Computer Sciences Department

Symbolic Implementation of the Best Transformer

Thomas Reps
Mooly Sagiv
Greta Yorsh

Technical Report #1468

January 2003

UNIVERSITY OF
WISCONSIN
MADISON

Symbolic Implementation of the Best Transformer

Thomas Reps[1], Mooly Sagiv[2], and Greta Yorsh[2]

[1] Comp. Sci. Dept., University of Wisconsin; reps@cs.wisc.edu
[2] School of Comp. Sci., Tel-Aviv University; {msagiv,gretay}@post.tau.ac.il

Abstract. This paper shows how to achieve, under certain conditions, abstract-interpretation algorithms that enjoy the best possible precision for a given abstraction. The key idea is a simple process of successive approximation that makes repeated calls to a theorem prover, and obtains the best abstract value for a set of concrete stores that are represented symbolically, using a logical formula.

1 Introduction

Abstract interpretation [8] is a well-established technique for automatically proving certain program properties. In abstract interpretation, sets of program stores are represented in a conservative manner by abstract values. Each program statement is given an interpretation over abstract values that is conservative with respect to its interpretation over corresponding sets of concrete stores; that is, the result of "executing" a statement must be an abstract value that describes a superset of the concrete stores that actually arise. This methodology guarantees that the results of abstract interpretation overapproximate the sets of concrete stores that actually arise at each point in the program.

In [9], it is shown that, under certain reasonable conditions, it is possible to give a *specification* of the most-precise abstract interpretation for a given abstract domain. For a Galois connection defined by abstraction function α and concretization function γ, the best abstract post operator for transition τ, denoted by $\mathrm{Post}^\sharp[\tau]$, can be expressed in terms of the concrete post operator for τ, $\mathrm{Post}[\tau]$, as follows:

$$\mathrm{Post}^\sharp[\tau] = \alpha \circ \mathrm{Post}[\tau] \circ \gamma. \tag{1}$$

This defines the limit of precision obtainable using a given abstraction. However, Eqn. (1) is non-constructive; it does not provide an *algorithm* for finding or applying $\mathrm{Post}^\sharp[\tau]$.

Graf and Saïdi [13] showed that theorem provers can be used to generate best abstract transformers for abstract domains that are fixed, finite, Cartesian products of Boolean values. (The use of such domains is known as *predicate abstraction*; predicate abstraction is also used in SLAM [1] and other systems [10].) The work presented in this paper shows how some of the benefits enjoyed by applications that use the predicate-abstraction approach can also be enjoyed by applications that use abstract domains that are not amenable to the use of predicate abstraction. In particular, this paper's results apply to arbitrary finite-height abstract domains, not just to Cartesian products of Booleans. For example, it applies to the abstract domains used for constant propagation and common-subexpression elimination [16]. When applied to a predicate-abstraction domain, the method has the same worst-case complexity as the Graf-Saïdi method.

To understand where the difficulties lie, consider how they are addressed in predicate abstraction. In general, the result of applying γ to an abstract value l is an infinite set of concrete stores; Graf and Saïdi sidestep this difficulty by performing γ symbolically, expressing the result of $\gamma(l)$ as a formula φ. They then introduce a function that, in effect, is the composition of α and $\mathrm{Post}[\tau]$: it applies $\mathrm{Post}[\tau]$ to φ and maps the result back to the abstract domain. In other words, Eqn. (1) is recast using two functions that work at the symbolic level, $\widehat{\gamma}$ and $\widehat{\alpha\mathrm{Post}}$,[3] such that $\widehat{\alpha\mathrm{Post}}[\tau] \circ \widehat{\gamma} = \alpha \circ \mathrm{Post}[\tau] \circ \gamma$.

To provide insight on what opportunities exist as we move from predicate-abstraction domains to the more general class of finite-height lattices, we first address a simpler problem than $\widehat{\alpha\mathrm{Post}}[\tau]$, namely,

> How can $\widehat{\alpha}$ be implemented? That is, how can one identify the most-precise abstract value of a given abstract domain that overapproximates a set of concrete stores that are represented symbolically?

We then employ the basic idea used in $\widehat{\alpha}$ to implement our own version of $\widehat{\alpha\mathrm{Post}}[\tau]$.

The remainder of the paper is organized as follows: Sect. 2 motivates the work by presenting an $\widehat{\alpha}$ procedure for a specific finite-height lattice. Sect. 3 introduces terminology and notation. Sect. 4 presents the general treatment of $\widehat{\alpha}$ procedures for finite-height lattices. Sect. 5 discusses symbolic techniques for implementing transfer functions (i.e., $\widehat{\alpha\mathrm{Post}}[\tau]$). Sect. 6 discusses how to extract information from an abstract value by evaluating a query. Sect. 7 makes some additional observations about the work. Sect. 8 discusses related work. App. A contains a proof of one of the results. (Other proofs have been omitted to conserve space.)

[3] In general, the diacritic $\widehat{\ }$ on a symbol indicates an operation that either produces or operates on a symbolic representation of a set of concrete stores.

2 Motivating Examples

This section presents a sequence of examples at a semi-formal level to motivate the work. A formal treatment is given in later sections. The examples concern a familiar pair of concrete and abstract domains, namely, the pair used in the constant-propagation problem: let *Var* denote the set of variables in the program being analyzed; the concrete domain is $2^{Var \to Z}$; the abstract domain is $(Var \to Z^{\top})_{\bot}$. The abstract value $\bot$ represents $\emptyset$; an abstract value such as $[\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0]$ represents all concrete stores in which program variables x and z are both mapped to 0.[4]

Predicate Abstraction A predicate-abstraction domain is based on some predicate set $\{B_j \stackrel{\text{def}}{=} \varphi_j \mid 1 \leq j \leq k\}$; however, the abstract domain $(Var \to Z^{\top})_{\bot}$ cannot be captured by a fixed, finite set of predicates, even if there is a Boolean predicate $B \stackrel{\text{def}}{=} (x = c)$ for each $\mathtt{x} \in Var$ and each distinct constant c that appears in the program. For instance, if the program is

$$\begin{array}{l} \mathtt{y} := 3 \\ \mathtt{x} := 4 * \mathtt{y} + 1 \end{array} \tag{2}$$

the predicate-abstraction domain based on the predicate set $\{B_1 \stackrel{\text{def}}{=} (y = 1), B_2 \stackrel{\text{def}}{=} (y = 3), B_3 \stackrel{\text{def}}{=} (y = 4), B_4 \stackrel{\text{def}}{=} (x = 1), B_5 \stackrel{\text{def}}{=} (x = 3), B_6 \stackrel{\text{def}}{=} (x = 4)\}$ does not provide an exact representation of the final state that arises, $[x \mapsto 13, y \mapsto 3]$. The best that can be done is to use the monomial[5]

$$\neg B_1 \wedge B_2 \wedge \neg B_3 \wedge \neg B_4 \wedge \neg B_5 \wedge \neg B_6,$$

which provides limited information about the value of x.

The phenomenon illustrated here is well known: the predicate-abstraction domain in which there is a Boolean predicate $B \stackrel{\text{def}}{=} (x = c)$ for each $\mathtt{x} \in Var$ and each distinct constant c that appears in the program is sufficient for copy-constant propagation, in which the only statements interpreted exactly are of the form x := c and x := y [12]; however, this domain is not sufficient for versions of constant propagation in which statements of the form x := a * y + b and x := y * z are interpreted exactly [20].

The α Function for Predicate-Abstraction Domains One of the virtues of the predicate-abstraction method is that it provides a procedure to obtain a most-precise abstract value (i.e., a monomial over the B_j), given (a specification of) a set of concrete stores as a logical formula ψ [13]. We will call this procedure $\widehat{\alpha}_{\mathrm{PA}}$;[6] it relies on the aid of a theorem prover, and can be defined as follows:

$$\widehat{\alpha}_{\mathrm{PA}}(\psi) = \begin{cases} \mathbf{ff} & \text{if } \psi \text{ is unsatisfiable} \\ \displaystyle\bigwedge_{j=1}^{k} \begin{cases} B_j & \text{if } \psi \Rightarrow \varphi_j \text{ is valid} \\ \neg B_j & \text{if } \psi \Rightarrow \neg\varphi_j \text{ is valid} \\ \mathbf{tt} & \text{otherwise} \end{cases} & \text{otherwise} \end{cases} \tag{3}$$

For instance, suppose that ψ is the formula $(y = 3) \wedge (x = 4 * y + 1)$, which captures the final state of program (2). For $\widehat{\alpha}_{\mathrm{PA}}((y = 3) \wedge (x = 4 * y + 1))$ to produce the answer $\neg B_1 \wedge B_2 \wedge \neg B_3 \wedge \neg B_4 \wedge \neg B_5 \wedge \neg B_6$, the theorem prover must be capable of demonstrating that the following formulas are valid:

$$\begin{array}{ll} (y = 3) \wedge (x = 4 * y + 1) \Rightarrow \neg(y = 1) & (y = 3) \wedge (x = 4 * y + 1) \Rightarrow \neg(x = 1) \\ (y = 3) \wedge (x = 4 * y + 1) \Rightarrow (y = 3) & (y = 3) \wedge (x = 4 * y + 1) \Rightarrow \neg(x = 3) \\ (y = 3) \wedge (x = 4 * y + 1) \Rightarrow \neg(y = 4) & (y = 3) \wedge (x = 4 * y + 1) \Rightarrow \neg(x = 4) \end{array} \tag{4}$$

This is depicted in Fig. 1.

Going Beyond Predicate Abstraction We now show that the ability to implement the α function of a Galois connection between a concrete and abstract domain is not limited to predicate-abstraction domains.

---

[4] We write abstract values in Courier typeface (e.g., $[\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0]$), and concrete stores in Roman typeface (e.g., $[x \mapsto 0, y \mapsto 43, z \mapsto 0]$).

[5] A *monomial* over $B_1, \ldots, B_k$ is a conjunction of B_j's, $\neg B_j$'s, and occurrences of **tt** in which each B_j appears at most once. The term **ff** is also a monomial.

[6] We refer to operations like $\widehat{\alpha}_{\mathrm{PA}}$ as "procedures" rather than as "algorithms" because they make calls to a theorem prover. Depending on the logic employed, and the power of the theorem prover being used, $\widehat{\alpha}_{\mathrm{PA}}$ may or may not terminate. It yields an algorithm in cases where the logic is decidable (e.g., weak monadic second-order logic and decidable fragments of first-order logic) and the theorem prover incorporates a decision procedure for the logic.

