstract interpretation has a well-deserved reputation of being a kind of "black art", and building sound, precise, scalable, and extensible analyzers is often a difficult process. This paper describes techniques that can lessen the burden on analysis designers. When they are applicable, the techniques presented in this paper address a long-standing open question—namely, how to raise the level of automation in abstract interpretation. Moreover, they provide help along all of four of the dimensions mentioned above:

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

precision: Unlike most conventional approaches to creating analyzers, our techniques achieve the fundamental limits of precision that abstract-interpretation theory establishes.

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

extensibility: If an additional abstract domain is added to an analyzer to track additional information, the reduced product [4, §10.1] can be obtained automatically [33, §6]. That is, information will be exchanged automatically between domains to produce the most-precise abstract values in each domain.

Let $\mathcal{C}$ be the concrete domain that describes the collecting semantics of the program. For a concrete transformer τ, let $\mathrm{Post}[\tau] : \mathcal{C} \to \mathcal{C}$ denote the operator that applies the concrete transformer. A set of invariants $\{I_k\}$ are said be *inductive* with respect to a set of transformers $\{\tau_{ij}\}$ if, for all i and j, $\mathrm{Post}[\tau_{ij}](\llbracket I_i \rrbracket) \subseteq \llbracket I_j \rrbracket$, where $\llbracket I_k \rrbracket \in \mathcal{C}$ denotes the meaning of I_k. The choice of a particular abstract domain $\mathcal{A}$ fixes a limit on the precision of the invariants identified by an analysis. If $\{I_k\} \subseteq \mathcal{A}$, then $\{I_k\}$ is said to be an inductive $\mathcal{A}$-*invariants*. (For brevity, we also refer to a single member I_j of $\{I_k\}$ as an inductive $\mathcal{A}$-invariant.) Furthermore, a *most-precise* inductive $\mathcal{A}$-invariant exists: $\mathrm{Post}[\tau]$ is monotonic in $\mathcal{C}$; given two $\mathcal{A}$-invariants, we can take their meet; thus, provided $\mathcal{A}$ is a meet semi-lattice, the most-precise inductive $\mathcal{A}$-invariant exists. [6]

As discussed in this paper, finding the most-precise inductive $\mathcal{A}$-invariant is attainable when $\mathcal{A}$ meets certain properties. Formally, suppose that one has a Galois connection $\mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$ between concrete domain $\mathcal{C}$ and abstract domain $\mathcal{A}$. For a concrete transformer τ, let $\widehat{\mathrm{Post}}[\tau] : \mathcal{A} \to \mathcal{A}$ be the most-precise abstract

[6] On the other hand, computing the *most precise* $\mathcal{A}$-*invariant* at a program point, defined as the abstraction of the collecting semantics at that point, is generally infeasible.

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

Observation 1 *Let program P consist of (i) nodes $N = \{n_i\}$ with enter node n_1, (ii) edges $E_P = \{n_i \to n_j\}$, and (iii) a concrete-state transformer $\tau_{i,j}$ associated with each edge $n_i \to n_j$. Let $\mathcal{A}$ be an abstract domain. The* ***best inductive invariant*** *(BII) for P that is expressible in $\mathcal{A}$ is the least fixed-point of the equation system*

$$V_1 = a_1 \qquad V_j = \bigsqcup_{n_i \to n_j \in E_P} \widehat{\text{Post}}[\tau_{i,j}](V_i), \tag{1}$$

where a_1 is the best value in $\mathcal{A}$ that over-approximates the set of allowed input states at the enter node n_1.

As a corollary of Obs. 1, we have

Observation 2 *When the least solution to Eqn. (1) can be obtained algorithmically, e.g., by Kleene iteration in an abstract domain with no infinite ascending chains, the BII problem reduces to the problem of applying $\widehat{\text{Post}}$.*

Unfortunately, the theory of abstract interpretation does not provide a useful algorithm for applying $\widehat{\text{Post}}[\tau]$ given the concrete transformer τ. $\widehat{\text{Post}}$ can be expressed in terms of α, γ, and $\text{Post}[\tau]$, as follows [4]:

$$\widehat{\text{Post}}[\tau] = \alpha \circ \text{Post}[\tau] \circ \gamma. \tag{2}$$

However, in most cases, the application of γ to an abstract value in Eqn. (2) would yield an intermediate result—a set of concrete states—that is either infinite or too large to fit in computer memory.

In practice, analysis designers typically give up on Eqn. (2)—thereby giving up on attaining the best inductive $\mathcal{A}$-invariant—and manually write, for each concrete operation, an abstract transformer that satisfies the weaker condition

$$\text{Post}^\sharp[\tau] \sqsupseteq \alpha \circ \text{Post}[\tau] \circ \gamma. \tag{3}$$

Consequently, in practice, an analysis designer needs to manually write the abstract transformers for each concrete operation. This task can be tedious and error-prone, especially for machine code where most instructions involve bit-wise operations, as illustrated using two simple examples. Ex. 1.1 illustrates that applying $\widehat{\text{Post}}$ can be complex even for a single instruction, and Ex. 1.2 motivates why a technique for applying $\widehat{\text{Post}}$ for a sequence of instructions is desirable.

Example 1.1 Consider the Intel x86 instruction $\tau \stackrel{\text{def}}{=}$ `add bh,al`, which adds `al`, the low-order byte of 32-bit register `eax`, to `bh`, the second-to-lowest byte of 32-bit register `ebx`. The registers `eax` and `ecx` are not modified by the instruction. The semantics of τ can be expressed in quantifier-free bit-vector logic (QFBV) as the

formula φ_τ:

$$\varphi_\tau \stackrel{\text{def}}{=} \texttt{ebx}' = \begin{pmatrix} (\texttt{ebx}\ \&\ \texttt{0xFFFF00FF}) \\ |\ ((\texttt{ebx} + 256 * (\texttt{eax}\ \&\ \texttt{0xFF}))\ \&\ \texttt{0xFF00}) \end{pmatrix} \begin{matrix} \wedge\, \texttt{eax}' = \texttt{eax} \\ \wedge\, \texttt{ecx}' = \texttt{ecx}, \end{matrix} \tag{4}$$

where "&" and "|" denote bitwise-and and bitwise-or, respectively, and a symbol with a prime denotes the value of the symbol in the post-state. Eqn. (4) shows that the semantics of a seemingly simple `add` instruction involves non-linear bit-masking operations.

Now suppose that the abstract domain $\mathcal{A}$ is the domain of relational affine equalities among 32-bit registers [8]. We would like to compute $\widehat{\text{Post}}[\tau](a)$, where $a \in \mathcal{A}$ is $\texttt{ebx} = \texttt{ecx}$. Computing $\widehat{\text{Post}}[\tau](A)$ corresponds to finding the *strongest affine relation* that holds among $\texttt{eax}'$, $\texttt{ebx}'$, and $\texttt{ecx}'$ after τ is executed starting from any concrete state in $\gamma(a)$. For this example, $\widehat{\text{Post}}[\tau](a) \stackrel{\text{def}}{=} (2^{16}\texttt{ebx}' = 2^{16}\texttt{ecx}' + 2^{24}\texttt{eax}') \wedge (2^{24}\texttt{ebx}' = 2^{24}\texttt{ecx}')$. Multiplying by a power of 2 serves to shift bits to the left; because it is performed in arithmetic mod 2^{32}, bits shifted off the left end are unconstrained. Thus, the first conjunct of $\widehat{\text{Post}}[\tau](a)$ captures the relationship between the low-order two bytes of $\texttt{ebx}'$, the low-order two bytes of $\texttt{ecx}'$, and the low-order byte of $\texttt{eax}'$.

One approach to applying an abstract transformer is based on the structure of the term used to express the concrete transformer: a sound abstract operator—$\&^\sharp$, $+^\sharp$, $*^\sharp$, $|^\sharp$, and $=^\sharp$—is used in place of each concrete operator— &, +, ∗, |, and =, respectively. In general, such an operator-by-operator reinterpretation approach is sound, but is not guaranteed to compute $\widehat{\text{Post}}$. For this example, the reinterpretation approach results in $2^{24}\texttt{ebx}' = 2^{24}\texttt{ecx}'$, which is a strict over-approximation of $\widehat{\text{Post}}[\tau](a)$ [9]. □

Example 1.2 Consider the two Intel x86 instructions $\tau_1 \stackrel{\text{def}}{=}$ `push 42` and $\tau_1 \stackrel{\text{def}}{=}$ `pop eax`. The instruction τ_1 pushes the value 42 onto the stack, and τ_2 pops the value on top of the stack into the register `eax`.

