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 decision procedure, and obtains the best abstract value for a set of concrete stores that are represented symbolically, using a logical formula.

1 Introduction

Abstract interpretation [6] 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 [7], 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 [11] showed that decision procedures 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 [2] and other systems [8, 12].) 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 other than predicate-abstraction domains. 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 [14]. 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 $\widehat{\alpha\mathrm{Post}}[\tau]$,[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 contributions of the paper can be summarized as follows:

- The paper shows how some of the benefits enjoyed by predicate abstraction can be extended to arbitrary finite-height abstract domains. In particular, we describe methods for each of the operations needed to carry out abstract interpretation.
- With some logics, the result of applying $\mathrm{Post}[\tau]$ to a given set of concrete stores (represented symbolically) can also be expressed symbolically, as a formula ϕ'. In this case, we can proceed by computing $\widehat{\alpha}(\phi')$. For other logics, however, ϕ' cannot be expressed symbolically without passing to a more powerful logic. For instance,
  - If sets of concrete stores are represented with quantifier-free first-order logic, it may require quantified first-order logic to express $\mathrm{Post}[\tau]$.
  - If sets of concrete stores are represented with a decidable subset of first-order logic, it may require second-order logic to express $\mathrm{Post}[\tau]$.

  In such situations, the procedure that we give to compute $\widehat{\alpha\mathrm{Post}}[\tau]$ provides a way to compute the best transformer while staying within the original logic.

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 makes some additional observations about the work. Sect. 7 discusses related work.

2 Motivating Examples

This section presents several examples to motivate the work. The treatment here is at a semi-formal level; a more formal treatment is given in later sections. (This section assumes a certain amount of background on abstract interpretation; some readers may find it helpful to consult Sect. 3 before reading this section.)

The example concerns a simple concrete domain: let Var denote the set of variables in the program being analyzed; the concrete domain is $2^{Var \to \mathcal{Z}}$.

Predicate Abstraction A predicate-abstraction domain $\mathcal{PA}[\mathcal{B}]$ is based on a set $\mathcal{B}$ of predicate names, each of which has an associated defining formula: $\mathcal{B} = \{B_j \stackrel{\text{def}}{=} \varphi_j \mid 1 \leq j \leq k\}$. Each value in $\mathcal{PA}[\mathcal{B}]$ is a set of possibly negated symbols drawn from $\mathcal{B}$, where each symbol B_j is either present in positive or negative form (but not both), or absent entirely. For instance, with $\mathcal{B} = \{B_1 \stackrel{\text{def}}{=} \varphi_1, B_2 \stackrel{\text{def}}{=} \varphi_2, B_3 \stackrel{\text{def}}{=} \varphi_3\}$, values in $\mathcal{PA}[\mathcal{B}]$ include $\{\neg B_1, B_2, \neg B_3\}$, $\{B_1, B_2\}$, $\{\neg B_3\}$, and $\emptyset$.

[3] We use the diacritic $\widehat{\ }$ on a symbol to indicate an operation that either produces or operates on a symbolic representation of a set of concrete stores.

We will use a predicate-abstraction domain in which there is a Boolean predicate $B \stackrel{\text{def}}{=} (x = c)$ for each $\mathtt{x} \in \mathit{Var}$ 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 is based on the predicate set $\{B_1 \stackrel{\text{def}}{=} (y = 1), B_2 \stackrel{\text{def}}{=} (y = 3), B_3 \stackrel{\text{def}}{=} (y = 4), B_4 \stackrel{\text{def}}{=} (x = 1), B_5 \stackrel{\text{def}}{=} (x = 3), B_6 \stackrel{\text{def}}{=} (x = 4)\}$. Note that this domain 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 abstract value $\{\neg B_1, B_2, \neg B_3, \neg B_4, \neg B_5, \neg B_6\}$, which provides limited information about the value of x.

Our choice of predicate-abstraction domain $\mathcal{PA}[\{B_1, B_2, B_3, B_4, B_5, B_6\}]$ was made solely for the sake of simplicity. With a different choice of predicates, we could have retained a greater or lesser amount of information about the value of x in the state after program (2); however, there would always be some program that gives rise to a state in which information is lost.