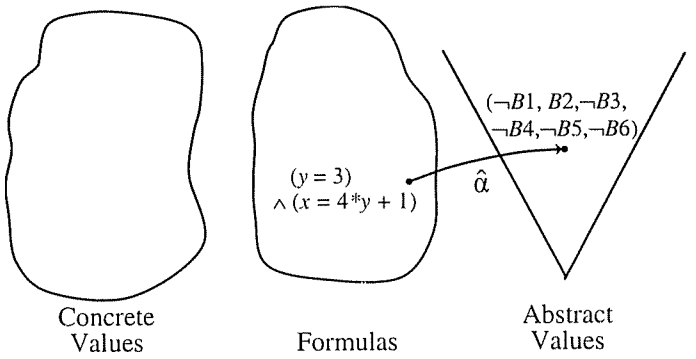

Fig. 1. $\widehat{\alpha}_{\mathrm{PA}}((y = 3) \wedge (x = 4 * y + 1))$ produces the abstract value $(\neg B_1, B_2, \neg B_3, \neg B_4, \neg B_5, \neg B_6)$, which denotes the monomial $\neg B_1 \wedge B_2 \wedge \neg B_3 \wedge \neg B_4 \wedge \neg B_5 \wedge \neg B_6$, by using a theorem prover to establish the formulas given in Eqn. (4). Note that $\widehat{\alpha}_{\mathrm{PA}}$ makes no direct use of the space of concrete values (cf. Figs. 3(a)–3(d)).

In contrast to the predicate-abstraction domain $\{B_1, B_2, B_3, B_4, B_5, B_6\}$, the constant-propagation domain $(Var \to \mathcal{Z}^\top)_\perp$ does allow the final state of program (2) to be represented exactly, and the procedure that we give, which we call $\widehat{\alpha}_{\mathrm{CP}}$, is capable of finding it. $\widehat{\alpha}_{\mathrm{CP}}$ is actually an instance of the general procedure for implementing $\widehat{\alpha}$ functions of Galois connections presented in Fig. 2: $\widehat{\alpha}_{\mathrm{CP}}$ is the instance in which the return type L is $(Var \to \mathcal{Z}^\top)_\perp$, and "structure" in line [5] means "concrete store".

```
[1]  L α̂(formula ψ) {
[2]    ans := ⊥
[3]    φ := ψ
[4]    while (φ is satisfiable) {
[5]      Select a structure S such that S ⊨ φ
[6]      ans := ans ⊔ β(S)
[7]      φ := φ ∧ ¬γ̂(ans)
[8]    }
[9]    return ans
[10] }
```

Fig. 2. An algorithm to obtain, with the aid of a theorem prover, a most-precise abstract value that overapproximates a set of concrete stores. In Sect. 2, the return type L is $(Var \to \mathcal{Z}^\top)_\perp$, and "structure" in line [5] means "concrete store".

As with procedure $\widehat{\alpha}_{\mathrm{PA}}$, $\widehat{\alpha}_{\mathrm{CP}}$ is permitted to make calls to a theorem prover (see line [5] of Fig. 2). We make one assumption that goes beyond what is assumed in predicate abstraction, namely, we assume that the theorem prover is a satisfiability checker that is capable of returning a satisfying assignment, or, equivalently, that it is a validity checker that returns a counterexample. (In the latter case, the counterexample obtained by calling `ProveValid(¬φ)` is a suitable satisfying assignment.)

The other operations used in procedure $\widehat{\alpha}_{\mathrm{CP}}$ are β, $\sqcup$, and $\widehat{\gamma}$:

- The concrete and abstract domains are related by a Galois connection defined by a representation function β that maps a concrete store $S \in Var \to \mathcal{Z}$ to an abstract value $\beta(S) \in (Var \to \mathcal{Z}^\top)_\perp$. For instance, β maps the concrete store $[x \mapsto 13, y \mapsto 3]$ to the abstract value $[\mathrm{x} \mapsto 13, \mathrm{y} \mapsto 3]$.
- $\sqcup$ is the join operation in $(Var \to \mathcal{Z}^\top)_\perp$. For instance,

$$[\mathrm{x} \mapsto 0, \mathrm{y} \mapsto 43, \mathrm{z} \mapsto 0] \sqcup [\mathrm{x} \mapsto 0, \mathrm{y} \mapsto 46, \mathrm{z} \mapsto 0] = [\mathrm{x} \mapsto 0, \mathrm{y} \mapsto \top, \mathrm{z} \mapsto 0].$$

- There is an operation $\widehat{\gamma}$ that maps an abstract value l to a formula $\widehat{\gamma}(l)$ such that l and $\widehat{\gamma}(l)$ represent the same set of concrete stores. For instance, we have

$$\widehat{\gamma}([\mathrm{x} \mapsto 0, \mathrm{y} \mapsto \top, \mathrm{z} \mapsto 0]) = (x = 0) \wedge (z = 0).$$

  The resulting formula contains no term involving y because $\mathrm{y} \mapsto \top$ does not place any restrictions on the value of y.

  Operation $\widehat{\gamma}$ permits the concretization of an abstract store to be represented symbolically, using a logical formula. This allows sets of concrete stores to be manipulated symbolically, via operations on formulas.

Given a specification of a set of concrete stores as a logical formula ψ, both $\widehat{\alpha}_{\mathrm{PA}}(\psi)$ and $\widehat{\alpha}_{\mathrm{CP}}(\psi)$ find most-precise abstract values in their respective domains. In contrast to procedure $\widehat{\alpha}_{\mathrm{PA}}$, the call $\widehat{\alpha}_{\mathrm{CP}}((y = 3) \wedge (x = 4 * y + 1))$ finds an abstract value that is an exact representation of the final state of program (2), namely, $[\mathtt{x} \mapsto 13, \mathtt{y} \mapsto 3]$. This value is arrived at by the following sequence of operations:

Initialization:	$\mathtt{ans} := \bot$	
	$\varphi := (y = 3) \wedge (x = 4 * y + 1)$	
Iteration 1:	$S := [x \mapsto 13, y \mapsto 3]$	// A satisfying concrete store
	$\mathtt{ans} := \bot \sqcup \beta([x \mapsto 13, y \mapsto 3])$	
	$= [\mathtt{x} \mapsto 13, \mathtt{y} \mapsto 3]$	
	$\widehat{\gamma}(\mathtt{ans}) = (x = 13) \wedge (y = 3)$	
	$\varphi := (y = 3) \wedge (x = 4 * y + 1) \wedge \neg((x = 13) \wedge (y = 3))$	
	$= (y = 3) \wedge (x = 4 * y + 1) \wedge ((x \neq 13) \vee (y \neq 3))$	
	$=$ **ff**	
Iteration 2:	φ is unsatisfiable	
Return value:	$[\mathtt{x} \mapsto 13, \mathtt{y} \mapsto 3]$	

The loop terminates and $\widehat{\alpha}_{\mathrm{CP}}$ returns the abstract value $[\mathtt{x} \mapsto 13, \mathtt{y} \mapsto 3]$.

This example is a somewhat degenerate one because only one iteration of the loop body is performed. To illustrate $\widehat{\alpha}_{\mathrm{CP}}$ on a slightly more challenging example, consider the program

$$\begin{array}{l} \mathtt{z} := 0 \\ \mathtt{x} := \mathtt{y} * \mathtt{z} \end{array} \tag{5}$$

and suppose that ψ is the formula $(z = 0) \wedge (x = y * z)$, which captures the final state of program (5). The following sequence of operations would be performed during the invocation of $\widehat{\alpha}_{\mathrm{CP}}((z = 0) \wedge (x = y * z))$:

Initialization:	$\mathtt{ans} := \bot$	
	$\varphi := (z = 0) \wedge (x = y * z)$	
Iteration 1:	$S := [x \mapsto 0, y \mapsto 43, z \mapsto 0]$	// Some satisfying concrete store
	$\mathtt{ans} := \bot \sqcup \beta([x \mapsto 0, y \mapsto 43, z \mapsto 0])$	
	$= [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 43, \mathtt{z} \mapsto 0]$	
	$\widehat{\gamma}(\mathtt{ans}) = (x = 0) \wedge (y = 43) \wedge (z = 0)$	
	$\varphi := (z = 0) \wedge (x = y * z) \wedge \neg((x = 0) \wedge (y = 43) \wedge (z = 0))$	
	$= (z = 0) \wedge (x = y * z) \wedge ((x \neq 0) \vee (y \neq 43) \vee (z \neq 0))$	
	$= (z = 0) \wedge (x = y * z) \wedge (y \neq 43)$	
Iteration 2:	$S := [x \mapsto 0, y \mapsto 46, z \mapsto 0]$	// Some satisfying concrete store
	$\mathtt{ans} := [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 43, \mathtt{z} \mapsto 0] \sqcup \beta([x \mapsto 0, y \mapsto 46, z \mapsto 0])$	
	$= [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 43, \mathtt{z} \mapsto 0] \sqcup [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 46, \mathtt{z} \mapsto 0]$	
	$= [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0]$	
	$\widehat{\gamma}(\mathtt{ans}) = (x = 0) \wedge (z = 0)$	
	$\varphi := (z = 0) \wedge (x = y * z) \wedge (y \neq 43) \wedge ((x \neq 0) \vee (z \neq 0))$	
	$=$ **ff**	
Iteration 3:	φ is unsatisfiable	
Return value:	$[\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0]$	

In effect, $\widehat{\alpha}_{\mathrm{CP}}$ has automatically discovered that in the abstract world the best treatment of the multiplication operator is for it to be non-strict in $\top$. That is, 0 is a multiplicative annihilator that supersedes $\top$: $0 = \top * 0$.