As in Ex. 1.1, we consider the abstract domain $\mathcal{A}$ of relational affine equalities among 32-bit registers [8]. We would like to compute $\widehat{\text{Post}}[\tau_1;\tau_2](a)$, where $a = \top$; that is, we want to apply the abstract transformer for the sequential composition of τ_1 and τ_2. One approach is to compute the value $\widehat{\text{Post}}[\tau_2](\widehat{\text{Post}}[\tau_1](a))$. $\widehat{\text{Post}}[\tau_1](a) \stackrel{\text{def}}{=} \top$, because $\mathcal{A}$ is able to only capture relations between registers, and is incapable of holding onto properties of values on the stack. Consequently, $\widehat{\text{Post}}[\tau_2](\widehat{\text{Post}}[\tau_1](a)) = \widehat{\text{Post}}[\tau_2](\top) = \top$. In contrast, $\widehat{\text{Post}}[\tau_1;\tau_2](A) \stackrel{\text{def}}{=} \texttt{eax}' = 42$. □

Exs. 1.1 and 1.2 illustrate the fact that an abstract domain can be expressive enough to capture invariants before and after a sequence of operations or instructions, but not capable of capturing invariants at some intermediate point. As illustrated in Ex. 1.1, the application of a sequence of sound abstract *operations* can lose precision because it is necessary to express the intermediate result in the limited language supported by the abstract domain. Ex. 1.2 illustrates that a similar phenomenon holds at the level of a sequence of *instructions*: again, the need to

express an intermediate result in the limited language supported by the abstract domain can cause a loss of precision.

The precision obtained from a solution to BII depends on the set of "observation points" (or, equivalently the equation system being solved). For instance, in Ex. 1.2, the strawman solution does compute the *best* inductive $\mathcal{A}$-invariant if the intermediate point between τ_1 and τ_2 is observable. Ex. 1.2 shows that, from the standpoint of precision, the fewer observation points, the better. As with many methods in automatic program analysis and verification, our method normally requires that each loop be cut by at least one observation point. Thus, the set of loop headers would be a natural choice for the set of observation points. The take-away from this discussion is that it can be desirable to have a procedure that is capable of applying $\widehat{\text{Post}}$ for an arbitrary loop-free sequence of instructions.

Most program-analysis steps, including the application of $\widehat{\text{Post}}$, can be cast as a problem that requires bridging a gap between (i) the use of logic for specifying program semantics and program correctness, and (ii) abstract interpretation. The reason logic comes into play is because the problem of needing to apply γ in Eqn. (2) can be side-stepped by working with *symbolic* representations of sets of states (i.e., using formulas in some logic $\mathcal{L}$). The use of $\mathcal{L}$-formulas to represent sets of states is convenient because logic can also be used to specify a language's concrete semantics; i.e., the concrete semantics of a transformer $\text{Post}[\tau]$ can be stated as a formula $\varphi_\tau \in \mathcal{L}$ that specifies the relation between input and output states.

The key insight behind our work is that the problem of bridging the gap between the use of logic and abstract interpretation can be cast as the problem of *symbolic abstraction* [25,29,31,33]: "Given a formula $\varphi \in \mathcal{L}$, find the most-precise value in abstract domain $\mathcal{A}$ that over-approximates the meaning of φ". In other words, the symbolic abstraction of φ, denoted by $\widehat{\alpha}(\varphi)$, is the strongest consequence of φ that can be expressed in $\mathcal{A}$. In particular, when $\mathcal{L}$ is powerful enough to express the full semantics of a given programming language of interest, and $\widehat{\gamma}$ is an operation that maps $a \in \mathcal{A}$ to an $\mathcal{L}$-formula that captures $\gamma(a)$,

$$\widehat{\text{Post}}[\tau](a) = \widehat{\alpha}(\varphi_\tau \wedge \widehat{\gamma}(a)). \tag{5}$$

Eqn. (5) yields the following insight: the problem of applying $\widehat{\text{Post}}$ reduces to the problem of symbolic abstraction.

Several frameworks, parameterized by $\mathcal{L}$ and $\mathcal{A}$, have been developed for solving the symbolic-abstraction problem [25,33,29]. In addition, some of the frameworks [33,29] implement an "anytime" algorithm—i.e., the algorithm can be equipped with a monitor, and if too much time or space is being used, the algorithm can be stopped at any time, and provide a safe (over-approximating) answer.

A key advantage of these symbolic-abstraction frameworks is that they help to raise the level of automation in abstract interpretation: an analysis designer need only implement a comparatively small number of operations: $\widehat{\gamma}$, β, and $\sqcup$ (see §2; Alg. 1). If implementations of $\sqcap$ and $\sqsubseteq$ are also provided, they enable the use of anytime algorithms (see §2; Alg. 2).

It is worthwhile recapping the advantages and non-standard aspects of our approach:

- The symbolic-abstraction approach to implementing a program analyzer imposes much less of a burden on an analysis designer than conventional approaches: he need only supply a small number of operations. [7] Moreover, the operations are almost entirely agnostic to both the programming language to be analyzed and the logic used to specify the concrete semantics.
  - One of the required operations (called β, see §2) involves converting a concrete state σ to the most-precise element of the abstract domain that over-approximates $\{\sigma\}$. When the analysis is cast as finding the least fixed-point of an equation system (cf. Eqn. (1)), β is the only direct connection to the programming language and logic in use.
  - The rest of the operations that the analysis designer must specify are *abstract-domain* operations. In particular, the analysis designer does not explicitly specify either abstract transformers (e.g., an abstract version of "`x := y + 1`") or specific abstract operations for the operations of the programming language or logic ($\&^\sharp$, $+^\sharp$, $*^\sharp$, $|^\sharp$, etc.).
- Eqn. (1) has a conventional form, but is non-standard because it uses the application of the most-precise abstract transformer $\widehat{\text{Post}}[\tau_{i,j}]$, whereas most work on abstract interpretation uses less-precise abstract transformers. For instance, some systems only support the state transformations of a restricted modeling language—e.g., affine programs [5,22] or Boolean programs [1]. Transformations outside the modeling language are over-approximated very coarsely. For example, for an assignment statement "$\mathsf{x} := \mathsf{e}$" in the original program, if e is an expression that uses any non-modeled operator, the statement is modeled as "$\mathsf{x} := ?$" or, equivalently, "$\mathsf{havoc}(\mathsf{x})$". (That is, after "$\mathsf{x} := \mathsf{e}$" executes, x can hold any value.) This translation from a program to the program-modeling language already involves some loss of precision.

  In contrast, the application of $\widehat{\text{Post}}[\tau_{i,j}]$ in Eqn. (1) always incorporates the full concrete semantics of $\tau_{i,j}$ (i.e., without an a priori abstraction step).
- The analyzers obtained via our approach can find best inductive $\mathcal{A}$-invariants.

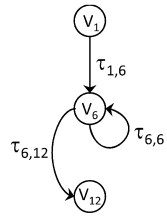
Not only does the work provide insight on fundamental limits in abstract interpretation, the algorithms that we present are also practical. We created an invariant-generation tool called Santini based on the principles of symbolic abstraction. Santini uses the predicate-abstraction domain that can infer arbitrary Boolean combinations of a given set of predicates. The implementation of the abstract domain was simple: just 1200 lines of C# code. We then compared the performance of Santini with Houdini [11], which is a well-established tool that infers only conjunctive invariants from a given set of predicates. We ran the Corral model checker [18] using invariants supplied by Houdini and Santini. For 19 examples for which Corral gave an indefinite result using Houdini-supplied invariants, invariants discovered using Santini allowed Corral to give a definite result in 9 cases (47%); see [31, §5].