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, given (a specification of) a set of concrete stores as a logical formula ψ [11]. We will call this procedure $\widehat{\alpha}_{\mathrm{PA}}$; it relies on the aid of a decision procedure, and can be defined as follows:

$$\widehat{\alpha}_{\mathrm{PA}}(\psi) = \{B_j \mid \psi \Rightarrow \varphi_j \text{ is valid}\} \cup \{\neg B_j \mid \psi \Rightarrow \neg\varphi_j \text{ is valid}\} \tag{3}$$

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 $\{\neg B_1, B_2, \neg B_3, \neg B_4, \neg B_5, \neg B_6\}$, the decision procedure must demonstrate that the following formulas are valid:

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

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. In particular, we will demonstrate this for the abstract domain used in the constant-propagation problem: $(\mathit{Var} \to \mathcal{Z}^{\top})_{\perp}$. The abstract value $\perp$ represents $\emptyset$; an abstract value such as $[\mathtt{x} \mapsto \mathtt{0}, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto \mathtt{0}]$ represents all concrete stores in which program variables x and z are both mapped to 0.[4]

The procedure to implement $\widehat{\alpha}$ for the constant-propagation domain, which we call $\widehat{\alpha}_{\mathrm{CP}}$, is actually an instance of a general procedure for implementing $\widehat{\alpha}$ functions that applies to a family of Galois connections. It is presented in Fig. 1; $\widehat{\alpha}_{\mathrm{CP}}$ is the instance of this procedure in which the return type L is $(\mathit{Var} \to \mathcal{Z}^{\top})_{\perp}$, and "structure" in line [5] means "concrete store".