Fig. 3 presents a sequence of diagrams that illustrate schematically algorithm $\widehat{\alpha}$ from Fig. 2. In general, $\widehat{\alpha}(\psi)$ carries out a process of successive approximation, making repeated calls to the theorem prover. Initially, φ is set to ψ and ans is set to $\bot$. On each iteration of the loop in $\widehat{\alpha}$, the value of ans becomes a better approximation of the desired answer, and the value of φ describes a smaller set of concrete stores, namely, those stores described by ψ that are not, as yet, covered by ans. For instance, at line [7] of Fig. 2 during Iteration 1 of the second example of $\widehat{\alpha}_{\mathrm{CP}}(\psi)$, ans has the value $[\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 43, \mathtt{z} \mapsto 0]$, and the update to φ, $\varphi := \varphi \wedge \neg\widehat{\gamma}(\mathtt{ans})$, sets φ to $(z = 0) \wedge (x = y * z) \wedge (y \neq 43)$. Thus, φ describes exactly the stores that are described by ψ, but are not, as yet, covered by ans.

Each time around the loop, $\widehat{\alpha}$ selects a concrete store S such that $S \models \varphi$. Then $\widehat{\alpha}$ uses β and $\sqcup$ to perform what can be viewed as a "generalization" operation: β converts concrete store S into an abstract store; the current value of ans is augmented with $\beta(S)$ using $\sqcup$. For instance, at line [6] of Fig. 2 during Iteration 2 of the second example of $\widehat{\alpha}_{\mathrm{CP}}(\psi)$, ans's value is changed from $[\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 43, \mathtt{z} \mapsto 0]$ to $[\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 43, \mathtt{z} \mapsto 0] \sqcup \beta([x \mapsto 0, y \mapsto 46, z \mapsto 0])$ $= [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0]$. In other words, the generalization from two possible values for y, 43 and 46, is $\top$, which indicates that y may not be a constant at the end of the program.

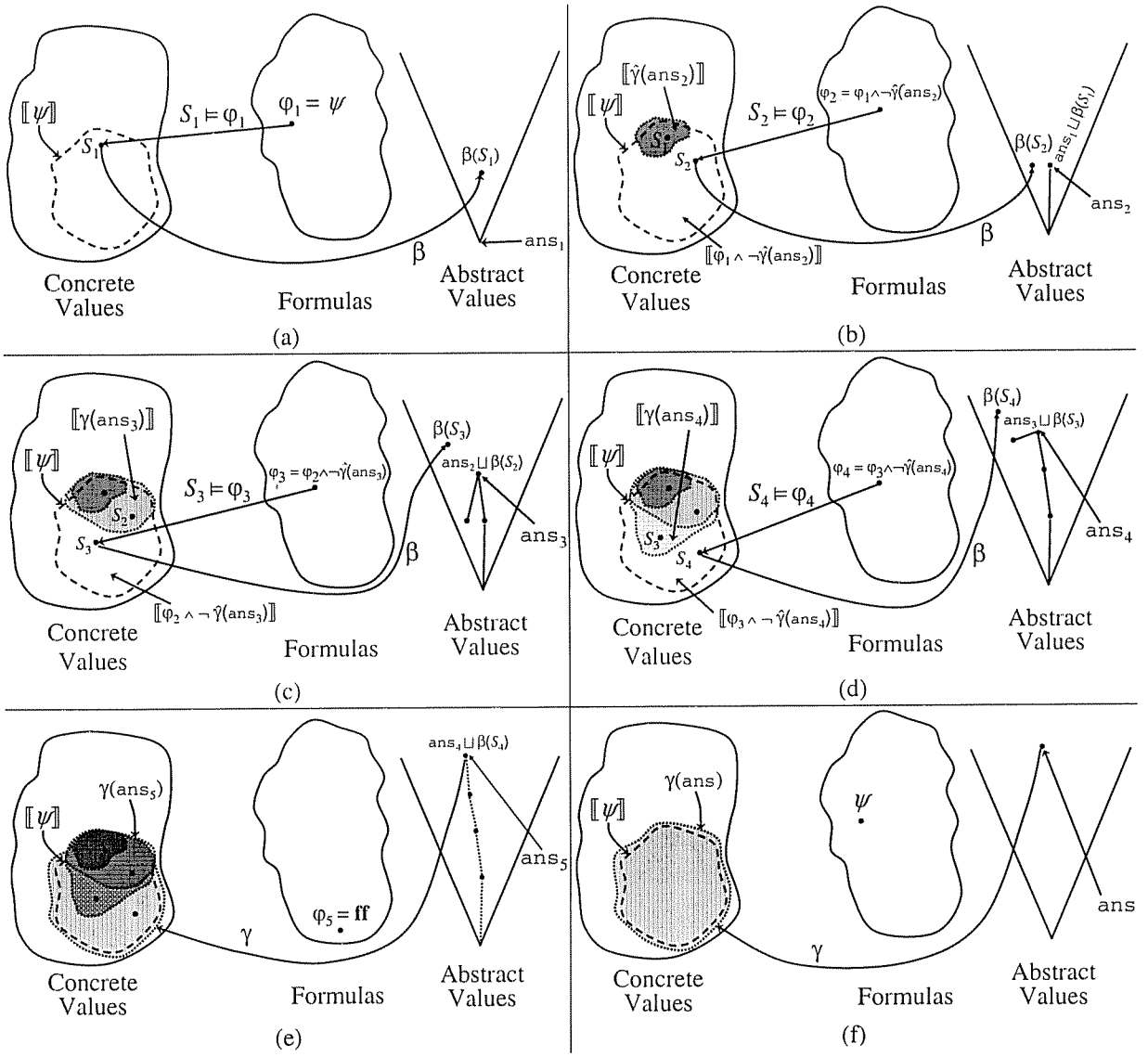

Fig. 3. Schematic diagrams that illustrate the process carried out by algorithm $\widehat{\alpha}(\psi)$ from Fig. 2; φ_i, S_i, and $\mathtt{ans}_i$ denote the values of φ, S, and ans during the i^{th} iteration. (a) Initially, φ_1 is set to ψ and $\mathtt{ans}_1$ is set to $\bot$; S_1 is a structure such that $S_1 \models \varphi_1$. (b) $\mathtt{ans}_2$ is set to $\mathtt{ans}_1 \sqcup \beta(S_1) = \beta(S_1)$; φ_2 is set to $\varphi_1 \wedge \neg\widehat{\gamma}(\mathtt{ans}_2)$; S_2 is a structure such that $S_2 \models \varphi_2$. Note that S_2 belongs to $[\![\varphi_2]\!] = [\![\varphi_1 \wedge \neg\widehat{\gamma}(\mathtt{ans}_2)]\!]$. (c) $\mathtt{ans}_3$ is set to $\mathtt{ans}_2 \sqcup \beta(S_2)$; φ_3 is set to $\varphi_2 \wedge \neg\widehat{\gamma}(\mathtt{ans}_3)$; S_3 is a structure such that $S_3 \models \varphi_3$. (d) $\mathtt{ans}_4$ is set to $\mathtt{ans}_3 \sqcup \beta(S_3)$; φ_4 is set to $\varphi_3 \wedge \neg\widehat{\gamma}(\mathtt{ans}_4)$; S_4 is a structure such that $S_4 \models \varphi_4$. (e) $\mathtt{ans}_5$ is set to $\mathtt{ans}_4 \sqcup \beta(S_4)$; φ_5 is set to $\varphi_4 \wedge \neg\widehat{\gamma}(\mathtt{ans}_5)$. In the case portrayed here, the loop terminates at this point because $\varphi_5 = \mathbf{ff}$. The desired answer is held in $\mathtt{ans}_5$. (f) $\widehat{\alpha}(\psi)$ obtains the most-precise abstract value ans that overapproximates $[\![\psi]\!]$.

Sect. 4 presents a general framework for implementing α functions of Galois connections, formalized in terms of logic, that generalizes the foregoing examples.

3 Terminology and Notation

3.1 Representing Concrete States using Logical Structures

For us, concrete stores are *logical structures*. The advantage of adopting this outlook is that it allows potentially infinite sets of concrete stores to be represented using formulas.

Definition 1. *Let $P = \{p_1, \ldots, p_m\}$ be a finite set of predicate symbols, each with a fixed arity; let P_i denote the set of predicate symbols with arity i. Let $C = \{c_1, \ldots, c_n\}$ be a finite set of constant symbols. Let $F = \{f_1, f_2, \ldots, f_p\}$ be*

a finite set of function symbols each with a fixed arity; let F_i denote the set of function symbols with arity i. We denote the vocabulary of the analyzed program by $V = \langle P, C, F \rangle$. A **concrete store over** *V is a tuple $S = \langle U, \iota_p, \iota_c, \iota_f \rangle$ in which*

- *U is a (possibly infinite) set of individuals.*
- *ι_p is the interpretation of predicate symbols, i.e., for every predicate symbol $p \in P_i$, $\iota_p(p) \subseteq U^i$ denotes the set of i-tuples for which p holds.*
- *ι_c is the interpretation of constant symbols, i.e., for every constant symbol $c \in C$, $\iota_c(c) \in U$ denotes the individual associated with c.*
- *ι_f is the interpretation of function symbols, i.e., for every function symbol $f \in F_i$, $\iota_f(f) : U^i \to U$ maps i-tuples into an individual.*

We denote the (infinite) set of structures by $ConcreteStruct[V]$.

Example 1. In Sect. 2, we considered concrete stores to be members of $Var \to \mathcal{Z}$. This is a common way to define concrete stores; however, in the remainder of the paper concrete stores are identified with logical structures. A store in which program variables are bound to integer values is a logical structure $\langle \mathcal{Z}, \iota_{IntPreds}, \iota_{Var}, \iota_{IntFuncs} \rangle$ over vocabulary $\langle IntPreds, Var, IntFuncs \rangle$, where

- $IntPreds = \{<, \leq, =, \neq, \geq, >, \ldots\}$; $\iota_{IntPreds}$ gives these symbols their usual meanings.
- ι_{Var} is a mapping of program variables to integers.
- $IntFuncs = \{+, -, *, /, \ldots\}$; $\iota_{IntFuncs}$ gives these symbols their usual meanings.

For instance, an example concrete store for a program in which $Var = \{x, y, z\}$ is

$$\langle \mathcal{Z}, \iota_{IntPreds}, [x \mapsto 0, y \mapsto 2, z \mapsto 0], \iota_{IntFuncs} \rangle. \tag{6}$$

Because $\mathcal{Z}$, $\iota_{IntPreds}$, and $\iota_{IntFuncs}$ are common to all our examples, henceforth we abbreviate a store such as (6) by $\iota_c = [x \mapsto 0, y \mapsto 2, z \mapsto 0]$.