[7] In this discussion, we are talking about the implementation of the analysis component that deals with abstract transformers. We assume that we already have a procedure to obtain the set of concrete transformers $\{\tau_{ij}\}$, expressed in a suitable decidable logic, for each loop-free fragment of a program.

(a)

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

(b)



(c)

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

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

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

Fig. 1. (a) Example program. (b) Dependences among node-variables in the program's equation system (over node-variables $\{V_1, V_6, V_{12}\}$). (c) The transition relations among $\{V_1, V_6, V_{12}\}$ (expressed as formulas).

This experiment shows that it is possible to quickly implement a correct and precise invariant generator that uses an expressive abstract domain. The other instantiation of our BII framework is based on WALi [16], a tool for solving program-analysis problems using an abstract domain, which we have used to perform machine-code analysis [31, §5].

Proofs for theorems in this paper can be found in [31].

2 Basic Insights

Fig. 1(a) shows an example program that we will use to illustrate finding the best inductive invariant when the abstract domain is the domain of relational affine equalities over integers mod 2^{32} [8]. We concentrate on lines (1), (6), and (12). Fig. 1(b) depicts the dependences in the equation system over node-variables $\{V_1, V_6, V_{12}\}$. Fig. 1(c) gives formulas for the transition relations among $\{V_1, V_6, V_{12}\}$. The remainder of this section illustrates how to solve the BII problem for the following equation system, which corresponds to Figs. 1(b) and 1(c):

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

It is convenient to rewrite these equations as

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

Solving the BII Problem from Below. In the most basic approach to solving

Algorithm 1: $\widehat{\text{Post}}^{\uparrow}[\tau](v)$

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

Algorithm 2: $\widehat{\text{Post}}^{\updownarrow}[\tau](v)$

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

the BII problem, we assume that we have an essentially standard fixed-point solver that performs chaotic iteration. The method will create successively better under-approximations to the solution, until it finds a fixed point, which will also be the best inductive invariant. We illustrate the algorithm on Eqn. (6).

What is special, compared to standard equation solvers, is that each application of the right-hand side of an equation in Eqn. (6)—defined by the corresponding formula in Fig. 1(c)—is given the *best-transformer interpretation* by means of a function $\widehat{\text{Post}}$ for applying the best abstract transformer. That is, $\widehat{\text{Post}}$ satisfies $\widehat{\text{Post}}[\tau](a) = (\alpha \circ \text{Post}[\tau] \circ \gamma)(a)$. A specific instance of $\widehat{\text{Post}}$ is the function $\widehat{\text{Post}}^{\uparrow}$, given as Alg. 1 [25].

Each call to $\widehat{\text{Post}}^{\uparrow}[\tau](v)$ performs a successive-approximation process, working up from $\bot$ (line 2), to identify v' such that $v' = (\alpha \circ \text{Post}[\tau] \circ \gamma)(v)$. This successive-approximation process is based on the following principle:

- $\widehat{\text{Post}}^{\uparrow}[\tau](v)$ computes the answer via a (sophisticated) iterative sampling process, using an SMT solver. Blocking clauses are used to push the solver to explore a new part of the state space on each iteration.

To guarantee that the algorithm terminates with the most-precise result, $\widehat{\text{Post}}^{\uparrow}$ imposes certain requirements: (i) abstract domain $\mathcal{A}$ must be a join semi-lattice with a least element $\bot$, and have no infinite ascending chains;[8] (ii) logic $\mathcal{L}$ must be closed under $\wedge$ and $\neg$; and (iii) certain operations must be available:

- The operation of *symbolic concretization* (line 5 of Alg. 1), 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., $\gamma(a) = [\![\widehat{\gamma}(a)]\!]$, where $[\![\cdot]\!]$ denotes the meaning function of $\mathcal{L}$).
- Given a formula $\psi \in \mathcal{L}$, `Model`(ψ) returns (i) a satisfying model S if an SMT solver is able to determine that ψ is satisfiable in a given time limit, (ii) None if the SMT solver is able to determine that ψ is unsatisfiable in the time limit, and (iii) TimeOut otherwise.

The formula $\widehat{\gamma}(v) \wedge \tau \wedge \neg\widehat{\gamma}(\mathit{lower}')$ to which `Model` is applied in line 5 is

[8] For domains with infinite ascending chains, the techniques is this paper can be used to generate sound but not necessarily most-precise transformers. Specifically, the use of join in the algorithms can be replaced by the use of widening operators to guarantee termination.

a transition formula, and thus `Model` returns a two-state model $\langle S, S' \rangle$, which we refer to as a *state-pair*. (In general, we use unprimed variables to denote pre-state quantities, and primed variables to denote post-state quantities.)

- The *representation function* β (line 11 of Alg. 1) maps a singleton concrete state $S \in \mathcal{C}$ to the least value in $\mathcal{A}$ that over-approximates $\{S\}$.

The variable $lower'$ is initialized to $\bot$ (line 2). Then, on each iteration of the while-loop on lines 3–11 a concrete state-pair $\langle S, S' \rangle$ is identified that satisfies transition relation τ—as constrained by $\widehat{\gamma}(v)$ and $\neg\widehat{\gamma}(lower')$—(line 5), and the abstraction $\beta(S')$ of post-state S' is joined into $lower'$ (line 11). Each concrete state-pair $\langle S, S' \rangle$ is obtained by calling an SMT solver to obtain a satisfying assignment of the transition relation under consideration. Abstract value $lower'$ characterizes the set of already-found post-states. To ensure that a new post-state is found on each iteration, the formula $\neg\widehat{\gamma}(lower')$ is used as a blocking clause for the next call on the SMT solver; see the third conjunct in line 5. Thus, the algorithm does not merely ask the SMT solver for a new post-state; it requests a post-state that guarantees progress. That is, the post-state S' returned by the SMT solver is one for which $lower' \sqsupsetneq lower' \sqcup \beta(S')$.

Fig. 2 shows a possible chaotic-iteration sequence when a BII solver is invoked to find the best inductive affine-equality invariant for Eqn. (6), namely, $\langle \mathtt{V}_1 \mapsto \top, \mathtt{V}_6 \mapsto [\mathtt{a} = \mathtt{b}, \mathtt{x} = \mathtt{y}], \mathtt{V}_{12} \mapsto [\mathtt{a} = \mathtt{b}, \mathtt{x} = \mathtt{y}] \rangle$.[9] Note that this abstract value corresponds exactly to the loop-invariant and exit-invariant shown in the comments on lines 6 and 12 of Fig. 1(a). Moreover, the same abstract value would be arrived at no matter what sequence of choices is made during chaotic iteration to find the least fixed-point of Eqn. (6).

One such sequence is depicted in Fig. 2, where three chaotic-iteration steps are performed before the least fixed point is found. The three steps propagate information from V_1 to V_6; from V_6 to V_6; and from V_6 to V_{12}, respectively. (At this point, to discover that chaotic iteration has quiesced, the solver would have to do some additional work, which we have not shown because it does not provide any additional insight on how BII problems are solved.)