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

```
[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. 1. An algorithm to obtain, with the aid of a decision procedure, a most-precise abstract value that overapproximates a set of concrete stores. In Sect. 2, the return type L is $(\mathit{Var} \to \mathcal{Z}^{\top})_{\perp}$, 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 decision procedure (see line [5] of Fig. 1). We make one assumption that goes beyond what is assumed in predicate abstraction, namely, we assume that the decision procedure 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 $\mathtt{ProveValid}(\neg\varphi)$ 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 $S \in \mathit{Var} \to \mathcal{Z}$ to an abstract value $\beta(S) \in (\mathit{Var} \to \mathcal{Z}^{\top})_{\perp}$. For instance, β maps the concrete store $[x \mapsto 13, y \mapsto 3]$ to the abstract value $[\mathtt{x} \mapsto 13, \mathtt{y} \mapsto 3]$.
- $\sqcup$ is the join operation in $(\mathit{Var} \to \mathcal{Z}^{\top})_{\perp}$. For instance,

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

- 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

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

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

  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.

To see how $\widehat{\alpha}_{\mathrm{CP}}$ works, consider the program

$$\begin{array}{l} \mathtt{z} := 0 \\ \mathtt{x} := \mathtt{y} * \mathtt{z} \end{array} \tag{4}$$

and suppose that ψ is the formula $(z = 0) \wedge (x = y * z)$, which captures the final state of program (4). The following sequence of operations would be performed during the invocation of $\widehat{\alpha}_{\mathrm{CP}}((z=0) \wedge (x = y * z))$:

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

At this point the loop terminates, and $\widehat{\alpha}_{\mathrm{CP}}$ returns the abstract value $[\mathtt{x} \mapsto \mathtt{0}, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto \mathtt{0}]$. 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, $\mathtt{0}$ is a multiplicative annihilator that supersedes $\top$: $\mathtt{0} = \top * \mathtt{0}$.

In general, $\widehat{\alpha}(\psi)$ carries out a process of successive approximation, making repeated calls to a decision procedure. 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. 1 during Iteration 1 of the second example of $\widehat{\alpha}_{\mathrm{CP}}(\psi)$, `ans` has the value $[\mathtt{x} \mapsto \mathtt{0}, \mathtt{y} \mapsto \mathtt{43}, \mathtt{z} \mapsto \mathtt{0}]$, and the update to φ, $\varphi := \varphi \wedge \neg\widehat{\gamma}(\mathsf{ans})$, 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. 1 during Iteration 2 of the second example of $\widehat{\alpha}_{\mathrm{CP}}(\psi)$, `ans`'s value is changed from $[\mathtt{x} \mapsto \mathtt{0}, \mathtt{y} \mapsto \mathtt{43}, \mathtt{z} \mapsto \mathtt{0}]$ to $[\mathtt{x} \mapsto \mathtt{0}, \mathtt{y} \mapsto \mathtt{43}, \mathtt{z} \mapsto \mathtt{0}] \sqcup \beta([x \mapsto 0, y \mapsto 46, z \mapsto 0]) = [\mathtt{x} \mapsto \mathtt{0}, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto \mathtt{0}]$. 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. 2 presents a sequence of diagrams that illustrate schematically algorithm $\widehat{\alpha}$ from Fig. 1.

3 Terminology and Notation

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 $C = \{c_1, \ldots, c_n\}$*

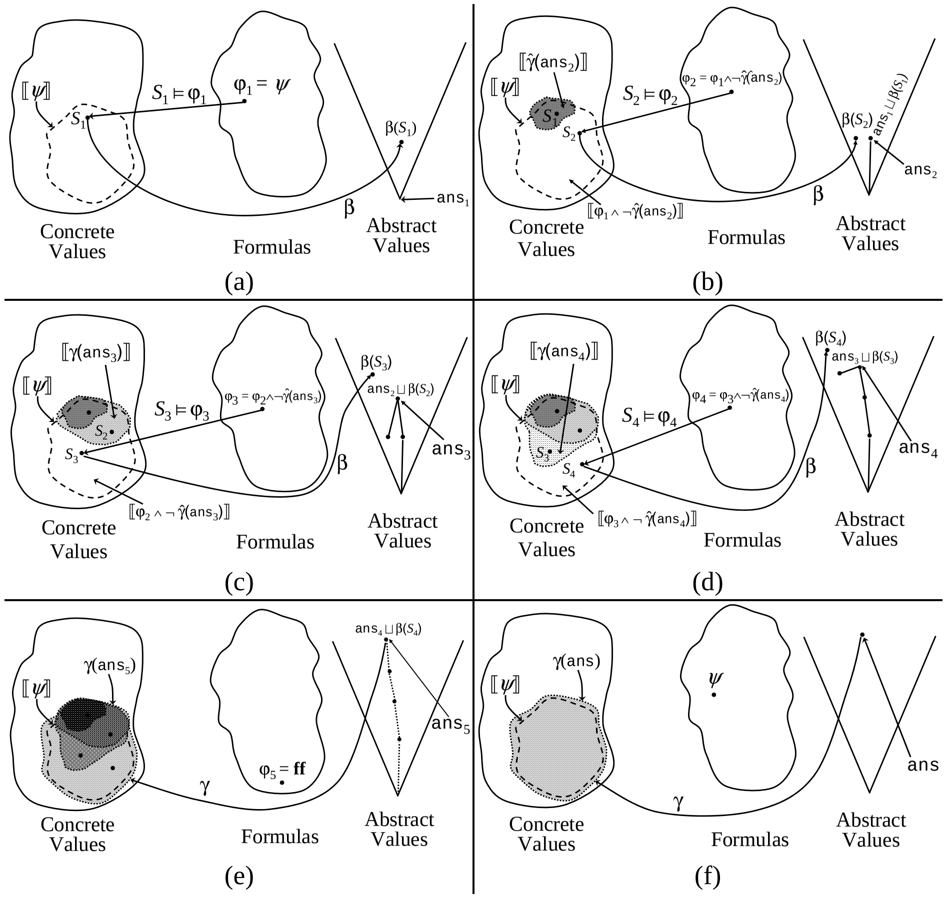