3.2 Syntax and Semantics of First-Order Logic

To manipulate sets of structures symbolically, we use formulas of first-order logic with equality, whose syntax is defined as follows:

Definition 2. *The languages of* **terms** *and* **formula** *over* **vocabulary** *$V = \langle P, C, F \rangle$ are defined by*

$t \in Terms$	$c \in C$	$\varphi \in Formulas$	$t ::= v \mid c \mid f(t_1, \ldots, t_k)$
$v \in Variables$	$f \in F$	$p \in P$	$\varphi ::= \mathbf{ff} \mid \mathbf{tt} \mid p(t_1, \ldots, t_k) \mid (t_1 = t_2) \mid (\neg\varphi_1) \mid (\varphi_1 \vee \varphi_2) \mid (\exists v_1 : \varphi_1)$

The sets of free variables of terms and formulas are defined as usual. A formula is **closed** *when it has no free variables.*

We use several shorthand notations: $(t_1 \neq t_2) \stackrel{\text{def}}{=} \neg(t_1 = t_2)$; $\varphi_1 \Rightarrow \varphi_2 \stackrel{\text{def}}{=} (\neg\varphi_1 \vee \varphi_2)$; $\varphi_1 \wedge \varphi_2 \stackrel{\text{def}}{=} \neg(\neg\varphi_1 \vee \neg\varphi_2)$; $\varphi_1 \Leftrightarrow \varphi_2 \stackrel{\text{def}}{=} (\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1)$; and $\forall v : \varphi \stackrel{\text{def}}{=} \neg\exists v : \neg\varphi$. The order of precedence among the connectives, from highest to lowest, is as follows: $\neg$, $\wedge$, $\vee$, $\forall$, and $\exists$. We drop parentheses wherever possible, except for emphasis.

The standard Tarskian semantics for first-order logic is defined as follows:

Definition 3. *Let $S = \langle U, \iota_p, \iota_c, \iota_f \rangle$ be a concrete structure over V. An* **assignment** *Z is a function that maps free variables to individuals (i.e., an assignment has the functionality $Z : \{v_1, v_2, \ldots\} \to U$). An assignment that is defined on all free variables of a formula φ is called* **complete** *for φ. (We will assume that every assignment Z that arises in connection with the discussion of some formula φ is complete for φ.)*

We define the meaning of term t (denoted by $[\![t]\!]^{\langle S,Z \rangle}$) inductively, as follows: (i) For a constant symbol c, $[\![c]\!]^{\langle S,Z \rangle} = \iota_c(c)$. (ii) For a variable v, $[\![v]\!]^{\langle S,Z \rangle} = Z(v)$. (iii) If $t_1, t_2, \ldots, t_l$ are terms, and $f \in F$ is a function of arity l, then

$$[\![f(t_1, \ldots, t_l)]\!]^{\langle S,Z \rangle} = \iota_f(f)([\![t_1]\!]^{\langle S,Z \rangle}, \ldots, [\![t_l]\!]^{\langle S,Z \rangle}).$$

S and Z **satisfy** *φ (denoted by $S, Z \models \varphi$) when one of the following holds:*

- *$\varphi \equiv \mathbf{tt}$*
- *$\varphi \equiv p(t_1, \ldots, t_k)$ and $\langle [\![t_1]\!]^{\langle S,Z \rangle}, \ldots, [\![t_k]\!]^{\langle S,Z \rangle} \rangle \in \iota_p(p)$.*
- *$\varphi \equiv (t_1 = t_2)$ and $[\![t_1]\!]^{\langle S,Z \rangle} = [\![t_2]\!]^{\langle S,Z \rangle}$.*
- *$\varphi \equiv \neg\varphi_0$ and $S, Z \models \varphi_0$ does not hold.*
- *$\varphi \equiv \varphi_1 \vee \varphi_2$, and either $S, Z \models \varphi_1$ or $S, Z \models \varphi_2$.*
- *$\varphi \equiv \exists v_1 : \varphi_1$ and there exists an individual $u \in U$, such that $S, Z[v_1 \mapsto u] \models \varphi_1$.*

For a closed formula φ, we omit the assignment in the satisfaction relation, and merely write $S \models \varphi$. We also use the notation $[\![\varphi]\!]$ to denote the set of concrete structures that satisfy φ: $[\![\varphi]\!] = \{S \mid S \in ConcreteStruct[V], S \models \varphi\}$.

Example 2.

$$[\![(x = 0) \wedge (z = 0)]\!] = \left\{ \begin{array}{l} \iota_c = [x \mapsto 0, y \mapsto 0, z \mapsto 0], \iota_c = [x \mapsto 0, y \mapsto 1, z \mapsto 0], \\ \iota_c = [x \mapsto 0, y \mapsto 2, z \mapsto 0], \ldots \end{array} \right\}$$

3.3 Lattices

Definition 4. *A* **complete join semilattice** *$L = \langle L, \sqsubseteq, \bigsqcup, \bot \rangle$ is a partially ordered set with partial order $\sqsubseteq$. For every subset X of L, the least upper bound of X, denoted by $\bigsqcup X$, must be a member of L.*

The minimal element $\bot \in L$ is $\bigsqcup \emptyset$. We use $x \sqcup y$ as a shorthand for $\bigsqcup\{x, y\}$. We write $x \sqsubset y$ when $x \sqsubseteq y$ and $x \neq y$.

The powerset of concrete stores $2^{ConcreteStruct[V]}$ is a complete join semilattice, where (i) $X \sqsubseteq Y$ iff $X \subseteq Y$, (ii) $\bot = \emptyset$, and (iii) $\bigsqcup = \bigcup$.

Definition 5. *Let $L = \langle L, \sqsubseteq, \bigsqcup, \bot \rangle$ be a complete join semilattice. A* **strictly increasing chain in** *L is a sequence of values $l_1, l_2, \ldots$, such that $l_i \sqsubset l_{i+1}$. We say that L* **has finite height** *if every strictly increasing chain is finite.*

3.4 Abstract Domains

We now define an abstract domain by means of a representation function [21].

Definition 6. *Given a complete join semilattice $L = \langle L, \sqsubseteq, \bigsqcup, \bot \rangle$ and a* **representation function** *β: $ConcreteStruct[V] \to L$ such that for all $S \in ConcreteStruct[V]$ $\beta(S) \neq \bot$, a Galois connection $2^{ConcreteStruct[V]} \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} L$ is defined by extending β pointwise, i.e., for $XS \subseteq ConcreteStruct[V]$ and $l \in L$,*

$$\alpha(XS) = \bigsqcup_{S \in XS} \beta(S) \qquad \gamma(l) = \{S \mid S \in ConcreteStruct[V], \beta(S) \sqsubseteq l\}$$

It is straightforward to show that this defines a Galois connection, i.e., (i) α and γ are monotonic, (ii) α distributes over $\cup$, (iii) $XS \subseteq \gamma(\alpha(XS))$, and (iv) $\alpha(\gamma(l)) \sqsubseteq l$.

We say that l **overapproximates** *a set of concrete stores XS if $\gamma(l) \supseteq XS$. It is straightforward to show that $\alpha(XS)$ is the most-precise (i.e., least) abstract value that overapproximates XS.*

Example 3. In our examples, the abstract domain will continue to be the one introduced in Sect. 2, namely, $(Var \to \mathcal{Z}^\top)_\bot$. As we saw in Sect. 2, β maps a concrete store like $\iota_c = [x \mapsto 0, y \mapsto 2, z \mapsto 0]$ to an abstract value $[\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 2, \mathtt{z} \mapsto 0]$. Thus,

$$\begin{aligned} \alpha\left(\left\{ \begin{array}{l} \iota_c = [x \mapsto 0, y \mapsto 0, z \mapsto 0], \\ \iota_c = [x \mapsto 0, y \mapsto 2, z \mapsto 0] \end{array} \right\}\right) &= \beta(\iota_c = [x \mapsto 0, y \mapsto 0, z \mapsto 0]) \sqcup \beta(\iota_c = [x \mapsto 0, y \mapsto 2, z \mapsto 0]) \\ &= [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 0, \mathtt{z} \mapsto 0] \sqcup [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 2, \mathtt{z} \mapsto 0] \\ &= [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0]. \end{aligned}$$

Suppose that abstract value l is $[\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0]$. Because $\mathtt{y} \mapsto \top$ does not place any restrictions on the value of y, we have

$$\gamma(l) = \left\{ \begin{array}{l} \iota_c = [x \mapsto 0, y \mapsto 0, z \mapsto 0], \iota_c = [x \mapsto 0, y \mapsto 1, z \mapsto 0], \\ \iota_c = [x \mapsto 0, y \mapsto 2, z \mapsto 0], \ldots \end{array} \right\}$$

4 Symbolic Implementation of the α Function

This section presents a general framework for implementing α functions of Galois connections using procedure $\widehat{\alpha}$ from Fig. 2. $\widehat{\alpha}(\psi)$ finds the most-precise abstract value in a finite-height lattice, given a specification of a set of concrete stores as a logical formula ψ. $\widehat{\alpha}$ represents sets of concrete stores symbolically, using formulas, and invokes a theorem prover on each iteration.

The assumptions of the framework are rather minimal:

- The concrete domain is the power set of $ConcreteStruct[V]$.

- The concrete and abstract domains are related by a Galois connection defined by a representation function β that maps a structure $S \in \mathit{ConcreteStruct}[V]$ to an abstract value $\beta(S)$.
- It is possible to take the join of two abstract values.
- There is an operation $\widehat{\gamma}$ that maps an abstract value l to a formula $\widehat{\gamma}(l)$ such that

$$[\![\widehat{\gamma}(l)]\!] = \gamma(l). \tag{7}$$

Operation $\widehat{\gamma}$ permits the concretization of an abstract value to be represented symbolically, using a logical formula, which allows sets of concrete stores to be manipulated symbolically, via operations on formulas. (In this paper, we use first-order logic; in general, however, other logics could be used.)