The Value of a Bilateral Algorithm. $\widehat{\text{Post}}^{\uparrow}$ is not resilient to timeouts. A query to the SMT solver—or the cumulative time for $\widehat{\text{Post}}^{\uparrow}$—might take too long, in which case the only answer that is safe for $\widehat{\text{Post}}^{\uparrow}$ to return is $\top$ (line 7 of Alg. 1). To remedy this situation, we use a *bilateral* algorithm for $\widehat{\text{Post}}$ [29]. A bilateral algorithm maintains both an under-approximation and an over-approximation of the desired answer. The advantage of a bilateral algorithm is that in case of a timeout, it is safe to return the over-approximation. At various stages of the algorithm, effort is expended on improving the over-approximation, and if the algorithm were given additional time, it might return a better answer. (Such an algorithm is called an

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

Initialization: $\texttt{ans} := \langle \mathtt{V}_1 \mapsto \top, \mathtt{V}_6 \mapsto \bot, \mathtt{V}_{12} \mapsto \bot \rangle$

Iteration 1: $V_6 := V_6 \sqcup \widehat{\text{Post}}^{\uparrow}[\tau_{1,6}](V_1)$

$= \bot \sqcup \widehat{\text{Post}}^{\uparrow}[\tau_{1,6}](\top)$

$lower' := \bot$

$\langle S, S' \rangle := \texttt{Model}(\texttt{true} \wedge \tau_{1,6} \wedge \neg\widehat{\gamma}(\bot))$

$= \begin{bmatrix} a \mapsto 42, b \mapsto 27, x \mapsto 5, y \mapsto 99, \\ a' \mapsto 17, b' \mapsto 17, x' \mapsto 0, y' \mapsto 0 \end{bmatrix}$ // A satisfying concrete state-pair

$lower' := \bot \sqcup [\mathtt{a}' = 17, \mathtt{b}' = 17, \mathtt{x}' = 0, \mathtt{y}' = 0]$

$\langle S, S' \rangle := \texttt{Model}(\texttt{true} \wedge \tau_{1,6} \wedge \neg\widehat{\gamma}([\mathtt{a}' = 17, \mathtt{b}' = 17, \mathtt{x}' = 0, \mathtt{y}' = 0]))$

$= \begin{bmatrix} a \mapsto 73, b \mapsto 2, x \mapsto 15, y \mapsto 19, \\ a' \mapsto 28, b' \mapsto 28, x' \mapsto 0, y' \mapsto 0 \end{bmatrix}$ // A satisfying concrete state-pair

$lower' := [\mathtt{a}' = 17, \mathtt{b}' = 17, \mathtt{x}' = 0, \mathtt{y}' = 0] \sqcup [\mathtt{a}' = 28, \mathtt{b}' = 28, \mathtt{x}' = 0, \mathtt{y}' = 0]$

$v' := [\mathtt{a}' = \mathtt{b}', \mathtt{x}' = 0, \mathtt{y}' = 0]$

$V_6 := \bot \sqcup [\mathtt{a} = \mathtt{b}, \mathtt{x} = 0, \mathtt{y} = 0] = [\mathtt{a} = \mathtt{b}, \mathtt{x} = 0, \mathtt{y} = 0]$

$\texttt{ans} := \langle \mathtt{V}_1 \mapsto \top, \mathtt{V}_6 \mapsto [\mathtt{a} = \mathtt{b}, \mathtt{x} = 0, \mathtt{y} = 0], \mathtt{V}_{12} \mapsto \bot \rangle$

Iteration 2: $V_6 := V_6 \sqcup \widehat{\text{Post}}^{\uparrow}[\tau_{6,6}](V_6)$

$= [\mathtt{a} = \mathtt{b}, \mathtt{x} = 0, \mathtt{y} = 0] \sqcup \widehat{\text{Post}}^{\uparrow}[\tau_{6,6}]([\mathtt{a} = \mathtt{b}, \mathtt{x} = 0, \mathtt{y} = 0])$

$lower' := \bot$

$\langle S, S' \rangle := \texttt{Model}(\widehat{\gamma}([\mathtt{a} = \mathtt{b}, \mathtt{x} = 0, \mathtt{y} = 0]) \wedge \tau_{6,6} \wedge \neg\widehat{\gamma}(\bot))$

$= \begin{bmatrix} a \mapsto 56, b \mapsto 56, x \mapsto 0, y \mapsto 0, \\ a' \mapsto 58, b' \mapsto 58, x' \mapsto 1, y' \mapsto 1 \end{bmatrix}$ // A satisfying concrete state-pair

$lower' := \bot \sqcup [\mathtt{a}' = 58, \mathtt{b}' = 58, \mathtt{x}' = 1, \mathtt{y}' = 1]$

$\langle S, S' \rangle := \texttt{Model}(\widehat{\gamma}([\mathtt{a} = \mathtt{b}, \mathtt{x} = 0, \mathtt{y} = 0]) \wedge \tau_{6,6} \wedge \neg\widehat{\gamma}([\mathtt{a}' = 58, \mathtt{b}' = 58, \mathtt{x}' = 1, \mathtt{y}' = 1]))$

$= \begin{bmatrix} a \mapsto 16, b \mapsto 16, x \mapsto 0, y \mapsto 0, \\ a' \mapsto 18, b' \mapsto 18, x' \mapsto 1, y' \mapsto 1 \end{bmatrix}$ // A satisfying concrete state-pair

$lower' := [\mathtt{a}' = 58, \mathtt{b}' = 58, \mathtt{x}' = 1, \mathtt{y}' = 1] \sqcup [\mathtt{a}' = 18, \mathtt{b}' = 18, \mathtt{x}' = 1, \mathtt{y}' = 1]$

$v' := [\mathtt{a}' = \mathtt{b}', \mathtt{x}' = 1, \mathtt{y}' = 1]$

$V_6 := [\mathtt{a} = \mathtt{b}, \mathtt{x} = 0, \mathtt{y} = 0] \sqcup [\mathtt{a} = \mathtt{b}, \mathtt{x} = 1, \mathtt{y} = 1] = [\mathtt{a} = \mathtt{b}, \mathtt{x} = \mathtt{y}]$

$\texttt{ans} := \langle \mathtt{V}_1 \mapsto \top, \mathtt{V}_6 \mapsto [\mathtt{a} = \mathtt{b}, \mathtt{x} = \mathtt{y}], \mathtt{V}_{12} \mapsto \bot \rangle$

Iteration 3: $V_{12} := V_{12} \sqcup \widehat{\text{Post}}^{\uparrow}[\tau_{6,12}](V_6)$

$= \bot \sqcup \widehat{\text{Post}}^{\uparrow}[\tau_{6,12}]([\mathtt{a} = \mathtt{b}, \mathtt{x} = \mathtt{y}])$

$lower' := \bot$

$\langle S, S' \rangle := \texttt{Model}(\widehat{\gamma}([\mathtt{a} = \mathtt{b}, \mathtt{x} = \mathtt{y}]) \wedge \tau_{6,6} \wedge \neg\widehat{\gamma}(\bot))$

$= \begin{bmatrix} a \mapsto 17, b \mapsto 17, x \mapsto 99, y \mapsto 99, \\ a' \mapsto 17, b' \mapsto 17, x' \mapsto 99, y' \mapsto 99 \end{bmatrix}$ // A satisfying concrete state-pair

$lower' := \bot \sqcup [\mathtt{a}' = 17, \mathtt{b}' = 17, \mathtt{x}' = 99, \mathtt{y}' = 99]$

$\langle S, S' \rangle := \texttt{Model}(\widehat{\gamma}([\mathtt{a} = \mathtt{b}, \mathtt{x} = \mathtt{y}]) \wedge \tau_{6,6} \wedge \neg\widehat{\gamma}([\mathtt{a}' = 17, \mathtt{b}' = 17, \mathtt{x}' = 99, \mathtt{y}' = 99]))$

$= \begin{bmatrix} a \mapsto 28, b \mapsto 28, x \mapsto 35, y \mapsto 35, \\ a' \mapsto 28, b' \mapsto 28, x' \mapsto 35, y' \mapsto 35 \end{bmatrix}$ // A satisfying concrete state-pair

$lower' := [\mathtt{a}' = 17, \mathtt{b}' = 17, \mathtt{x}' = 99, \mathtt{y}' = 99] \sqcup [\mathtt{a}' = 28, \mathtt{b}' = 28, \mathtt{x}' = 35, \mathtt{y}' = 35]$

$v' := [\mathtt{a}' = \mathtt{b}', \mathtt{x}' = \mathtt{y}']$

$V_{12} := \bot \sqcup [\mathtt{a} = \mathtt{b}, \mathtt{x} = \mathtt{y}] = [\mathtt{a} = \mathtt{b}, \mathtt{x} = \mathtt{y}]$

$\texttt{ans} := \langle \mathtt{V}_1 \mapsto \top, \mathtt{V}_6 \mapsto [\mathtt{a} = \mathtt{b}, \mathtt{x} = \mathtt{y}], \mathtt{V}_{12} \mapsto [\mathtt{a} = \mathtt{b}, \mathtt{x} = \mathtt{y}] \rangle$

Fixed Point!