Fig. 2. Schematic diagrams that illustrate the process carried out by algorithm $\widehat{\alpha}(\psi)$ from Fig. 1; φ_i, S_i, and $\texttt{ans}_i$ denote the values of φ, S, and $\texttt{ans}$ during the i^{th} iteration. (a) Initially, φ_1 is set to ψ and $\texttt{ans}_1$ is set to $\bot$; S_1 is a structure such that $S_1 \models \varphi_1$. (b) $\texttt{ans}_2$ is set to $\texttt{ans}_1 \sqcup \beta(S_1) = \beta(S_1)$; φ_2 is set to $\varphi_1 \wedge \neg\widehat{\gamma}(\mathsf{ans}_2)$; S_2 is a structure such that $S_2 \models \varphi_2$. Note that S_2 belongs to $[\![\varphi_2]\!] = [\![\varphi_1 \wedge \neg\widehat{\gamma}(\mathsf{ans}_2)]\!]$. (c) $\texttt{ans}_3$ is set to $\texttt{ans}_2 \sqcup \beta(S_2)$; φ_3 is set to $\varphi_2 \wedge \neg\widehat{\gamma}(\mathsf{ans}_3)$; S_3 is a structure such that $S_3 \models \varphi_3$. (d) $\texttt{ans}_4$ is set to $\texttt{ans}_3 \sqcup \beta(S_3)$; φ_4 is set to $\varphi_3 \wedge \neg\widehat{\gamma}(\mathsf{ans}_4)$; S_4 is a structure such that $S_4 \models \varphi_4$. (e) $\texttt{ans}_5$ is set to $\texttt{ans}_4 \sqcup \beta(S_4)$; φ_5 is set to $\varphi_4 \wedge \neg\widehat{\gamma}(\mathsf{ans}_5)$. In the case portrayed here, the loop terminates at this point because $\varphi_5 = \mathbf{ff}$. The desired answer is held in $\texttt{ans}_5$. (f) $\widehat{\alpha}(\psi)$ obtains the most-precise abstract value $\texttt{ans}$ that overapproximates $[\![\psi]\!]$.

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. A **logical structure** *over* **vocabulary** *$V = \langle P, C, F\rangle$ 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 $f \in F_i$, $\iota_f(f)\colon U^i \to U$ maps i-tuples into an individual.*

Typically, some subset of the predicate symbols, constant symbols, and function symbols have an interpretation that is fixed in advance; this defines a family of intended models. We denote the (infinite) set of structures over V, where the interpretations of $I \subseteq V$ are fixed in advance, by $\mathit{ConcreteStruct}[V, I]$.

Example 1. In Sect. 2, we considered concrete stores to be members of $\mathit{Var} \to \mathcal{Z}$. 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 $\langle \mathcal{Z}, \emptyset, \iota_{\mathit{Var}}, \emptyset \rangle$ over vocabulary $\langle \mathit{IntPreds}, \mathit{Var} \cup \mathit{IntConsts}, \mathit{IntFuncs} \rangle$, where ι_{Var} is a mapping of program variables to integers, and the symbols in $\mathit{IntPreds} = \{<, \leq, =, \neq, \geq, >, \ldots\}$, $\mathit{IntConsts} = \{0, -1, 1, -2, 2, \ldots\}$, and $\mathit{IntFuncs} = \{+, -, *, /, \ldots\}$ have their usual meanings. For instance, an example concrete store for a program in which $\mathit{Var} = \{x, y, z\}$ is

$$\langle \mathcal{Z}, \emptyset, [x \mapsto 0, y \mapsto 2, z \mapsto 0], \emptyset \rangle. \tag{5}$$

Henceforth, we abbreviate a store such as (5) by $\iota_c = [x \mapsto 0, y \mapsto 2, z \mapsto 0]$.

To manipulate sets of structures symbolically, we use formulas of first-order logic with equality. If S is a logical structure and φ is a closed formula, the notation $S \models \varphi$ means that S satisfies φ according to the standard Tarskian semantics for first-order logic (e.g., see [10]). We use $[\![\varphi]\!]$ to denote the set of concrete structures that satisfy φ: $[\![\varphi]\!] = \{S \mid S \in \mathit{ConcreteStruct}[V, I], S \models \varphi\}$.

Example 2.

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

Definition 2. *A* **complete join semilattice** $L = \langle L, \sqsubseteq, \bigsqcup, \bot \rangle$ *is a partially ordered set with partial order $\sqsubseteq$, such that for every subset X of L, L contains a least upper bound (or* **join***), denoted by $\bigsqcup X$.*

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

Definition 3. *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.*

We now define an abstract domain by means of a representation function [18].

Definition 4. *Given a complete join semilattice $L = \langle L, \sqsubseteq, \bigsqcup, \bot \rangle$ and a* **representation function** *$\beta\colon \mathit{ConcreteStruct}[V, I] \to L$ such that for all $S \in$*

$\mathit{ConcreteStruct}[V,I]$ $\beta(S) \neq \bot$, a Galois connection $2^{\mathit{ConcreteStruct}[V,I]} \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} L$ is defined by extending β pointwise, i.e., for $XS \subseteq \mathit{ConcreteStruct}[V,I]$ and $l \in L$,