Example 4. As we saw in Sect. 2, because $\mathtt{y} \mapsto \top$ does not place any restrictions on the value of y, we have $\widehat{\gamma}([\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0]) = (x = 0) \wedge (z = 0)$. From Exs. 2 and 3, we know that

$$\begin{aligned}[\![(x = 0) \wedge (z = 0)]\!] &= \left\{ \begin{array}{l} \iota_c = [x \mapsto 0, y \mapsto 0, z \mapsto 0], \iota_c = [x \mapsto 0, y \mapsto 1, z \mapsto 0], \\ \iota_c = [x \mapsto 0, y \mapsto 2, z \mapsto 0], \ldots \end{array} \right\} \\ &= \gamma([\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0]),\end{aligned}$$

and thus Eqn. (7) is satisfied. For $l \in (\mathit{Var} \to \mathcal{Z}^{\top})_{\perp}$, $\widehat{\gamma}(l)$ is defined as follows:

$$\widehat{\gamma}(l) = \begin{cases} \mathbf{ff} & \text{if } l = \perp \\ \bigwedge\limits_{\substack{v \in \mathit{Var}, \\ l(v) \neq \top}} (v = l(v)) & \text{otherwise} \end{cases}$$

Specification of Alpha Procedure $\widehat{\alpha}$ is to implement α, given a specification of a set of concrete stores as a logical formula ψ. Therefore, $\widehat{\alpha}$ must have the property that for all ψ, $\widehat{\alpha}(\psi) = \alpha([\![\psi]\!])$.

Note that a logical formula ψ represents the set of concrete stores $[\![\psi]\!]$; thus, $\alpha([\![\psi]\!])$ (and hence $\widehat{\alpha}(\psi)$, as well) is the most-precise abstract value that overapproximates the set of concrete stores represented symbolically by ψ.

Implementation of Alpha Procedure $\widehat{\alpha}$ is given in Fig. 2.

Example 5. Two examples of calls on $\widehat{\alpha}$ for the constant-propagation domain $(\mathit{Var} \to \mathcal{Z}^{\top})_{\perp}$ were given in Sect. 2. In generalizing the idea from Sect. 2, concrete stores have been identified with logical structures, so instead of writing, e.g., $S := [x \mapsto 0, y \mapsto 43, z \mapsto 0]$, we would now write $S := \iota_c = [x \mapsto 0, y \mapsto 43, z \mapsto 0]$.

Theorem 1. *Suppose that the abstract domain has finite height of at most h. Given input ψ, $\widehat{\alpha}(\psi)$ has the following properties:*

(i) *The loop on lines [4]–[8] in procedure $\widehat{\alpha}$ is executed at most h times.*
(ii) *$\widehat{\alpha}(\psi) = \alpha([\![\psi]\!])$ (i.e., $\widehat{\alpha}(\psi)$ computes the most-precise abstract value that overapproximates the set of concrete stores represented symbolically by ψ).*

Proof. *See App. A.*

5 Symbolic Implementation of Transfer Functions

5.1 Transfer Functions for Statements

If Q is a set of predicate, constant, or function symbols, let Q' denote the same set of symbols, but with a $'$ attached to each symbol (i.e., $q \in Q$ iff $q' \in Q'$).

The interpretation of statements involves the specification of transition relations using formulas. Such formulas will be over a "two-store vocabulary" $V \cup V' = \langle P \cup P', C \cup C', F \cup F' \rangle$, where unprimed symbols will be referred to as *present-state* symbols, and primed symbols as *next-state* symbols. If S and S' are structures over vocabularies $V = \langle P, C, F \rangle$ and $V' = \langle P', C', F' \rangle$, respectively, the satisfaction relation for a two-store formula τ will be written as $\langle S, S' \rangle \models \tau$.

Example 6. The formula that expresses the semantics of an assignment $\mathtt{x} := \mathtt{y} * \mathtt{z}$ with respect to stores over vocabulary $\langle P, C, F \rangle$, denoted by $\tau_{\mathtt{x}:=\mathtt{y}*\mathtt{z}}$, can be specified as

$$\tau_{\mathtt{x}:=\mathtt{y}*\mathtt{z}} \stackrel{\text{def}}{=} (x' = y * z) \wedge (y' = y) \wedge (z' = z).$$

For parallel form, we will also assume that we have two isomorphic abstract domains, L and L', and associated variants of β and $\widehat{\gamma}$

$$\beta\colon \mathit{ConcreteStruct}[V] \to L \qquad \beta'\colon \mathit{ConcreteStruct}[V'] \to L'$$
$$\widehat{\gamma}\colon L \to \mathit{Formula}[V] \qquad \widehat{\gamma}'\colon L' \to \mathit{Formula}[V']$$

For the constant-propagation domain, this just means that a next-state abstract value produced by one transition, e.g., $[\mathtt{x}' \mapsto 0, \mathtt{y}' \mapsto \top, \mathtt{z}' \mapsto 0] \in L'$, can be identified as the present-state abstract value $[\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0] \in L$ for the next transition.

Specification Given a formula τ for a statement's transition relation, the result of applying τ to a set of concrete stores XS is

$$\mathrm{Post}[\tau](XS) = \{S' \mid \text{exists } S \in XS \text{ such that } \langle S, S' \rangle \models \tau\}.$$

(Note that this is a set of structures over vocabulary V'.) $\widehat{\alpha\mathrm{Post}}[\tau](l)$ is to return the most-precise abstract value in L' that overapproximates $\mathrm{Post}[\tau](\gamma(l))$.

Implementation $\widehat{\alpha\mathrm{Post}}[\tau](l)$ can be computed by the procedure presented in Fig. 4. After φ is initialized to $\widehat{\gamma}(l) \wedge \tau$ in line [3], $\widehat{\alpha\mathrm{Post}}$ operates very much like $\widehat{\alpha}$, except that only abstractions of the S' structures are accumulated in variable `ans'` (see lines [5] and [6]). On each iteration of the loop in $\widehat{\alpha\mathrm{Post}}$, the value of `ans'` becomes a better approximation of the desired answer, and the value of φ describes a smaller set of concrete stores, namely, those $V \cup V'$ stores that are described by $\widehat{\gamma}(l) \wedge \tau$, but whose range (i.e., projection on the next-state symbols) is not, as yet, covered by `ans'`.

```
[1]  L' αPost(two-store-formula τ over vocabulary V ∪ V', L l) {
[2]    ans' := ⊥'
[3]    φ := γ̂(l) ∧ τ
[4]    while (φ is satisfiable) {
[5]      Select a structure pair ⟨S,S'⟩ such that ⟨S,S'⟩ ⊨ φ
[6]      ans' := ans' ⊔ β'(S')
[7]      φ := φ ∧ ¬γ̂'(ans')
[8]    }
[9]    return ans'
[10] }
```

Fig. 4. An algorithm that implements $\widehat{\alpha\mathrm{Post}}[\tau](l)$.

Example 7. Suppose that $l = [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0]$, and the statement to be interpreted is $\mathtt{x} := \mathtt{y} * \mathtt{z}$. Then $\widehat{\gamma}(l)$ is the formula $(z = 0)$, and $\tau_{\mathtt{x}:=\mathtt{y}*\mathtt{z}}$ is the formula $(x' = y * z) \wedge (y' = y) \wedge (z' = z)$. Fig. 5 shows why we have

$$\widehat{\alpha\mathrm{Post}}[\tau_{\mathtt{x}:=\mathtt{y}*\mathtt{z}}]([\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0]) = [\mathtt{x}' \mapsto 0, \mathtt{y}' \mapsto \top, \mathtt{z}' \mapsto 0].$$

Space precludes us from discussing other operators that can also be handled by techniques similar to Fig. 4, such as $\mathrm{Pre}[\tau]$.

5.2 Transfer Functions for Conditions

Specification The interpretation of a condition φ with respect to a given abstract value l must "pass through" all structures that are both represented by l and satisfy φ, i.e., those in $\gamma(l) \cap [\![\varphi]\!]$. Thus, the most-precise approximation to the interpretation of condition φ, denoted by $\mathrm{Assume}^\sharp[\varphi](l)$, is defined by

$$\mathrm{Assume}^\sharp[\varphi](l) = \alpha(\gamma(l) \cap [\![\varphi]\!]).$$

Implementation $\mathrm{Assume}^\sharp[\varphi](l)$ can be computed by the following method:

$$\mathrm{Assume}^\sharp[\varphi](l) = \widehat{\alpha}(\widehat{\gamma}(l) \wedge \varphi).$$

Initialization: $\texttt{ans}' := \bot'$

$\varphi := (z = 0) \wedge (x' = y * z) \wedge (y' = y) \wedge (z' = z)$

Iteration 1: $\langle S, S' \rangle := \iota_c = \begin{bmatrix} x \mapsto 5, y \mapsto 17, z \mapsto 0 \\ x' \mapsto 0, y' \mapsto 17, z' \mapsto 0 \end{bmatrix}$ // Some satisfying structure

$\texttt{ans}' := [\texttt{x}' \mapsto 0, \texttt{y}' \mapsto 17, \texttt{z}' \mapsto 0]$

$\widehat{\gamma}'(\texttt{ans}') = (x' = 0) \wedge (y' = 17) \wedge (z' = 0)$

$\varphi := (z = 0) \wedge (x' = y * z) \wedge (y' = y) \wedge (z' = z) \wedge ((x' \neq 0) \vee (y' \neq 17) \vee (z' \neq 0))$

$= (z = 0) \wedge (x' = y * z) \wedge (y' = y) \wedge (z' = z) \wedge (y' \neq 17)$

Iteration 2: $\langle S, S' \rangle := \iota_c = \begin{bmatrix} x \mapsto 12, y \mapsto 99, z \mapsto 0 \\ x' \mapsto 0, y' \mapsto 99, z' \mapsto 0 \end{bmatrix}$ // Some satisfying structure

$\texttt{ans}' := [\texttt{x}' \mapsto 0, \texttt{y}' \mapsto 17, \texttt{z}' \mapsto 0] \sqcup [\texttt{x}' \mapsto 0, \texttt{y}' \mapsto 99, \texttt{z}' \mapsto 0]$

$= [\texttt{x}' \mapsto 0, \texttt{y}' \mapsto \top, \texttt{z}' \mapsto 0]$

$\widehat{\gamma}'(\texttt{ans}') = (x' = 0) \wedge (z' = 0)$

$\varphi := (z = 0) \wedge (x' = y * z) \wedge (y' = y) \wedge (z' = z) \wedge (y' \neq 17) \wedge ((x' \neq 0) \vee (z' \neq 0))$

$= \mathbf{ff}$

Iteration 3: φ is unsatisfiable

Return value: $[\texttt{x}' \mapsto 0, \texttt{y}' \mapsto \top, \texttt{z}' \mapsto 0]$

Fig. 5. Operations performed during a call $\widehat{\alpha\text{Post}}[\tau_{\texttt{x:=y*z}}]([\texttt{x} \mapsto \top, \texttt{y} \mapsto \top, \texttt{z} \mapsto 0])$.

