Bilateral Algorithms for Symbolic Abstraction

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

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

Abstract. Given a concrete domain $\mathcal{C}$, a concrete operation $\tau : \mathcal{C} \to \mathcal{C}$, and an abstract domain $\mathcal{A}$, a fundamental problem in abstract interpretation is to find the *best abstract transformer* $\tau^{\#} : \mathcal{A} \to \mathcal{A}$ that over-approximates τ. This problem, as well as several other operations needed by an abstract interpreter, can be reduced to the problem of *symbolic abstraction*: the symbolic abstraction of a formula φ in logic $\mathcal{L}$, denoted by $\widehat{\alpha}(\varphi)$, is the best value in $\mathcal{A}$ that over-approximates the meaning of φ. When the concrete semantics of τ is defined in $\mathcal{L}$ using a formula φ_τ that specifies the relation between input and output states, the best abstract transformer $\tau^{\#}$ can be computed as $\widehat{\alpha}(\varphi_\tau)$.

In this paper, we present a new framework for performing symbolic abstraction, discuss its properties, and present several instantiations for various logics and abstract domains. The key innovation is to use a *bilateral* successive-approximation algorithm, which maintains both an over-approximation and an under-approximation of the desired answer. The advantage of having a non-trivial over-approximation is that it makes the technique resilient to timeouts.

1 Introduction

For several years, we have been investigating connections between abstract interpretation and logic—in particular, how to harness decision procedures to obtain algorithms for several fundamental primitives used in abstract interpretation. This paper presents new results on this topic.

Like several previous papers [24, 16, 11, 31], this paper concentrates on the problem of developing an algorithm for *symbolic abstraction*: the symbolic abstraction of a formula φ in logic $\mathcal{L}$, denoted by $\widehat{\alpha}(\varphi)$, is the best value in a given abstract domain $\mathcal{A}$ that over-approximates the meaning of φ [24]. To be more precise, given a formula $\varphi \in \mathcal{L}$, let $[\![\varphi]\!]$ denote the meaning of φ—i.e., the set of concrete states that satisfy φ. Then $\widehat{\alpha}(\varphi)$ is the unique value $a \in \mathcal{A}$ such that (i) $[\![\varphi]\!] \subseteq \gamma(a)$, and (ii) for all $a' \in \mathcal{A}$ for which $[\![\varphi]\!] \subseteq \gamma(a')$, $a \sqsubseteq a'$. In this paper, we present a new framework for performing symbolic abstraction, discuss its properties, and present several instantiations for various logics and abstract domains.

Several key operations needed by an abstract interpreter can be reduced to symbolic abstraction. For instance, one use of symbolic abstraction is to bridge the gap between concrete semantics and an abstract domain. Cousot and Cousot [5] gave a *specification* of the most-precise abstract interpretation of a concrete operation τ that is possible in a given abstract domain:

> Given a Galois connection $\mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$, the *best abstract transformer*, $\tau^{\#} : \mathcal{A} \to \mathcal{A}$, is the most precise abstract operator possible that over-approximates τ. $\tau^{\#}$ can be expressed as follows: $\tau^{\#} = \alpha \circ \tau \circ \gamma$.

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

In contrast, it is often convenient to use a logic $\mathcal{L}$ to state the concrete semantics of transformer τ as a formula $\varphi_\tau \in \mathcal{L}$ that specifies the relation between input and output states. Then, using an *algorithm for symbolic abstraction*, a representation of $\tau^{\#}$ can be computed as $\widehat{\alpha}(\varphi_\tau)$.