Fig. 2. A possible chaotic-iteration sequence when a BII solver is invoked to find the best inductive affine-equality invariant for Eqn. (6). The parts of the trace enclosed in boxes show the actions that take place in calls to Alg. 1 ($\widehat{\text{Post}}^{\uparrow}$). (By convention, primes are dropped from the abstract value returned from a call on $\widehat{\text{Post}}^{\uparrow}$.)

"anytime algorithm".) In our case, the over-approximation serves as an "insurance policy" against being forced to return $\top$ in case the $\widehat{\text{Post}}$ algorithm runs out of time.

Alg. 2 shows our bilateral algorithm, called $\widehat{\text{Post}}^{\updownarrow}$. The differences between $\widehat{\text{Post}}^{\uparrow}$ and $\widehat{\text{Post}}^{\updownarrow}$ are highlighted in gray. Most concern the variables *upper*′ or p'. The requirements for $\widehat{\text{Post}}^{\updownarrow}$ are only slightly different from those for $\widehat{\text{Post}}^{\uparrow}$:

- Abstract domain $\mathcal{A}$ must be a lattice (i.e., with both meet and join) with a least element $\bot$ and a greatest element $\top$.
- `AbstractConsequence` must meet the following requirements:
  Let *lower*′ and *upper*′ be two $\mathcal{A}$ values such that *lower*′ $\sqsubseteq$ *upper*′. If $p' =$ `AbstractConsequence`(*lower*′, *upper*′), then $p' \sqsupseteq$ *lower*′ and $p' \not\sqsupseteq$ *upper*′.
  This property ensures that each iteration of the while-loop on lines 3–11 makes progress: The concrete state-pair $\langle S, S' \rangle$ is obtained by calling `Model`$(\widehat{\gamma}(v) \wedge \tau \wedge \neg\widehat{\gamma}(p'))$ (line 5). If the formula is not satisfiable, then *upper*′ is decreased by meeting it with p' (line 9); otherwise, *lower*′ is increased by joining it with $\beta(S')$ (line 11).
  There may be multiple values p' that satisfy the conditions for being an abstract consequence of *lower*′ and *upper*′. Furthermore, $\widehat{\text{Post}}^{\updownarrow}$ can be modified to pick a different abstract consequence if some choice leads to an SMT solver timeout. This modified algorithm, as well as an algorithm for computing the abstract consequence that is applicable to any conjunctive abstract domain, is described in [29, §3].
- It may appear that it is necessary for $\mathcal{A}$ to have no infinite descending chains (as well as no infinite ascending chains). However, we can modify the algorithm slightly to ensure that (i) it does not get trapped updating *upper*′ along an infinite descending chain, and that (ii) it exits when *lower*′ has converged to $\widehat{\text{Post}}[\tau](v)$. We can accomplish these goals by forcing the algorithm to perform the basic iteration step from $\widehat{\text{Post}}^{\uparrow}$ at least once every N iterations, for some fixed N. (See [30, App. C] for further discussion.)

$\widehat{\text{Post}}^{\updownarrow}$ initializes *upper*′ to $\top$ (line 1). A value for p' is obtained by calling `AbstractConsequence`(*lower*′, *upper*′) (line 4). The SMT solver is invoked by calling `Model`$(\widehat{\gamma}(v) \wedge \tau \wedge \neg\widehat{\gamma}(p'))$ (line 5). If the formula has a model $\langle S, S' \rangle$, Alg. 2 proceeds as in Alg. 1: *lower*′ $\leftarrow$ *lower*′ $\sqcup\, \beta(S')$. If the formula has no model, then $\widehat{\text{Post}}[\tau](v) \sqsubseteq p'$, and thus it is safe to update *upper*′ by performing a meet with p' (line 9). Moreover, this step is guaranteed to decrease the value of *upper*′ because $p' \not\sqsupseteq$ *upper*′.

When there is an SMT-solver timeout (line 6), $\widehat{\text{Post}}^{\updownarrow}$ returns *upper*′ as the answer (lines 7, 12, and 13). In this case, the value returned can be an over-approximation of the desired answer $\widehat{\text{Post}}[\tau](v)$; however, if the loop exits without a timeout, then *lower*′ must equal *upper*′, and the return value equals $\widehat{\text{Post}}[\tau](v)$.

Fig. 3 shows a possible trace of Iteration 2 from Fig. 2 when the call to $\widehat{\text{Post}}^{\uparrow}$ (Alg. 1) is replaced by a call to $\widehat{\text{Post}}^{\updownarrow}$ (Alg. 2). Note how a collection of individual, non-trivial, upper-bounding constraints are acquired on the second, third,

$$
\begin{aligned}
V_6 &:= V_6 \sqcup \widehat{\mathrm{Post}}^{\updownarrow}[\tau_{6,6}](V_6) \\
&= [\mathtt{a} = \mathtt{b}, \mathtt{x} = 0, \mathtt{y} = 0] \sqcup \widehat{\mathrm{Post}}^{\updownarrow}[\tau_{6,6}]([\mathtt{a} = \mathtt{b}, \mathtt{x} = 0, \mathtt{y} = 0]) \\
\hline
\mathit{upper}' &:= \top \\
\mathit{lower}' &:= \bot \\
p' &:= \texttt{AbstractConsequence}(\bot, \top) \\
&= \bot \\
\langle S, S' \rangle &:= \texttt{Model}(\widehat{\gamma}([\mathtt{a} = \mathtt{b}, \mathtt{x} = 0, \mathtt{y} = 0]) \wedge \tau_{6,6} \wedge \neg\widehat{\gamma}(\bot)) \\
&= \begin{bmatrix} a \mapsto 56, b \mapsto 56, x \mapsto 0, y \mapsto 0, \\ a' \mapsto 58, b' \mapsto 58, x' \mapsto 1, y' \mapsto 1 \end{bmatrix} \quad \begin{matrix} \text{// A satisfying concrete} \\ \text{// state-pair} \end{matrix} \\
\mathit{lower}' &:= \bot \sqcup [\mathtt{a}' = 58, \mathtt{b}' = 58, \mathtt{x}' = 1, \mathtt{y}' = 1] \\
p' &:= \texttt{AbstractConsequence}([\mathtt{a}' = 58, \mathtt{b}' = 58, \mathtt{x}' = 1, \mathtt{y}' = 1], \top) \\
&= [\mathtt{x}' = 1] \\
\langle S, S' \rangle &:= \texttt{Model}(\widehat{\gamma}([\mathtt{a} = \mathtt{b}, \mathtt{x} = 0, \mathtt{y} = 0]) \wedge \tau_{6,6} \wedge \neg\widehat{\gamma}([\mathtt{x}' = 1])) \\
&= \mathsf{None} \\
\mathit{upper}' &:= \top \sqcap [\mathtt{x}' = 1] = [\mathtt{x}' = 1] \\
p' &:= \texttt{AbstractConsequence}([\mathtt{a}' = 58, \mathtt{b}' = 58, \mathtt{x}' = 1, \mathtt{y}' = 1], [\mathtt{x}' = 1]) \\
&= [\mathtt{y}' = 1] \\
\langle S, S' \rangle &:= \texttt{Model}(\widehat{\gamma}([\mathtt{a} = \mathtt{b}, \mathtt{x} = 0, \mathtt{y} = 0]) \wedge \tau_{6,6} \wedge \neg\widehat{\gamma}([\mathtt{y}' = 1])) \\
&= \mathsf{None} \\
\mathit{upper}' &:= [\mathtt{x}' = 1] \sqcap [\mathtt{y}' = 1] = [\mathtt{x}' = 1, \mathtt{y}' = 1] \\
p' &:= \texttt{AbstractConsequence}([\mathtt{a}' = 58, \mathtt{b}' = 58, \mathtt{x}' = 1, \mathtt{y}' = 1], [\mathtt{x}' = 1, \mathtt{y}' = 1]) \\
&= [\mathtt{a}' = 58] \\
\langle S, S' \rangle &:= \texttt{Model}(\widehat{\gamma}([\mathtt{a} = \mathtt{b}, \mathtt{x} = 0, \mathtt{y} = 0]) \wedge \tau_{6,6} \wedge \neg\widehat{\gamma}([\mathtt{a}' = 58])) \\
&= \begin{bmatrix} a \mapsto 19, b \mapsto 19, x \mapsto 0, y \mapsto 0, \\ a' \mapsto 21, b' \mapsto 21, x' \mapsto 1, y' \mapsto 1 \end{bmatrix} \quad \begin{matrix} \text{// A satisfying concrete} \\ \text{// state-pair} \end{matrix} \\
\mathit{lower}' &:= [\mathtt{a}' = 58, \mathtt{b}' = 58, \mathtt{x}' = 1, \mathtt{y}' = 1] \sqcup [\mathtt{a}' = 21, \mathtt{b}' = 21, \mathtt{x}' = 1, \mathtt{y}' = 1] \\
&= [\mathtt{a}' = \mathtt{b}', \mathtt{x}' = 1, \mathtt{y}' = 1] \\
p' &:= \texttt{AbstractConsequence}([\mathtt{a}' = \mathtt{b}', \mathtt{x}' = 1, \mathtt{y}' = 1], [\mathtt{x}' = 1, \mathtt{y}' = 1]) \\
&= [\mathtt{a}' = \mathtt{b}'] \\
\langle S, S' \rangle &:= \texttt{Model}(\widehat{\gamma}([\mathtt{a} = \mathtt{b}, \mathtt{x} = 0, \mathtt{y} = 0]) \wedge \tau_{6,6} \wedge \neg\widehat{\gamma}([\mathtt{a}' = \mathtt{b}'])) \\
&= \mathsf{None} \\
\mathit{upper}' &:= [\mathtt{x}' = 1, \mathtt{y}' = 1] \sqcap [\mathtt{a}' = \mathtt{b}'] \\
&= [\mathtt{a}' = \mathtt{b}', \mathtt{x}' = 1, \mathtt{y}' = 1] \\
\mathit{lower}' \neq \mathit{upper}' &= \mathsf{false} \\
v' &:= [\mathtt{a}' = \mathtt{b}', \mathtt{x}' = 1, \mathtt{y}' = 1] \\
\hline
V_6 &:= [\mathtt{a} = \mathtt{b}, \mathtt{x} = 0, \mathtt{y} = 0] \sqcup [\mathtt{a} = \mathtt{b}, \mathtt{x} = 1, \mathtt{y} = 1] = [\mathtt{a} = \mathtt{b}, \mathtt{x} = \mathtt{y}] \\
\mathtt{ans} &:= \langle \mathtt{V}_1 \mapsto \top, \mathtt{V}_6 \mapsto [\mathtt{a} = \mathtt{b}, \mathtt{x} = \mathtt{y}], \mathtt{V}_{12} \mapsto \bot \rangle
\end{aligned}
$$