Example 8.

$$\begin{aligned}
\text{Assume}^\sharp[(y < z)]([\texttt{x} \mapsto 0, \texttt{y} \mapsto 2, \texttt{z} \mapsto 7]) &= \widehat{\alpha}((x = 0) \wedge (y = 2) \wedge (z = 7) \wedge (y < z)) \\
&= [\texttt{x} \mapsto 0, \texttt{y} \mapsto 2, \texttt{z} \mapsto 7] \\
\text{Assume}^\sharp[(y \geq z)]([\texttt{x} \mapsto 0, \texttt{y} \mapsto 2, \texttt{z} \mapsto 7]) &= \widehat{\alpha}((x = 0) \wedge (y = 2) \wedge (z = 7) \wedge (y \geq z)) \\
&= \bot \\
\text{Assume}^\sharp[(y < z)]([\texttt{x} \mapsto 0, \texttt{y} \mapsto \top, \texttt{z} \mapsto 7]) &= \widehat{\alpha}((x = 0) \wedge (z = 7) \wedge (y < z)) \\
&= [\texttt{x} \mapsto 0, \texttt{y} \mapsto \top, \texttt{z} \mapsto 7] \\
\text{Assume}^\sharp[(y = z)]([\texttt{x} \mapsto 0, \texttt{y} \mapsto \top, \texttt{z} \mapsto 7]) &= \widehat{\alpha}((x = 0) \wedge (z = 7) \wedge (y = z)) \\
&= [\texttt{x} \mapsto 0, \texttt{y} \mapsto 7, \texttt{z} \mapsto 7]
\end{aligned}$$

6 Querying Abstract Values

For clients of abstract interpretation, such as verification tools, program optimizers, program-understanding tools, etc., an abstract value in the fixed-point solution may not provide information in exactly the form that the client needs. For instance, the information that the abstract value at `p` in the fixed-point solution is $[\texttt{x} \mapsto 0, \texttt{y} \mapsto \top]$ may not *itself* be the information that a client needs. Therefore, a fundamental issue for clients of abstract interpretation is how to extract information from an abstract value.

Information extraction means query evaluation. For instance, if a program optimizer poses the query "Does program condition `x == 0` evaluate to `true` in all stores that arise at program point `p`?" and the answer is **tt** (for instance, because the abstract value at `p` in the fixed-point solution is $[\texttt{x} \mapsto 0, \texttt{y} \mapsto \top]$), then it has sufficient information to make the simplification

`p: if (x == 0) then` S_1 `else` S_2 `fi` $\Rightarrow$ `p:` S_1.

The observation that information extraction is query evaluation allows us to establish what it means to give the most-precise conservative answer to a query. An abstract value l represents a set of concrete stores XS; ideally, a query φ should return an answer that summarizes the result of posing φ against each concrete store $S \in XS$:

- If φ is true for each S, the summary answer should be **tt**.
- If φ is false for each S, the summary answer should be **ff**.
- If φ is true for some $S \in X$ but false for some $S' \in X$, the summary answer can only be "unknown".

This section addresses the problem of extracting information from an abstract value, and gives a procedure for determining the most-precise conservative answer to a query.

Specification This paper harnesses the tools of 2-valued logic for program analysis. However, the operation of extracting information from an abstract value is inherently 3-valued; to specify the extraction of information from an abstract value, it is convenient to introduce a third truth value, **?**, to denote uncertainty. We say that the values **ff** and **tt** are definite values and that **?** is an indefinite value.

Using this notation, the most-precise conservative value that can be reported for the value of formula φ in the concrete structures represented by l, denoted by $\langle\!\langle \varphi \rangle\!\rangle(l)$, is

$$\langle\!\langle \varphi \rangle\!\rangle(l) = \begin{cases} \mathbf{tt} & \text{if } S \models \varphi \text{ for all } S \in \gamma(l) \\ \mathbf{ff} & \text{if } S \not\models \varphi \text{ for all } S \in \gamma(l) \\ \mathbf{?} & \text{otherwise} \end{cases} \tag{8}$$

Eqn. (8) can also be written as

$$\langle\!\langle \varphi \rangle\!\rangle(l) = \begin{cases} \mathbf{tt} & \text{if } \gamma(l) \subseteq [\![\varphi]\!] \\ \mathbf{ff} & \text{if } \gamma(l) \subseteq [\![\neg\varphi]\!] \\ \mathbf{?} & \text{otherwise} \end{cases} \tag{9}$$

Example 9. Suppose that l is $[\mathrm{x} \mapsto 0, \mathrm{y} \mapsto \top, \mathrm{z} \mapsto 0]$ so that, as in Ex. 3,

$$\gamma(l) = \left\{ \begin{array}{l} \iota_c = [x \mapsto 0, y \mapsto 0, z \mapsto 0], \iota_c = [x \mapsto 0, y \mapsto 1, z \mapsto 0], \\ \iota_c = [x \mapsto 0, y \mapsto 2, z \mapsto 0], \ldots \end{array} \right\}$$

Now suppose that $\varphi \equiv (y = 1)$.

$$[\![(y = 1)]\!] = \left\{ \begin{array}{l} \iota_c = [x \mapsto 0, y \mapsto 1, z \mapsto 0], \iota_c = [x \mapsto 1, y \mapsto 1, z \mapsto 0], \\ \iota_c = [x \mapsto 2, y \mapsto 1, z \mapsto 0], \ldots \\ \iota_c = [x \mapsto 0, y \mapsto 1, z \mapsto 1], \iota_c = [x \mapsto 1, y \mapsto 1, z \mapsto 1], \\ \iota_c = [x \mapsto 2, y \mapsto 1, z \mapsto 1], \ldots \end{array} \right\}$$

Therefore, neither of the following hold:

$$\gamma(l) \subseteq [\![(y = 1)]\!] \qquad\qquad \gamma(l) \subseteq [\![(y \neq 1)]\!]$$

and thus $\langle\!\langle (y = 1) \rangle\!\rangle([\mathrm{x} \mapsto 0, \mathrm{y} \mapsto \top, \mathrm{z} \mapsto 0]) = \mathbf{?}$. Now suppose that $\varphi \equiv (x = y * z)$.

$$[\![(x = y * z)]\!] = \left\{ \begin{array}{l} \iota_c = [x \mapsto 0, y \mapsto 0, z \mapsto 0], \iota_c = [x \mapsto 0, y \mapsto 1, z \mapsto 0], \\ \iota_c = [x \mapsto 0, y \mapsto 2, z \mapsto 0], \ldots \\ \iota_c = [x \mapsto 0, y \mapsto 0, z \mapsto 1], \iota_c = [x \mapsto 1, y \mapsto 1, z \mapsto 1], \\ \iota_c = [x \mapsto 2, y \mapsto 2, z \mapsto 1], \ldots \\ \iota_c = [x \mapsto 0, y \mapsto 0, z \mapsto 2], \iota_c = [x \mapsto 2, y \mapsto 1, z \mapsto 2], \\ \iota_c = [x \mapsto 4, y \mapsto 2, z \mapsto 2], \ldots \end{array} \right\}$$

Thus, $\gamma(l) \subseteq [\![(x = y * z)]\!]$, and hence $\langle\!\langle (x = y * z) \rangle\!\rangle([\mathrm{x} \mapsto 0, \mathrm{y} \mapsto \top, \mathrm{z} \mapsto 0]) = \mathbf{tt}$.

Implementation Using symbolic techniques, $\langle\!\langle \varphi \rangle\!\rangle(l)$ can be computed as follows:

$$\langle\!\langle \varphi \rangle\!\rangle(l) = \begin{cases} \mathbf{tt} & \text{if } \widehat{\gamma}(l) \Rightarrow \varphi \text{ is valid} \\ \mathbf{ff} & \text{if } \widehat{\gamma}(l) \Rightarrow \neg\varphi \text{ is valid} \\ \mathbf{?} & \text{otherwise} \end{cases} \tag{10}$$

Example 10. Suppose that l is $[\mathrm{x} \mapsto 0, \mathrm{y} \mapsto \top, \mathrm{z} \mapsto 0]$ so that, as in Ex. 4, $\widehat{\gamma}(l) = (x = 0) \wedge (z = 0)$.

- When $\varphi \equiv (y = 1)$, neither of the following two formulas is valid:

$$(x = 0) \wedge (z = 0) \Rightarrow (y = 1) \qquad\qquad (x = 0) \wedge (z = 0) \Rightarrow (y \neq 1)$$

  Therefore, the value for $\langle\!\langle (y = 1) \rangle\!\rangle([\mathrm{x} \mapsto 0, \mathrm{y} \mapsto \top, \mathrm{z} \mapsto 0])$ computed according to Eqn. (10) is **?**.
- When $\varphi \equiv (x = y * z)$, the formula $(x = 0) \wedge (z = 0) \Rightarrow (x = y * z)$ is valid; therefore, the value for $\langle\!\langle (x = y * z) \rangle\!\rangle([\mathrm{x} \mapsto 0, \mathrm{y} \mapsto \top, \mathrm{z} \mapsto 0])$ computed according to Eqn. (10) is **tt**.

Each of these answers agrees with the corresponding answer obtained in Ex. 9.

Relationship to Supervaluational Semantics The role of **?** in Eqns. (8), (9), and (10) is different from the situation that arises with most abstract-interpretation algorithms when they return an indefinite answer [21,23]. For instance, in our past work on program-analysis methods based on 3-valued logic [23], a query on an abstract value l that results in **?** provides no information about the values that the query would have in the concrete stores that l represents: the query could evaluate to **tt** in all such stores, **ff** in all such stores, or **tt** in some stores and **ff** in others.

In contrast, in the approach followed in this paper, **?** does provide a certain amount of definite information: when the result of a query is **?**, we know that

- l definitely represents at least one concrete store on which the query evaluates to **tt**.
- l definitely represents at least one concrete store on which the query evaluates to **ff**.