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

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, $(\mathit{Var} \to \mathcal{Z}^{\top})_{\bot}$. 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,

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

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 $\mathtt{y}$, 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. 1. $\widehat{\alpha}(\psi)$ 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 decision procedure on each iteration.

The assumptions of the framework are rather minimal:

- The concrete domain is the power set of $\mathit{ConcreteStruct}[V,I]$.
- The concrete and abstract domains are related by a Galois connection defined by a representation function β that maps a structure $S \in \mathit{ConcreteStruct}[V,I]$ 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

$$[\![\widehat{\gamma}(l)]\!] = \gamma(l). \tag{6}$$

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 $\mathtt{y}$, 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

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

and thus Eqn. (6) 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, $\widehat{\alpha}(\psi) = \alpha(\llbracket \psi \rrbracket)$.

Note that a logical formula ψ represents the set of concrete stores $\llbracket \psi \rrbracket$; thus, $\alpha(\llbracket \psi \rrbracket)$ (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. 1.

Example 5. A trace of a call on $\widehat{\alpha}$ for the constant-propagation domain $(\mathit{Var} \to \mathcal{Z}^\top)_\perp$ was presented 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) $\widehat{\alpha}(\psi) = \alpha(\llbracket \psi \rrbracket)$ *(i.e., $\widehat{\alpha}(\psi)$ computes the most-precise abstract value that overapproximates the set of concrete stores represented symbolically by ψ).*

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 "double 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.[5] The satisfaction relation for a two-vocabulary formula τ will be written as $\langle S, S' \rangle \models \tau$, where S and S' are structures over vocabularies $V = \langle P, C, F \rangle$ and $V' = \langle P', C', F' \rangle$, respectively; $\langle S, S' \rangle$ is called a *two-vocabulary structure*.

[5] For economy of notation, we will not duplicate the symbols $I \subseteq V$ whose interpretation is fixed in advance.

Example 6. The formula that expresses the semantics of an assignment $\mathtt{x} := \mathtt{y} * \mathtt{z}$ with respect to stores over vocabulary $\langle \mathit{IntPreds}, \mathit{Var} \cup \mathit{Var}' \cup \mathit{IntConsts}, \mathit{IntFuncs} \rangle$, denoted by $\tau_{\mathtt{x:=y*z}}$, can be specified as $\tau_{\mathtt{x:=y*z}} \stackrel{\text{def}}{=} (x' = y * z) \wedge (y' = y) \wedge (z' = z)$.

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

$$\begin{array}{ll} \beta\colon \mathit{ConcreteStruct}[V, I] \to L & \beta'\colon \mathit{ConcreteStruct}[V', I] \to L' \\ \widehat{\gamma}\colon L \to \mathit{Formula}[V] & \widehat{\gamma}'\colon L' \to \mathit{Formula}[V'] \end{array}$$

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.[6]

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. 3. After φ is initialized to $\widehat{\gamma}(l) \wedge \tau$ in line [3], $\widehat{\alpha\mathrm{Post}}[\tau]$ 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 $\widehat{\alpha\mathrm{Post}}[\tau]$, 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' α̂Post(two-vocabulary formula τ over V ∪ V', L l) {
[2]     ans' := ⊥'
[3]     φ := γ̂(l) ∧ τ
[4]     while (φ is satisfiable) {
[5]       Select a two-vocabulary structure ⟨S,S'⟩ s.t. ⟨S,S'⟩ ⊨ φ
[6]       ans' := ans' ⊔ β'(S')
[7]       φ := φ ∧ ¬γ̂'(ans')
[8]     }
[9]     return ans'
[10] }
```