Fig. 3. A possible trace of Iteration 2 from Fig. 2 when the call to $\widehat{\mathrm{Post}}^{\uparrow}$ (Alg. 1) is replaced by a call to $\widehat{\mathrm{Post}}^{\updownarrow}$ (Alg. 2).

and fifth calls to `AbstractConsequence`: $[\mathtt{x}' = 1]$, $[\mathtt{y}' = 1]$, and $[\mathtt{a}' = \mathtt{b}']$, respectively. By these means, *upper*′ works its way down the chain $\top \sqsupset [\mathtt{x}' = 1] \sqsupset [\mathtt{x}' = 1, \mathtt{y}' = 1] \sqsupset [\mathtt{a}' = \mathtt{b}', \mathtt{x}' = 1, \mathtt{y}' = 1]$. After each call to `AbstractConsequence`,

the abstract-consequence constraint is tested to see if it really is an upper bound on the answer. For instance, the fourth call to `AbstractConsequence` returns $[\mathtt{a}' = 58]$. The assertion that $[\mathtt{a}' = 58]$ is an upper-bounding constraint is refuted by the concrete state-pair

$$\langle S, S' \rangle = \left[\begin{array}{c} a \mapsto 19, b \mapsto 19, x \mapsto 0, y \mapsto 0, \\ a' \mapsto 21, b' \mapsto 21, x' \mapsto 1, y' \mapsto 1 \end{array} \right],$$

which is used to improve the value of *lower*′.

The important point is that if Iteration 2 is taking too much time, $\widehat{\text{Post}}^{\updownarrow}$ can be stopped and *upper*′ returned as the answer. In contrast, if $\widehat{\text{Post}}^{\uparrow}$ is stopped because it is taking too much time, the only safe answer that can be returned is $\top$. The "can-be-stopped-anytime" property of $\widehat{\text{Post}}^{\updownarrow}$ can make a significant difference in the final answer. For instance, suppose that $\widehat{\text{Post}}^{\uparrow}$ and $\widehat{\text{Post}}^{\updownarrow}$ both stop early during Iteration 2 (Figs. 2 and 3, respectively), and that $\widehat{\text{Post}}^{\updownarrow}$ returns $[\mathtt{x} = \mathtt{1}, \mathtt{y} = \mathtt{1}]$, whereas $\widehat{\text{Post}}^{\uparrow}$ returns $\top$. Assuming no further timeouts take place during the evaluation of Eqn. (6), the respective final answers would be

$$\widehat{\text{Post}}^{\uparrow} : \langle \mathtt{v}_1 \mapsto \top, \mathtt{v}_6 \mapsto \top, \mathtt{v}_{12} \mapsto \top \rangle$$
$$\widehat{\text{Post}}^{\updownarrow} : \langle \mathtt{v}_1 \mapsto \top, \mathtt{v}_6 \mapsto [\mathtt{x} = \mathtt{y}], \mathtt{v}_{12} \mapsto [\mathtt{x} = \mathtt{y}] \rangle$$

Because of the timeout, the answer computed by $\widehat{\text{Post}}^{\updownarrow}$ is not the *best* inductive affine-equality invariant; however, the answer establishes that the equality constraint $[\mathtt{x} = \mathtt{y}]$ holds at both lines 6 and 12 of Fig. 1(a).

Attaining the Best Inductive $\mathcal{A}$-Invariant.

Lemma 2.1 *The least fixed-point of Eqn. (1) (the best $\mathcal{A}$-transformer equations of a transition system) is the best inductive invariant expressible in $\mathcal{A}$.*

Corollary 2.2 *Applying an equation solver to the best $\mathcal{A}$-transformer equations, using either $\widehat{\text{Post}}^{\uparrow}$ or $\widehat{\text{Post}}^{\updownarrow}$ to evaluate equation right-hand sides, finds the best inductive $\mathcal{A}$-invariant if there are no timeouts during the evaluation of any right-hand side.*

3 Best Inductive Invariants and Interprocedural Analysis

In this section, we present a method for solving the BII problem for multi-procedure programs. Our framework is similar to the so-called "functional approach" to interprocedural analysis of Sharir and Pnueli [28] (denoted by *SP*), which works with an abstract domain that abstracts transition relations. For instance, our approach

- also uses an abstract domain that abstracts transition relations, and
- creates a *summary transformer* for each reachable procedure P, which over-approximates the transition relation of P.

However, to make the symbolic-abstraction approach suitable for interprocedural analysis, the algorithm uses a generalization of $\widehat{\text{Post}}$, called $\widehat{\text{Compose}}[\tau](a)$, where

$\tau \in \mathcal{L}[\boldsymbol{v};\boldsymbol{v}']$ and $a \in \mathcal{A}[\boldsymbol{v};\boldsymbol{v}']$ both represent transition relations over the program variables $\boldsymbol{v}$. The goal of $\widehat{\text{Compose}}[\tau](a)$ is to create an $\mathcal{A}[\boldsymbol{v};\boldsymbol{v}']$ value that is the best over-approximation of a's action followed by τ's action. Furthermore, instead of Eqn. (1), we find the least solution to Eqns. (7)–(11) below, where each application of the right-hand side of an equation is given the *best-transformer interpretation*—in this case, by means of $\widehat{\text{Compose}}$.