This notion is related to the notion of *supervaluational semantics* [25, 4, 3, 22], which has also been called *thorough semantics* [5]. This concept arises in the context of partial logic and 3-valued logic, which are formalized in terms of logical structures in which the value of a predicate can be one of three values, **tt**, **ff**, or **?**, and truth values are ordered by the *information order*: $l_1 \sqsubseteq l_2$ iff $l_1 = l_2$ or $l_2 = \mathbf{?}$.

For instance, the supervaluational semantics of propositional logic is defined as follows: a 2-valued assignment a is a (finite) function from propositional variables to $\{\mathbf{ff}, \mathbf{tt}\}$; $[\![\varphi]\!](a)$ denotes the value of formula φ with respect to assignment a. A 3-valued assignment A is a (finite) function from propositional variables to $\{\mathbf{ff}, \mathbf{tt}, \mathbf{?}\}$; A represents all 2-valued assignments $a \sqsubseteq A$ (where $\sqsubseteq$ is lifted pointwise from truth values to assignments).

Definition 7. *Given a formula φ and assignment A, the* **3-valued supervaluational meaning** *of φ with respect to A, denoted by $\langle\!\langle \varphi \rangle\!\rangle(A)$, is the truth value in $\{\mathbf{ff}, \mathbf{tt}, \mathbf{?}\}$ defined by*

$$\langle\!\langle \varphi \rangle\!\rangle(A) = \begin{cases} \mathbf{tt} & \text{if } a \models \varphi \text{ for all } a \sqsubseteq A \\ \mathbf{ff} & \text{if } a \not\models \varphi \text{ for all } a \sqsubseteq A \\ \mathbf{?} & \text{otherwise} \end{cases} \tag{11}$$

Eqn. (11) has the same form as Eqn. (8) if we think of 3-valued assignments as abstractions of 2-valued assignments, with $\gamma(A) \stackrel{\text{def}}{=} \{a \mid a \sqsubseteq A\}$.

Bruns and Godefroid considered supervaluational semantics in the context of model checking partial Kripke structures [5]. The definition that they give for the supervaluational semantics also has a close resemblance to Eqn. (8):

> Let ϕ be a formula of any two-valued logic for which a satisfaction relation $\models$ is defined on complete Kripke structures. The truth value of ϕ in a state s of a partial Kripke structure M under the *thorough* interpretation, written $[(M, s) \models \phi]_t$, is defined as follows:

$$[(M, s) \models \phi]_t \stackrel{\text{def}}{=} \begin{cases} \mathbf{tt} & \text{if } (M', s') \models \phi \text{ for all } (M', s') \text{ in } \mathcal{C}(M, s) \\ \mathbf{ff} & \text{if } (M', s') \not\models \phi \text{ for all } (M', s') \text{ in } \mathcal{C}(M, s) \\ \mathbf{?} & \text{otherwise} \end{cases} \tag{12}$$

$\mathcal{C}(M, s)$ denotes the "completions" of state s of a partial Kripke structure M; $\mathcal{C}(M, s)$ is the set of all states s' of a complete Kripke structure M' such that $s \preceq s'$, where $s \preceq s'$ with respect to M and M' is a kind of "near bisimilarity" relation, except that the atomic propositions that hold in s may be more indefinite than those that hold in s'.[7]

Bruns and Godefroid give automaton-based decision procedures that can be used to implement the two tests needed in Eqn. (12), for the temporal logics CTL and LTL.

A difference between Eqns. (10) and (12) is that Eqn. (10) uses $\widehat{\gamma}$ to translate an abstract value to a formula in some logic. Eqn. (10) represents a reductionist strategy for providing a supervaluational evaluation procedure for an abstract domain by using existing logics and theorem provers/decision procedures. It shows how to obtain a supervaluational evaluation procedure whenever an appropriate logic, $\widehat{\gamma}$ function, and theorem prover/decision procedure are available.

7 Discussion

```
int x, y, z
Bool B1, B2
y := 3
x := 4 * y + 1
read(z)
B1 := z < 29
B2 := z < 27
if B1 then y := 5
if B2 then x := y + 8
```

Fig. 6. A program with correlated branches.

Theorem provers such as MACE [19], SEM [27], and Finder [24] can be used to implement the procedures presented in this paper because they return counterexamples. Such tools also exist for logics other than first-order logic; for example, MONA [17] can generate counterexamples for formulas in weak monadic second-order logic. Constraint-solving tools, such as ILOG Studio [15], can be used as well: the constraint problem to solve consists of the negation of the formula that is to be tested for validity; a feasible solution provides the counterexample. Simplify [11] provides counterexamples in symbolic form. (The details of how to adapt our algorithms when counterexamples are provided in symbolic form are beyond the scope of this paper.)

With the aid of Simplify, we have verified the constant-propagation examples in this paper, as well as examples that combine the constant-propagation domain with a predicate-abstraction domain. This is an additional benefit of our approach: it can be

[7] Eqn. (12) has been adapted to use the truth values **tt**, **ff**, and **?** employed in this paper.

used to generate the best transformer for combined domains, such as reduced cardinal product and those created using other domain constructors [9]. For example, the best transformer for the combined constant-propagation/predicate-abstraction domain determines that the variable x must be 13 at the end of the program given in Fig. 6.

It is usually the case that best transformers are non-compositional, i.e., the composition of the best transformers of two transitions may not be as precise as the best transformer of the composition of the transitions. One of the potential applications of $\widehat{\alpha\mathrm{Post}}$ is to develop more precise transformers for basic blocks: first create a formula that represents the compound transition for a basic block, and then apply $\widehat{\alpha\mathrm{Post}}$.

8 Related Work

This paper is most closely related to past work on predicate abstraction, which also uses theorem provers to implement most-precise versions of the basic abstract-interpretation operations. Predicate abstraction only applies to a family of finite-height abstract domains that are finite Cartesian products of Boolean values; our results generalize these ideas to a broader setting. In particular, our work shows that when a small number of conditions are met, most of the benefits that predicate-abstraction domains enjoy can also be enjoyed in arbitrary abstract domains of finite height, and possibly infinite cardinality. However, as illustrated in Figs. 1 and 3, our techniques are fundamentally different from the ones used in predicate abstraction. Although both $\widehat{\alpha}_{\mathrm{PA}}$ and the general procedure $\widehat{\alpha}$ of Fig. 2 use multiple calls on a theorem prover to pass from the space of formulas to the domain of abstract values, $\widehat{\alpha}_{\mathrm{PA}}$ goes *directly* from a formula to an abstract value, whereas $\widehat{\alpha}$ of Fig. 2 makes use of the domain of *concrete* values in a critical way: each time around the loop, $\widehat{\alpha}$ selects a concrete value S such that $S \models \varphi$; $\widehat{\alpha}$ uses β and $\sqcup$ to generalize from concrete value S to an abstract value.

We have sometimes been asked "How do your techniques compare with predicate abstraction augmented with an iterative-refinement scheme that generates new predicates, as in SLAM [2] or BLAST [14]?". We do not have a complete answer to this question; however, a few observations can be made:

- For the simple examples used for illustrative purposes in this paper, iterative refinement would obtain suitable predicates with appropriate constant values in one iteration. Our techniques achieve the desired precision using roughly the same theorem-proving support, but do not rely on heuristics-based machinery for changing the abstract domain in use.
- Our results extend ideas employed in the setting of predicate abstraction to a more general setting. In a companion paper [26], we have shown how to apply the approach presented here to a non-trivial abstract domain—essentially the one used in our past work on shape analysis [23] and in the TVLA system [18]). In particular, [26] shows how to define $\widehat{\gamma}$ for shape-analysis abstractions; the other two required operations, β and $\sqcup$, carry over from [23].

  In on-going work, we are investigating the feasibility of actually applying the techniques from this paper using the $\widehat{\gamma}$ from [26] to perform abstract interpretation for shape-analysis abstractions. This approach could be more precise than TVLA because it would use best abstract transformers. (We are also investigating the feasibility of using this approach to develop a more precise and scalable version of TVLA by using *assume-guarantee* reasoning. The idea is to allow arbitrary first-order formulas with transitive closure to be used to express pre- and post-conditions, and to analyze the code for each procedure separately.)
- This paper studies the problem "How can one obtain most-precise results for a *given* abstract domain?". Iterative refinement addresses a different problem: "How can one go about *improving* an abstract domain?" These are orthogonal questions.

  The question of how to go about improving an abstract domain has not yet been studied for abstract domains as rich as the ones in which our techniques can be applied. This is the subject of future work, and thus something about which one can only speculate. However, we have observed that our approach does provide a fundamental primitive for mapping values from one abstract domain to another: suppose that L_1 and L_2 are two different abstract domains that meet the conditions of the framework; given $l_1 \in L_1$, the most-precise value $l_2 \in L_2$ that overapproximates $\gamma_1(l_1)$ is obtained by $l_2 = \widehat{\alpha}_2(\widehat{\gamma}_1(l_1))$.

  The domain-changing primitive opens up several possibilities for future work. For example, counterexample-guided abstraction-refinement strategies [7, 6] identify the shortest invalid prefix of a spurious counterexample trace, and then refine the abstract domain to eliminate invalid transitions out of the last valid abstract state of the prefix. The domain-changing primitive appears to provide a systematic way to salvage information from the counterexample trace: for instance, it can be invoked to convert the last valid abstract state of the prefix into an appropriate abstract state in the refined abstract domain. Moreover, it yields the most-precise value that any conservative salvaging operation is allowed to produce.

In summary, because our results enable a better separation of concerns between the issue of how to obtain most-precise results for a *given* abstract domain and that of how to *improve* an abstract domain, they contribute to a better understanding of abstraction and symbolic approaches to abstract interpretation.