To see how symbolic abstraction can yield better results than conventional approaches to the creation of abstract transformers, consider an example from machine-code analysis: the x86 instruction "`add bh,al`" adds `al`, the low-order byte of 32-bit register `eax`, to `bh`, the second-to-lowest byte of 32-bit register `ebx`. The semantics of this instruction can be expressed in quantifier-free bit-vector (QFBV) logic as

$$\varphi_I \overset{\text{def}}{=} \texttt{ebx}' = \begin{pmatrix} (\texttt{ebx} \ \& \ \texttt{0xFFFF00FF}) \\ \mid ((\texttt{ebx} + 256 * (\texttt{eax} \ \& \ \texttt{0xFF})) \ \& \ \texttt{0xFF00}) \end{pmatrix} \wedge \texttt{eax}' = \texttt{eax}, \qquad (1)$$

where "&" and "|" denote bitwise-and and bitwise-or, respectively. Eqn. (1) shows that the semantics of the instruction involves non-linear bit-masking operations.

Now suppose that abstract domain $\mathcal{A}$ is the domain of affine relations over integers mod 2^{32} [11]. For this abstract domain, $\widehat{\alpha}(\varphi_I)$ is $(2^{16}\texttt{ebx}' = 2^{16}\texttt{ebx} + 2^{24}\texttt{eax}) \wedge (\texttt{eax}' = \texttt{eax})$, which captures the relationship between the low-order two bytes of `ebx` and the low-order byte of `eax`. It is the best over-approximation to Eqn. (1) that can be expressed as an affine relation. In contrast, a more conventional approach to creating an abstract transformer for φ_I is to use operator-by-operator reinterpretation of Eqn. (1). The resulting abstract transformer would be $(\texttt{eax}' = \texttt{eax})$, which loses all information about `ebx`. Such loss in precision is exacerbated when considering larger loop-free blocks of instructions.

Motivation. Reps, Sagiv, and Yorsh (RSY) [24] presented a framework for computing $\widehat{\alpha}$ that applies to any logic and abstract domain that satisfies certain conditions. King and Søndergaard [16] gave a specific $\widehat{\alpha}$ algorithm for an abstract domain of Boolean affine relations. Elder et al. [11] extended their algorithm to affine relations in arithmetic modulo 2^w—i.e., for some bit-width w of bounded integers. (When the generalized algorithm is applied to φ_I from Eqn. (1), it finds the $\widehat{\alpha}(\varphi_I)$ formula indicated above.) Because the generalized algorithm is similar to the Boolean one, we refer to it as KS. We use RSY[AR] to denote the RSY framework instantiated for the abstract domain of affine relations modulo 2^w.

The RSY[AR] and KS algorithms resemble one another in that they both find $\widehat{\alpha}(\varphi)$ via successive approximation from "below". However, *the two algo-*

rithms are not the same. As discussed in §2, although both the RSY[AR] and KS algorithms issue queries to a decision procedure, compared to the RSY[AR] algorithm, the KS algorithm issues *comparatively inexpensive* decision-procedure queries. Moreover, the differences in the two algorithms cause an order-of-magnitude difference in performance: in our experiments, *KS is approximately ten times faster* than RSY[AR].

These issues motivated us to (i) investigate the fundamental principles underlying the difference between the RSY[AR] and KS algorithms, and (ii) seek a *framework* into which the KS algorithm could be placed, so that its advantages could be transferred to other domains. A third motivating issue was that neither the RSY framework nor the KS algorithm are resilient to timeouts. Because the algorithms maintain only under-approximations of the desired answer, if the successive-approximation process takes too much time and needs to be stopped, they must return $\top$ to be sound. We desired an algorithm that could return a nontrivial (non-$\top$) value in case of a timeout.

The outcome of our work is a new *framework* for symbolic abstraction that

– is applicable to any abstract domain that satisfies certain conditions (similar to the RSY algorithm)
– uses a successive-approximation algorithm that is *parsimonious* in its use of the decision procedure (similar to the KS algorithm)
– is *bilateral*; that is, it maintains both an under-approximation and a (nontrivial) over-approximation of the desired answer, and hence is resilient to timeouts: the procedure can return the over-approximation if it is stopped at any point (unlike the RSY and KS algorithms).