A *program* is defined by a set of procedures P_i, $0 \le i \le K$, and represented by an interprocedural control-flow graph $G = (N, F)$. G consists of a collection of intraprocedural control-flow graphs $G_1, G_2, \ldots, G_K$, one of which, G_{main}, represents the program's main procedure. The node set N_i of $G_i = (N_i, F_i)$ is partitioned into five disjoint subsets: $N_i = E_i \uplus X_i \uplus C_i \uplus R_i \uplus L_i$. G_i contains exactly one *enter* node (i.e., $E_i = \{e_i\}$) and exactly one *exit* node (i.e., $X_i = \{x_i\}$). A procedure call in G_i is represented by two nodes, a *call* node $c \in C_i$ and a *return-site* node $r \in R_i$, and has two edges: (i) a *call-to-enter* edge from c to the enter node of the called procedure, and (ii) an *exit-to-return-site* edge from the exit node of the called procedure to r. The functions *call* and *ret* record matching call and return-site nodes: $\mathit{call}(r) = c$ and $\mathit{ret}(c) = r$. We assume that an enter node has no incoming edges except call-to-enter edges.

$$\phi(e_{main}) = Id|_{a_1} \qquad a_1 \in \mathcal{A} \text{ describes the set of initial stores at } e_{main} \tag{7}$$

$$\phi(e_p) = Id|_a \qquad e_p \in E,\ p \neq \mathit{main}, \text{ and } a = \bigsqcup_{c \in C,\, c \text{ calls } p} V_c \tag{8}$$

$$\phi(n) = \bigsqcup_{m \to n \in F} \widehat{\text{Compose}}[\tau_{m,n}, \phi(m)] \qquad \text{for } n \in N,\ n \notin (R \cup E) \tag{9}$$

$$\phi(n) = \widehat{\text{Compose}}[\widehat{\gamma}(\phi(x_q)), \phi(\mathit{call}(n))] \qquad \text{for } n \in R, \text{ and } \mathit{call}(n) \text{ calls } q \tag{10}$$

$$V_n = \text{range}(\phi(n)) \tag{11}$$

The equations involve two sets of "variables": $\phi(n)$ and V_n, where $n \in N$. $\phi(n)$ is a partial function that represents a summary of the transformation from $e_{proc(n)}$ to n. $Id|_a$ denotes the identity transformer restricted to inputs in $a \in \mathcal{A}$. The domain of $\phi(n)$ over-approximates the set of reachable states at $e_{proc(n)}$ from which it is possible to reach n; the range of $\phi(n)$ over-approximates the set of reachable states at n. V_n's value equals the range of $\phi(n)$.

$\widehat{\text{Compose}}^{\updownarrow}$ is essentially identical to Alg. 2, except that in line 5, $\widehat{\text{Compose}}^{\updownarrow}$ performs a query using a three-state formula,

$$\langle S, S', S'' \rangle \leftarrow \texttt{Model}(\widehat{\gamma}_{[\boldsymbol{v},\boldsymbol{v}']}(a) \wedge \tau_{[\boldsymbol{v}',\boldsymbol{v}'']} \wedge \neg \widehat{\gamma}_{[\boldsymbol{v},\boldsymbol{v}'']}(p')),$$

and in line 11, $\widehat{\text{Compose}}^{\updownarrow}$ applies a two-state version of β to S and S'', dropping S' completely: $\mathit{lower}' \leftarrow \mathit{lower}' \sqcup \beta(S, S'')$. ($\widehat{\text{Compose}}^{\uparrow}$ is defined similarly.)

An important difference between our algorithm and the *SP* algorithm is that in our algorithm, the initial abstract value for the enter node e_p for procedure p is specialized to the reachable inputs of p (see Eqn. (8)). In the *SP* algorithm, $\phi(e_p)$ is always set to *Id*. Fig. 4 illustrates the effect of specializing the abstract pre-

```
(1)  main() {
(2)      x = read_input();
(3)      while(x != 1) {
(4)          if(even(x)) halve();
(5)          else increase();
(6)      }
(7)  }
(8)  void halve() {
(9)      x = x ≫ 1;
(10) }
(11)
(12) void increase() {
(13)     x = 3*x + 1;
(14) }
```

		Abstract value at enter node e_p	
		Id	$Id\|_a$, where $a = \bigsqcup_{c \in C,\, c \text{ calls } p} V_c$
halve	pre-condition (e_{halve})	$x' = x$	$2^{31}x = 0 \wedge x' = x$
	post-condition (x_{halve})	$\top$	$2^{31}x = 0 \wedge x - 2x' = 0$
increase	pre-condition (e_{increase})	$x' = x$	$2^{31}x = 1 \wedge x' = x$
	post-condition (x_{increase})	$x' = 3x + 1$	$2^{31}x = 1 \wedge 2^{31}x' = 0 \wedge x' = 3x + 1$

Fig. 4. The effect of specializing the abstract pre-condition at enter node e_p, and the resulting strengthening of the inferred abstract post-condition.

condition at enter node e_p on the inferred abstract post-condition. The abstract domain used in Fig. 4 is the domain of affine equalities over machine integers [8]. With that domain, it is possible to express that a 32-bit variable x holds an even number: $2^{31}x = 0$. Consequently, the initial abstract value for enter node e_{halve} is the identity relation, constrained so that x is even. Similarly, the initial abstract value for enter node e_{increase} is the identity relation, constrained so that x is odd.

Note that the abstract value at the exit point x_p of a procedure p serves as a procedure summary—i.e., an abstraction of p's transition relation. Fig. 4 shows that by giving halve and increase more precise abstract values at the respective enter nodes, we obtained more precise procedure summaries at the respective exit points. In particular, for halve, the constraint $x - 2x' = 0$ provides a good characterization of the effect of a right-shift operation, but only when x is known to be even (cf. the entries for halve's post-condition in columns 3 and 4 of Fig. 4).

4 Related Work

Previous work related to BII falls into two categories:

(i) Work on $\widehat{\text{Post}}$ [13,23,25,35,26,17,2,8,21,33,29], which, as shown by Obs. 1 and Eqn. (1), provides a solution to the *intra*procedural BII problem.

(ii) Work specifically on the BII problem itself [11,34,12].

Much of the work in item (i) could be used for the *inter*procedural BII problem by (a) generalizing those algorithms to perform $\widehat{\text{Compose}}$, and (b) using them to solve Eqns. (7)–(11).

Houdini [11] is the first algorithm that we are aware of that solves a version of the BII problem. The paper on Houdini does not describe the work in terms of abstract interpretation. Santini [31, §5] was directly inspired by Houdini, as an effort to broaden Houdini's range of applicability.

Yorsh et al. [34] introduced a particularly interesting technique. Their algorithm for the BII problem can be viewed as basically solving Eqn. (1) using $\widehat{\text{Post}}^{\uparrow}$. How-

ever, they observed that it is not necessary to rely on calls to an SMT solver for *all* of the concrete states used by $\widehat{\text{Post}}^{\uparrow}$; instead, they used concrete execution of the program as a way to generate concrete states very cheaply. If for some program point q of interest they have state-set S_q, they obtain an under-approximation for the abstract value V_q by performing $V_q = \sqcup\{\beta(\sigma) \mid \sigma \in S_q\}$. This idea is similar in spirit to the computation of candidate invariants from execution information by Daikon [10]. Because $\widehat{\text{Post}}^{\updownarrow}$ works simultaneously from below and from above, the Yorsh et al. heuristic can be used to improve the speed of convergence of *lower*′ in line 11 of Alg. 2.

If we think of $\tau = \langle \ldots, \tau_{i,j}, \ldots \rangle$ as a monolithic transformer, an alternative way of stating the objective of intraprocedural BII is as follows:

- Given a concrete transformer τ and an abstract value $a \in \mathcal{A}$ that over-approximates the set of initial states, apply the best abstract transformer for τ^* to a (i.e., apply $\widehat{\text{Post}}[\tau^*](a)$).

This problem was the subject of a recent technical report by Garoche et al. [12].

Several of the techniques used in §2 and §3 are inspired by previously known methods. Alg. 1 is a variant of an algorithm due to Reps et al. [25, §5]. Alg. 2 adopts the ideas used in an anytime algorithm for symbolic abstraction [29, §3]. In interprocedural dataflow analysis, the idea of specializing the abstract value of an enter node has been used in a few earlier algorithms [24,27,15,14].