References

1. T. Ball, R. Majumdar, T. Millstein, and S.K. Rajamani. Automatic predicate abstraction of C programs. In *Conf. on Prog. Lang. Design and Impl.*, New York, NY, 2001. ACM Press.
2. T. Ball and S.K. Rajamani. The SLAM toolkit. In *Proc. Computer-Aided Verif.*, Lec. Notes in Comp. Sci., pages 260–264, 2001.
3. S. Blamey. Partial logic. In D.M. Gabbay and F. Guenthner, editors, *Handbook of Phil. Logic, 2nd. Ed., Vol. 5*, pages 261–353. Kluwer Acad., 2002.
4. S.R. Blamey. *Partial-Valued Logic*. PhD thesis, Univ. of Oxford, Oxford, Eng., 1980.
5. G. Bruns and P. Godefroid. Generalized model checking: Reasoning about partial state spaces. In *Proc. CONCUR*. Springer-Verlag, 2000.
6. E. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *Proc. Computer-Aided Verif.*, 2002.
7. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. Computer-Aided Verif.*, pages 154–169, July 2000.
8. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Symp. on Princ. of Prog. Lang.*, pages 238–252, New York, NY, 1977. ACM Press.
9. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282, New York, NY, 1979. ACM Press.
10. S. Das, D.L. Dill, and S. Park. Experience with predicate abstraction. In *Proc. Computer-Aided Verif.*, pages 160–171. Springer-Verlag, July 1999.
11. D. Detlefs, G. Nelson, and J. Saxe. Simplify. Compaq Systems Research Center, Palo Alto, CA, 1999.
12. C.N. Fischer and R.J. LeBlanc. *Crafting a Compiler*. Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1988.
13. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proc. Computer-Aided Verif.*, pages 72–83, June 1997.
14. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symp. on Princ. of Prog. Lang.*, pages 58–70, New York, NY, January 2002. ACM Press.
15. ILOG. ILOG optimization suite: White paper. ILOG S.A., Gentilly, France, 2001.
16. G.A. Kildall. A unified approach to global program optimization. In *Symp. on Princ. of Prog. Lang.*, pages 194–206, New York, NY, 1973. ACM Press.
17. N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS Notes Series NS-01-1, Univ. of Aarhus, January 2001.
18. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symp.*, pages 280–301, 2000.
19. W. McCune. *MACE User Manual and Guide*. Argonne Nat. Lab., May 2001.
20. M. Müller-Olm and H. Seidl. Polynomial constants are decidable. In *Static Analysis Symp.*, pages 4–19, 2002.
21. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
22. T. Reps, A. Loginov, and M. Sagiv. Semantic minimization of 3-valued propositional formulae. In *Proc. Symp. on Logic in Comp. Sci.*, July 2002.
23. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Trans. on Prog. Lang. and Syst.*, 24(3):217–298, 2002.
24. J. Slaney. *Finder – Finite Domain Enumerator, Version 3.0*. Aust. Nat. Univ., July 1995.
25. B.C. van Fraassen. Singular terms, truth-value gaps, and free logic. *J. Phil.*, 63(17):481–495, September 1966.
26. G. Yorsh, T. Reps, M. Sagiv, and R. Wilhelm. Logical characterizations of heap abstractions. Submitted for publication, 2003.
27. J. Zhang and H. Zhang. Generating models by SEM. In *Int. Conf. on Auto. Deduc.*, volume 1104 of *Lec. Notes in Art. Intell.*, pages 308–312. Springer-Verlag, 1996.

A Proof of Thm. 1

Lemma 1. *If $S \models \varphi$, then $\beta(S) \sqsubseteq \alpha([\![\varphi]\!])$.*

Proof. If $S \models \varphi$, then $S \in [\![\varphi]\!]$; therefore,

$$\alpha([\![\varphi]\!]) = \bigsqcup_{T \in [\![\varphi]\!]} \beta(T) \sqsupseteq \beta(S).$$

Theorem 1 *Suppose that the abstract domain has finite height h. Given input ψ, $\widehat{\alpha}(\psi)$ has the following properties:*

(i) *The loop on lines [4]–[8] in procedure $\widehat{\alpha}$ is executed at most h times.*
(ii) *$\widehat{\alpha}(\psi) = \alpha([\![\psi]\!])$ (i.e., $\widehat{\alpha}(\psi)$ computes the most-precise abstract value that overapproximates the set of concrete stores represented symbolically by ψ).*

Proof. Proof of (i): If the loop terminates after 0 iterations of the loop body, then (i) holds trivially. For the loop to terminate after 0 iterations, φ—and hence ψ—must be unsatisfiable, in which case the set of concrete structures represented by ψ is $\emptyset$; in this case, (ii) holds because $\bot$ is the most precise value that represents $\emptyset$.

Now suppose that the loop executes at least once. Let $\texttt{ans}_0 = \bot$ and, for $i \geq 1$, let $\texttt{ans}_i$ denote the value of ans after the i^{th} iteration. Similarly, let φ_i and S_i denote the values of φ and S, respectively, after the i^{th} iteration.

The $\texttt{ans}_i$ form a chain

$$\bot \sqsubseteq \texttt{ans}_1 \sqsubseteq \texttt{ans}_2 \sqsubseteq \ldots \sqsubseteq \texttt{ans}_i \sqsubseteq \ldots$$

From Defn. 6, $\beta(S_1)$ cannot equal $\bot$; hence $\bot \sqsubset \texttt{ans}_1$.

We will show that the $\texttt{ans}_i$ actually form a *strictly* increasing sequence. Because the lattice is of finite height h, this implies that (a) the chain $\texttt{ans}_i$ is of length $k \leq h + 1$, and (b) the loop is executed at most h times.

For the sake of argument, suppose that the $\texttt{ans}_i$ do not form a strictly increasing sequence; let $\texttt{ans}_i = \texttt{ans}_{i+1}$ be the first repetition:

$$\bot \sqsubset \texttt{ans}_1 \sqsubset \texttt{ans}_2 \sqsubset \ldots \sqsubset \texttt{ans}_i = \texttt{ans}_{i+1}.$$

By lines [5] and [6], $\texttt{ans}_{i+1} = \texttt{ans}_i \sqcup \beta(S_{i+1})$, where $S_{i+1} \models \varphi_i \wedge \neg\widehat{\gamma}(\texttt{ans}_i)$. Thus, $\texttt{ans}_i = \texttt{ans}_i \sqcup \beta(S_{i+1})$, which means that

$$\begin{aligned} \texttt{ans}_i &\sqsupseteq \beta(S_{i+1}) \\ &= \alpha(\{S_{i+1}\}). \end{aligned}$$

Applying γ to both sides, we have

$$\begin{aligned} \gamma(\texttt{ans}_i) &\supseteq (\gamma \circ \alpha)(\{S_{i+1}\}) \\ &\supseteq \{S_{i+1}\}. \end{aligned}$$

This implies that $S_{i+1} \in \gamma(\texttt{ans}_i)$, from which we conclude that

$$S_{i+1} \models \widehat{\gamma}(\texttt{ans}_i).$$

However, this contradicts

$$S_{i+1} \models \varphi_i \wedge \neg\widehat{\gamma}(\texttt{ans}_i),$$

and thus the assumption that the $\texttt{ans}_i$ do not form a strictly increasing sequence cannot be true.

Proof of (ii): Let $k \geq 0$ be the iteration on which the loop terminates. This part of the proof will be handled in two cases, in which we show (a) $\texttt{ans}_k \sqsubseteq \alpha([\![\psi]\!])$ and (b) $\texttt{ans}_k \sqsupseteq \alpha([\![\psi]\!])$.

Part (a): Show that $\texttt{ans}_k \sqsubseteq \alpha([\![\psi]\!])$.

We will show that $\texttt{ans}_i \sqsubseteq \alpha([\![\psi]\!])$ holds on each execution of line [4].

Base case ($i = 0$): $\bot \sqsubseteq \alpha([\![\psi]\!])$.

Induction step: Assume that $\texttt{ans}_i \sqsubseteq \alpha([\![\psi]\!])$ holds on each execution of line [4]; show that $\texttt{ans}_{i+1} \sqsubseteq \alpha([\![\psi]\!])$ holds.

By line [5], $S_{i+1} \models \varphi_i$. However, φ_i is of the form $\psi \wedge \neg\widehat{\gamma}(\mathtt{ans}_1) \wedge \ldots \wedge \neg\widehat{\gamma}(\mathtt{ans}_i)$. Thus,

$$\begin{array}{l} S_{i+1} \models \psi \\ \beta(S_{i+1}) \sqsubseteq \alpha([\![\psi]\!]) \quad \text{by Lem. 1} \end{array}$$

Consequently, $\mathtt{ans}_{i+1} = \mathtt{ans}_i \sqcup \beta(S_{i+1}) \sqsubseteq \alpha([\![\psi]\!])$.
Part (b): Show that $\mathtt{ans}_k \sqsupseteq \alpha([\![\psi]\!])$.
Suppose for the sake of argument that $\mathtt{ans}_k \sqsupseteq \alpha([\![\psi]\!])$ does not hold (*). Then by Part (a), we must have $\mathtt{ans}_k \sqsubset \alpha([\![\psi]\!])$, which means that there must exist $T \models \psi$ such that $\beta(T) \not\sqsubseteq \mathtt{ans}_k$. Because the $\mathtt{ans}_i$ form a chain,

$$\beta(T) \not\sqsubseteq \mathtt{ans}_i, \text{ for } 1 \leq j \leq k. \qquad (**)$$

From Part (i), we know that the loop terminates, and so φ_k is unsatisfiable at the end of procedure $\widehat{\alpha}$. In the remainder of the proof, we will show that $T \models \varphi_k$, which is a contradiction (because it implies that the loop did not terminate on iteration k), and hence assumption (*) must be incorrect.

We demonstrate the contradiction by showing that $T \models \varphi_i$ holds for each execution of line [4]:
Base case ($i = 0$): $\varphi_0 \equiv \psi$, and therefore $T \models \varphi_0$.
Induction step: Assume that $T \models \varphi_i$; show that $T \models \varphi_{i+1}$.
Because $\varphi_{i+1} = \varphi_i \wedge \neg\widehat{\gamma}(\mathtt{ans}_{i+1})$, we need to show that $T \models \neg\widehat{\gamma}(\mathtt{ans}_{i+1})$. Suppose, on the contrary, that $T \models \widehat{\gamma}(\mathtt{ans}_{i+1})$ (***).

$$\begin{array}{lll} \beta(T) \subseteq \alpha([\![\widehat{\gamma}(\mathtt{ans}_{i+1})]\!]) & & \text{by Lem. 1} \\ \quad = \alpha(\gamma(\mathtt{ans}_{i+1})) & & \text{by the definition of } \widehat{\gamma} \\ \quad \sqsubseteq \mathtt{ans}_{i+1} & & \text{because } \alpha \circ \gamma \sqsubseteq id \end{array}$$

This contradicts (**), which means that (***) cannot hold. Therefore, $T \models \neg\widehat{\gamma}(\mathtt{ans}_{i+1})$.

□