The key concept used in generalizing the KS algorithm is an operation that we call `AbstractConsequence` (Defn. 1, §3). We show that many abstract domains have an `AbstractConsequence` operation that enables the kind of inexpensive decision-procedure queries that we see in the KS algorithm (Thm. 2, §3).

Our experiments show that the bilateral algorithm for the AR domain improves precision at up to 15% of a program's control points (i.e., the beginning of a basic block that ends with a branch), and on average is more precise for 3.1% of the control points (computed as the arithmetic mean).

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

– We show how the KS algorithm can be modified into a *bilateral algorithm* that maintains sound under- and over-approximations of the answer (§2).
– We present a framework for symbolic abstraction based on a bilateral algorithm for computing $\widehat{\alpha}$ (§3).
– We give several instantiations of the framework (§3 and §4).
– We compare the performance of various algorithms (§2 and §5).

§6 discusses related work. Some additional technical material is given in Apps. A, B, and C.

2 Towards a Bilateral Algorithm

In this section, we compare the performance of the RSY[AR] and KS algorithms, and motivate the need for designing a "KS-like" framework for symbolic ab-

Algorithm 1: $\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}\langle\mathcal{L},\mathcal{A}\rangle(\varphi)$

```
1  $lower \leftarrow \bot$
2
3  while true do
4
5    $S \leftarrow$ Model($\varphi \wedge \neg\widehat{\gamma}(lower)$)
6    if $S$ is TimeOut then
7      return $\top$
8    else if $S$ is None then
9      break                        // $\varphi \Rightarrow \widehat{\gamma}(lower)$
10   else                           // $S \not\models \widehat{\gamma}(lower)$
11     $lower \leftarrow lower \sqcup \beta(S)$
12 $ans \leftarrow lower$
13 return $ans$
```

Algorithm 2: $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}(\varphi)$

```
1  $lower \leftarrow \bot$
2  $i \leftarrow 1$
3  while $i \le$ rows($lower$) do
4    $p \leftarrow$ Row($lower, -i$)    // $p \sqsupseteq lower$
5    $S \leftarrow$ Model($\varphi \wedge \neg\widehat{\gamma}(p)$)
6    if $S$ is TimeOut then
7      return $\top$
8    else if $S$ is None then
9      $i \leftarrow i + 1$                // $\varphi \Rightarrow \widehat{\gamma}(p)$
10   else                           // $S \not\models \widehat{\gamma}(p)$
11     $lower \leftarrow lower \sqcup \beta(S)$
12 $ans \leftarrow lower$
13 return $ans$
```

straction. We then show how the KS algorithm can be modified to be a bilateral algorithm, which provides insight on the generalized framework presented in §3.

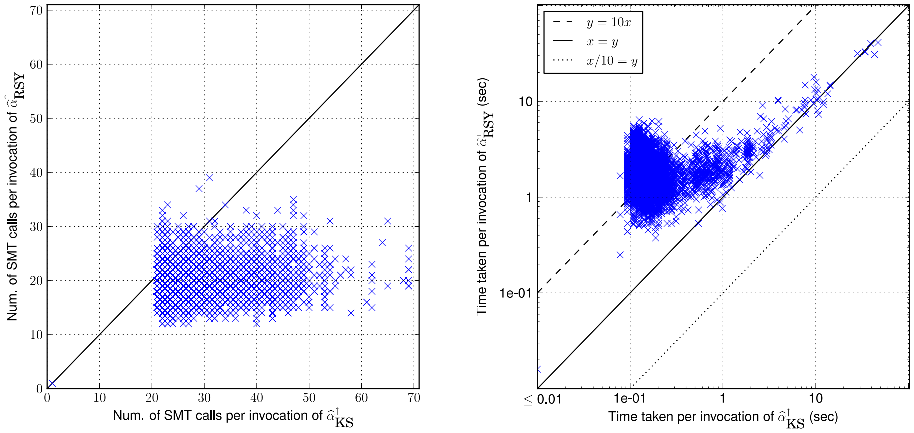
Alg. 1 shows the general RSY algorithm ($\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}\langle\mathcal{L},\mathcal{A}\rangle$) [24], which is parameterized on logic $\mathcal{L}$ and abstract domain $\mathcal{A}$. Alg. 2 shows the KS algorithm ($\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$) [16, 11], which is specific to the QFBV logic and the affine-relations (AR) domain. The following notation is used in the algorithms:

- The operation of *symbolic concretization* (line 5 of Algs. 1 and 2), denoted by $\widehat{\gamma}$, maps an abstract value $a \in \mathcal{A}$ to a formula $\widehat{\gamma}(a) \in \mathcal{L}$ such that a and $\widehat{\gamma}(a)$ represent the same set of concrete states (i.e., $\gamma(a) = [\![\widehat{\gamma}(a)]\!]$).
- Given a formula $\psi \in \mathcal{L}$, Model(ψ) returns (i) a satisfying model S if a decision procedure was able to determine that ψ is satisfiable in a given time limit, (ii) None if a decision procedure was able to determine that ψ is unsatisfiable in a given time limit, and (iii) TimeOut otherwise.
- The *representation function* β (line 11 of Algs. 1 and 2) maps a singleton concrete state $S \in \mathcal{C}$ to the least value in $\mathcal{A}$ that over-approximates $\{S\}$.

An abstract value in the AR domain is a conjunction of affine equalities, which can be represented in a normal form as a matrix in which each row expresses a non-redundant affine equality [11]. (Rows are 0-indexed.) Given a matrix m, rows(m) returns the number of rows of m (line 3 in Alg. 2), and Row($m, -i$), for $1 \le i \le$ rows(m), returns row (rows(m) $- i$) of m (line 4 in Alg. 2).

Both algorithms have a similar overall structure. Both are successive approximation algorithms: they compute a sequence of successively "larger" approximations to the set of states described by φ. Both maintain an under-approximation of the final answer in the variable *lower*, which is initialized to $\bot$ on line 1. Both call a decision procedure (line 5), and having found a model S that satisfies the query, the under-approximation is updated by performing a join (line 11).

The differences between Algs. 1 and 2 are highlighted in gray. The key difference is the nature of the decision-procedure query on line 5. $\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}$ uses *all* of *lower* to construct the query, while $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$ uses only a single row from *lower*



(a) (b)

Fig. 1. (a) Scatter plot showing of the number of decision-procedure queries during each pair of invocations of $\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}$ and $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$, when neither invocation had a decision-procedure timeout. (b) Log-log scatter plot showing the times taken by each pair of invocations of $\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}$ and $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$, when neither invocation had a decision-procedure timeout.

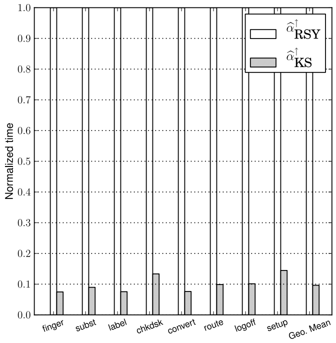


Fig. 2. Total time taken by all invocations of $\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}$ compared to that taken by $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$ for each of the benchmark executables. The running time is normalized to the corresponding time taken by $\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}$; lower numbers are better.

(line 4)—i.e., just a *single affine equality*, which has two consequences. On the one hand, $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$ could issue a larger number of queries compared to $\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}$. Suppose that the value of *lower* has converged to the final answer via a sequence of joins performed by the algorithm. To discover that convergence has occurred, $\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}$ has to issue just a single decision-procedure query, whereas $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$ has to confirm it by issuing $\texttt{rows}(\mathit{lower}) - i$ number of queries, proceeding row-by-row. On the other hand, each individual query issued by $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$ is simpler than the ones issued by $\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}$. Thus, *a priori*, it is not clear which algorithm will perform better in practice.