Fig. 3. An algorithm that implements $\widehat{\alpha\mathrm{Post}}[\tau](l)$.

[6] Alternatively, we could have used a single abstract domain, L, and the definitions

$$\begin{array}{ll} \beta\colon \mathit{ConcreteStruct}[V, I] \to L & \beta'\colon \mathit{ConcreteStruct}[V', I] \to L \\ \widehat{\gamma}\colon L \to \mathit{Formula}[V] & \widehat{\gamma}'\colon L \to \mathit{Formula}[V'] \end{array}$$

The motivation for using two abstract domains is to eliminate a possible source of confusion in the examples. By using separate abstract domains L and L', primed symbols always distinguish next-state abstract values from present-state ones.

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 $\tau_{\mathtt{x:=y*z}}$ is the formula $(x' = y * z) \wedge (y' = y) \wedge (z' = z)$. Fig. 4 shows why we have

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

$$\begin{array}{lrll}
\text{Initialization:} & \mathtt{ans}' &:= \bot' & \\
 & \varphi &:= (z = 0) \wedge (x' = y * z) \wedge (y' = y) \wedge (z' = z) & \\
\text{Iteration 1:} & \langle S, S' \rangle &:= \iota_c = \begin{bmatrix} x \mapsto 5, y \mapsto 17, z \mapsto 0 \\ x' \mapsto 0, y' \mapsto 17, z' \mapsto 0 \end{bmatrix} & \text{// Some satisfying structure} \\
 & \mathtt{ans}' &:= [\mathtt{x}' \mapsto 0, \mathtt{y}' \mapsto 17, \mathtt{z}' \mapsto 0] & \\
 & \widehat{\gamma}'(\mathtt{ans}') &= (x' = 0) \wedge (y' = 17) \wedge (z' = 0) & \\
 & \varphi &:= (z = 0) \wedge (x' = y * z) \wedge (y' = y) \wedge (z' = z) & \\
 & & \quad \wedge ((x' \neq 0) \vee (y' \neq 17) \vee (z' \neq 0)) & \\
 & &= (z = 0) \wedge (x' = y * z) \wedge (y' = y) \wedge (z' = z) \wedge (y' \neq 17) & \\
\text{Iteration 2:} & \langle S, S' \rangle &:= \iota_c = \begin{bmatrix} x \mapsto 12, y \mapsto 99, z \mapsto 0 \\ x' \mapsto 0, y' \mapsto 99, z' \mapsto 0 \end{bmatrix} & \text{// Some satisfying structure} \\
 & \mathtt{ans}' &:= [\mathtt{x}' \mapsto 0, \mathtt{y}' \mapsto 17, \mathtt{z}' \mapsto 0] \sqcup [\mathtt{x}' \mapsto 0, \mathtt{y}' \mapsto 99, \mathtt{z}' \mapsto 0] & \\
 & &= [\mathtt{x}' \mapsto 0, \mathtt{y}' \mapsto \top, \mathtt{z}' \mapsto 0] & \\
 & \widehat{\gamma}'(\mathtt{ans}') &= (x' = 0) \wedge (z' = 0) & \\
 & \varphi &:= (z = 0) \wedge (x' = y * z) \wedge (y' = y) \wedge (z' = z) \wedge (y' \neq 17) & \\
 & & \quad \wedge ((x' \neq 0) \vee (z' \neq 0)) & \\
 & &= \mathbf{ff} & \\
\text{Iteration 3:} & & \varphi \text{ is unsatisfiable} & \\
\text{Return value:} & & [\mathtt{x}' \mapsto 0, \mathtt{y}' \mapsto \top, \mathtt{z}' \mapsto 0] &
\end{array}$$

Fig. 4. Operations performed during a call $\widehat{\alpha\mathrm{Post}}[\tau_{\mathtt{x:=y*z}}]([\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0])$.

Theorem 2. *Suppose that the abstract domain has finite height of at most h. Given inputs τ and l, $\widehat{\alpha\mathrm{Post}}[\tau](l)$ has the following properties:*

(i) *The loop on lines [4]–[8] in procedure $\widehat{\alpha\mathrm{Post}}[\tau](l)$ is executed at most h times.*
(ii) $\widehat{\alpha\mathrm{Post}}[\tau](l) = \alpha(\mathrm{Post}[\tau](\gamma(l)))$ *(i.e., $\widehat{\alpha\mathrm{Post}}[\tau](l)$ computes the most-precise abstract value in L' that overapproximates* $\mathrm{Post}[\tau](\gamma(l))$*).*

The operator $\mathrm{Pre}[\tau]$ can be implemented using a procedure that is dual to Fig. 3.

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:

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

Example 8.

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

6 Discussion

This paper shows how the most-precise versions of the basic operations needed to create an abstract interpreter are, under certain conditions, implementable. These techniques use the idea of considering a first-order formula φ as a device for describing (or accepting) a set of concrete structures, namely, the set of structures that satisfy φ. Not every subset of concrete structures can be described by a first-order formula; however, it is straightforward to generalize the approach to other types of logics, which can be considered as alternative structure-description formalisms (possibly more powerful, possibly less powerful). For the basic approach to carry over, all that is required is that a decision procedure exist for the logic.