Recent work has also explored connections between abstract interpretation and decision procedures [32,33,6,7]. In particular, D'Silva et al. [7] generalize the algorithm for Conflict Driven Clause Learning used in SAT solvers to solve the lattice-theoretic problem of determining if an additive transformer on a Boolean lattice is always bottom. In contrast, our algorithms ([32,33], Eqn. (1), and §3) address a broader class of problems. Our algorithms apply to non-Boolean lattices. Moreover, provided there are no timeouts, our algorithms are capable of discovering if the most-precise answer is $\bot$. However, they are also useful when the most-precise answer is not $\bot$; in particular, our algorithms can be used to compute best transformers. Furthermore, our algorithms can be used to solve the BII problem (assuming no timeouts), and even when there are timeouts, they generate inductive program invariants.

The last two authors have worked on a system, called TSL (for "**T**ransformer **S**pecification **L**anguage") [19,20], that also aims to automate the creation of abstract-interpretation systems (primarily for machine-code analysis). TSL is based on a different approach to creating abstract transformers than symbolic abstract interpretation, but shares the property that the user first expresses the full semantics of the programming language of interest. With TSL, one specifies an instruction set's concrete operational semantics by writing an interpreter using a first-order functional language. The interpreter that specifies how each instruction transforms a concrete state. To define an abstract interpretation for an abstract domain $\mathcal{A}$, one defines "replacement" datatypes for TSL's numeric/bit-vector and map datatypes, along with 42 replacement numeric/bit-vector operations, 12 replacement map-access/update operations, plus a few additional operations, such as $\sqcup$, $\sqcap$, and widen.

From such a reinterpretation of the TSL meta-language, which is extended automatically to TSL expressions and functions, TSL creates sound over-approximating transformers. Standard analysis algorithms can then use the transformers to obtain inductive $\mathcal{A}$-invariants. However, there is generally some loss of precision with the reinterpretation approach, so the resulting analyses generally do not find *best* inductive $\mathcal{A}$-invariants.

5 Conclusion

Our experience is that the abstract-interpretation community is either unfamiliar with, or under-appreciates, the power and utility of symbolic abstraction. In this paper, we have explained how symbolic abstraction allows abstract interpreters to be created that are correct by construction: from a specification of the concrete semantics in logic, and a few operations on values in abstract domain $\mathcal{A}$, we automatically obtain a solution to the BII problem: "Given program P, and an abstract domain $\mathcal{A}$, find the most-precise inductive $\mathcal{A}$-invariant for P."

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

- The BII problem reduces to the problem of applying $\widehat{\text{Post}}$.
- The problem of applying $\widehat{\text{Post}}$ reduces to the problem of symbolic abstraction.
- The algorithm for symbolic abstraction is based on a (sophisticated) iterative sampling process, using an SMT solver. Blocking clauses are used to push the solver to explore a new part of the state space on each iteration.

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

References

[1] Ball, T. and S. Rajamani, *Bebop: A symbolic model checker for Boolean programs*, in: *Spin Workshop*, 2000.

[2] Brauer, J. and A. King, *Automatic abstraction for intervals using Boolean formulae*, in: *SAS*, 2010.

[3] Cousot, P. and R. Cousot, *Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in: *POPL*, 1977.

[4] Cousot, P. and R. Cousot, *Systematic design of program analysis frameworks*, in: *POPL*, 1979.

[5] Cousot, P. and N. Halbwachs, *Automatic discovery of linear constraints among variables of a program*, in: *POPL*, 1978.

[6] D'Silva, V., L. Haller and D. Kroening, *Satisfiability solvers are static analyzers*, in: *SAS*, 2012.

[7] D'Silva, V., L. Haller and D. Kroening, *Abstract conflict driven learning*, in: *POPL*, 2013.

[8] Elder, M., J. Lim, T. Sharma, T. Andersen and T. Reps, *Abstract domains of affine relations*, in: *SAS*, 2011.

[9] Elder, M., J. Lim, T. Sharma, T. Andersen and T. Reps, *Abstract domains of affine relations*, TR 1792, CS Dept., Univ. of Wisconsin, Madison, WI (2013).

[10] Ernst, M., J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz and C. Xiao, *The Daikon system for dynamic detection of likely invariants*, SCP **69** (2007).

[11] Flanagan, C. and K. R. M. Leino, *Houdini, an annotation assistant for Esc/Java*, in: *FME*, 2001.

[12] Garoche, P.-L., T. Kahsai and C. Tinelli, *Invariant stream generators using automatic abstract transformers based on a decidable logic*, Tech. Rep. CoRR abs/1205.3758, arXiv (2012).

[13] Graf, S. and H. Saïdi, *Construction of abstract state graphs with PVS*, in: *CAV*, 1997.

[14] Jeannet, B., A. Loginov, T. Reps and M. Sagiv, *A relational approach to interprocedural shape analysis*, in: *SAS*, 2004.

[15] Jeannet, B. and W. Serwe, *Abstracting call-stacks for interprocedural verification of imperative programs*, in: *AMAST*, 2004.

[16] Kidd, N., A. Lal and T. Reps, *WALi: The Weighted Automaton Library* (2007), www.cs.wisc.edu/wpis/wpds/download.php.

[17] King, A. and H. Søndergaard, *Automatic abstraction for congruences*, in: *VMCAI*, 2010.

[18] Lal, A., S. Qadeer and S. K. Lahiri, *A solver for reachability modulo theories*, in: *CAV*, 2012.

[19] Lim, J. and T. Reps, *A system for generating static analyzers for machine instructions*, in: *CC*, 2008.

[20] Lim, J. and T. Reps, *Tsl: A system for generating abstract interpreters and its application to machine-code analysis*, Trans. on Prog. Lang. and Syst. **35** (2013), pp. 4:1–4:59.

[21] Monniaux, D., *Automatic modular abstractions for template numerical constraints*, LMCS **6** (2010).

[22] Müller-Olm, M. and H. Seidl, *Precise interprocedural analysis through linear algebra*, in: *POPL*, 2004.

[23] Regehr, J. and A. Reid, *HOIST: A system for automatically deriving static analyzers for embedded systems*, in: *ASPLOS*, 2004.

[24] Reps, T., S. Horwitz and M. Sagiv, *Precise interprocedural dataflow analysis via graph reachability*, in: *POPL*, 1995.

[25] Reps, T., M. Sagiv and G. Yorsh, *Symbolic implementation of the best transformer*, in: *VMCAI*, 2004.

[26] Sankaranarayanan, S., H. Sipma and Z. Manna, *Scalable analysis of linear systems using mathematical programming*, in: *VMCAI*, 2005.

[27] Schwoon, S., "Model-Checking Pushdown Systems," Ph.D. thesis, Technical Univ. of Munich, Munich, Germany (2002).

[28] Sharir, M. and A. Pnueli, *Two approaches to interprocedural data flow analysis*, in: *Program Flow Analysis: Theory and Applications*, Prentice-Hall, 1981 .

[29] Thakur, A., M. Elder and T. Reps, *Bilateral algorithms for symbolic abstraction*, in: *SAS*, 2012.

[30] Thakur, A., M. Elder and T. Reps, *Bilateral algorithms for symbolic abstraction*, TR 1713, CS Dept., Univ. of Wisconsin, Madison, WI (2012).

[31] Thakur, A., A. Lal, J. Lim and T. Reps, *Posthat and all that: Attaining most-precise inductive invariants*, TR 1790, CS Dept., Univ. of Wisconsin, Madison, WI (2013).

[32] Thakur, A. and T. Reps, *A Generalization of Stålmarck's Method*, in: *SAS*, 2012.

[33] Thakur, A. and T. Reps, *A method for symbolic computation of precise abstract operations*, in: *CAV*, 2012.

[34] Yorsh, G., T. Ball and M. Sagiv, *Testing, abstraction, theorem proving: Better together!*, in: *ISSTA*, 2006.

[35] Yorsh, G., T. Reps and M. Sagiv, *Symbolically computing most-precise abstract operations for shape analysis*, in: *TACAS*, 2004.