Algorithm 3: $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}(\varphi)$

```
1
2  $lower \leftarrow \bot$
3  $i \leftarrow 1$
4  while $i \leq$ rows($lower$) do
5    $p \leftarrow$ Row($lower, -i$)
   // $p \sqsupseteq$ lower
6    $S \leftarrow$ Model($\varphi \wedge \neg\widehat{\gamma}(p)$)
7    if $S$ is TimeOut then
8      return $\top$
9    else if $S$ is None then
                                        // $\varphi \Rightarrow \widehat{\gamma}(p)$
10     $i \leftarrow i + 1$
11   else                               // $S \not\models \widehat{\gamma}(p)$
12     $lower \leftarrow lower \sqcup \beta(S)$
13 $ans \leftarrow lower$
14 return $ans$
```

Algorithm 4: $\widehat{\alpha}^{\updownarrow}_{\mathrm{KS+}}(\varphi)$

```
1  $upper \leftarrow \top$
2  $lower \leftarrow \bot$
3  $i \leftarrow 1$
4  while $i \leq$ rows($lower$) do
5    $p \leftarrow$ Row($lower, -i$)
   // $p \sqsupseteq$ lower, $p \not\sqsupseteq$ upper
6    $S \leftarrow$ Model($\varphi \wedge \neg\widehat{\gamma}(p)$)
7    if $S$ is TimeOut then
8      return $upper$
9    else if $S$ is None then
10     $upper \leftarrow upper \sqcap p$        // $\varphi \Rightarrow \widehat{\gamma}(p)$
       $i \leftarrow i + 1$
11   else                               // $S \not\models \widehat{\gamma}(p)$
12     $lower \leftarrow lower \sqcup \beta(S)$
13 $ans \leftarrow lower$
14 return $ans$
```

We compared the time for $\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}$ (instantiated for QFBV and the AR domain) and $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$ to compute basic-block transformers for a set of x86 executables. There was no overall timeout imposed on the invocation of the procedures, but each invocation of the decision procedure (line 5 in Algs. 1 and 2) had a timeout of 3 seconds. (Details of the experimental setup are described in §5.) Fig. 1(a) shows a scatter-plot of the *number of decision-procedure calls* in each invocation of $\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}$ versus the corresponding invocation of $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$, when neither of the procedures had a decision-procedure timeout. We see that $\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}$ issues fewer decision-procedure queries: on average (computed as an arithmetic mean), $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$ invokes 42% more calls to the decision procedure. Fig. 1(b) shows a log-log scatter-plot of the *total time* taken by each invocation of $\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}$ versus the time taken by $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$. As we can see, $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$ is much faster than $\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}$. Fig. 2 shows the total time taken by all invocations of $\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}$ compared to that taken by $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$ for each of the benchmark executables. The running time is normalized to the corresponding time taken by $\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}$; lower numbers are better. Overall, $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$ is about ten times faster than $\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}$.

The order-of-magnitude speedup can be attributed to the fact that each decision-procedure query is less expensive in $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$ compared to $\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}$. At line 4 in $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$, p is a single constraint; consequently, the decision-procedure query contains the *single* conjunct $\neg\widehat{\gamma}(p)$ (line 5). In contrast, at line 5 in $\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}$, *lower* is a *conjunction* of constraints, and consequently the decision-procedure query contains $\neg\widehat{\gamma}(lower)$, which is a *disjunction* of constraints.

Neither $\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}$ nor $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$ is resilient to timeouts. A decision-procedure query—or the cumulative time for $\widehat{\alpha}^{\uparrow}$—might take too long, in which case the only safe answer that can be returned is $\top$ (line 6 in Algs. 1 and 2). To remedy this

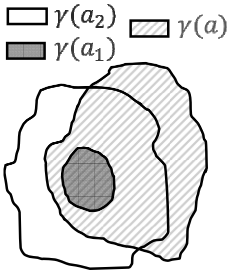


Fig. 3. Abstract Consequence: For all $a_1, a_2 \in \mathcal{A}$ such that $\gamma(a_1) \subsetneq \gamma(a_2)$, $a = \mathtt{AbstractConsequence}(a_1, a_2)$ implies $\gamma(a_1) \subseteq \gamma(a)$ and $\gamma(a) \not\supseteq \gamma(a_2)$.