Automatic theorem provers—such as MACE [16], SEM [20], and Finder [19]—can be used to implement the procedures presented in this paper because they return counterexamples to validity: a counterexample to the validity of $\neg\varphi$ is a structure that satisfies φ. Such tools also exist for logics other than first-order logic; for example, MONA [15] can generate counterexamples for formulas in weak monadic second-order logic.

Some tools, such as Simplify [9] and SVC [1], provide counterexamples in symbolic form, i.e., as a formula. The formula represents a *set* of counterexamples; any structure that satisfies the formula is a counterexample to the query. For example, if φ is $x \geq y$ at line [5] of Fig. 1, the value returned would be the formula $(x \geq y)$ itself, rather than a particular satisfying structure, such as $[x \mapsto 7, y \mapsto 3]$. This presents an obstacle because at line [6] β requires an argument that is a single structure. In the case of quantifier-free first-order logic with linear arithmetic, such a structure can be obtained by feeding the counterexample formula to a solver for mixed-integer programming, such as CPLEX [13].

```
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. 5. A program with correlated branches.

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 the approach: it can be used to generate the best transformer for combined domains, such as reduced cardinal product and those created using other domain constructors [7]. 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. 5.

7 Related Work

This paper is most closely related to past work on predicate abstraction, which also uses decision procedures 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, procedure $\widehat{\alpha}$ of Fig. 1 uses an approach that is fundamentally different from the one used in predicate abstraction. Although both approaches use multiple calls on a decision procedure 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. 1 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.

Procedure $\widehat{\alpha}$ is also related to an algorithm used in machine learning, called Find-S [17, Section 2.4]. In machine-learning terminology, both algorithms search a space of "hypotheses" to find the most-specific hypothesis that is consistent with the positive training examples of the "concept" to be learned. Find-S receives a sequence of training examples, and generalizes its current hypothesis each time it is presented with a positive training example that falls outside its current hypothesis. The problem settings for the two algorithms are slightly different: Find-S receives a sequence of positive and negative examples of the concept. $\widehat{\alpha}$ already starts with a precise statement of the concept in hand, namely, the formula ψ; on each iteration, the decision procedure is used to generate the next (positive) training example.

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 [3] or BLAST [12]?". We do not have a complete answer to this question; however, a few observations can be made:

- Our results extend ideas employed in the setting of predicate abstraction to a more general setting.
- 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 logical machinery (i.e., the availability of a decision procedure), but do not rely on heuristics-based machinery for changing the abstract domain in use.
- 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 [5, 4] 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. Stanford validity checker. "http://verify.stanford.edu/SVC/", 1999.
2. T. Ball, R. Majumdar, T. Millstein, and S.K. Rajamani. Automatic predicate abstraction of C programs. In *Prog. Lang. Design and Impl.*, New York, NY, 2001. ACM Press.
3. T. Ball and S.K. Rajamani. The SLAM toolkit. In *Computer-Aided Verif.*, Lec. Notes in Comp. Sci., pages 260–264, 2001.
4. E. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *Computer-Aided Verif.*, 2002.
5. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer-Aided Verif.*, pages 154–169, July 2000.
6. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Princ. of Prog. Lang.*, 1977.
7. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Princ. of Prog. Lang.*, pages 269–282, New York, NY, 1979. ACM Press.
8. S. Das, D.L. Dill, and S. Park. Experience with predicate abstraction. In *Computer-Aided Verif.*, pages 160–171. Springer-Verlag, July 1999.
9. D. Detlefs, G. Nelson, and J. Saxe. Simplify. Compaq Systems Research Center, Palo Alto, CA, 1999.
10. H.B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.
11. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer-Aided Verif.*, pages 72–83, June 1997.
12. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Princ. of Prog. Lang.*, pages 58–70, New York, NY, January 2002. ACM Press.
13. ILOG. ILOG optimization suite: White paper. ILOG S.A., Gentilly, France, 2001.
14. G.A. Kildall. A unified approach to global program optimization. In *Princ. of Prog. Lang.*, pages 194–206, New York, NY, 1973. ACM Press.
15. N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS Notes Series NS-01-1, Univ. of Aarhus, January 2001.
16. W. McCune. *MACE User Manual and Guide*. Argonne Nat. Lab., May 2001.
17. T.M. Mitchell. *Machine Learning*. WCB/McGraw-Hill, Boston, MA, 1997.
18. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
19. J. Slaney. *Finder – Finite Domain Enumerator, Version 3.0*. Aust. Nat. Univ., July 1995.
20. 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.