Algorithm 5: $\widetilde{\alpha}^{\updownarrow}\langle\mathcal{L},\mathcal{A}\rangle(\varphi)$

```
$upper \leftarrow \top$
$lower \leftarrow \bot$
while $lower \neq upper$ do
// $lower \sqsubsetneq upper$
  $p \leftarrow \mathtt{AbstractConsequence}(lower, upper)$
// $p \sqsupseteq lower, p \not\sqsupseteq upper$
  $S \leftarrow \mathtt{Model}(\varphi \wedge \neg\widehat{\gamma}(p))$
  if $S$ is TimeOut then
    return $upper$
  else if $S$ is None then          // $\varphi \Rightarrow \widehat{\gamma}(p)$
    $upper \leftarrow upper \sqcap p$
  else                              // $S \not\models \widehat{\gamma}(p)$
    $lower \leftarrow lower \sqcup \beta(S)$
$ans \leftarrow upper$
return $ans$
```

situation, we show how $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$ can be modified to maintain a non-trivial over-approximation of the desired answer. Alg. 4 is such a *bilateral* algorithm that maintains both an under-approximation and over-approximation of $\widehat{\alpha}(\varphi)$. The original $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$ is shown in Alg. 3 for comparison; the differences in the algorithms are highlighted in gray. (Note that line numbers are different in Algs. 2 and 3.)

The $\widetilde{\alpha}^{\updownarrow}_{\mathrm{KS+}}$ algorithm (Alg. 4) initializes the over-approximation (*upper*) to $\top$ on line 1. At any stage in the algorithm $\varphi \Rightarrow \widehat{\gamma}(upper)$. On line 10, it is sound to update *upper* by performing a meet with p because $\varphi \Rightarrow \widehat{\gamma}(p)$. Progress is guaranteed because $p \not\sqsupseteq upper$. In case of a decision-procedure timeout (line 7), Alg. 4 returns *upper* as the answer (line 8). We use "$\widetilde{\ }$" to emphasize the fact that $\widetilde{\alpha}^{\updownarrow}_{\mathrm{KS+}}(\varphi)$ can return an over-approximation of $\widehat{\alpha}(\varphi)$ in case of a timeout. However, if the loop exits without a timeout, then $\widetilde{\alpha}^{\updownarrow}_{\mathrm{KS+}}(\varphi)$ returns $\widehat{\alpha}(\varphi)$.

3 A Parametric Bilateral Algorithm

Like the original KS algorithm, $\widetilde{\alpha}^{\updownarrow}_{\mathrm{KS+}}$ applies only to the AR domain. The results presented in §2 provide motivation to generalize $\widetilde{\alpha}^{\updownarrow}_{\mathrm{KS+}}$ so that we can take advantage of its benefits with domains other than AR. In this section, we present the bilateral framework we developed, which applies to any abstract domain that satisfies the interface defined below.

We first introduce the *abstract-consequence* operation, which is the key operation in our generalized algorithm:

Definition 1. *An operation* $\mathtt{AbstractConsequence}(\cdot,\cdot)$ *is an* ***acceptable abstract-consequence operation*** *iff for all* $a_1, a_2 \in \mathcal{A}$ *such that* $a_1 \sqsubsetneq a_2$, $a = \mathtt{AbstractConsequence}(a_1, a_2)$ *implies* $a_1 \sqsubseteq a$ *and* $a \not\sqsupseteq a_2$. □

Fig. 3 illustrates Defn. 1 graphically, using the concretizations of a_1, a_2, and a.

Algorithm 6: `AbstractConsequence`(a_1, a_2) for conjunctive domains

```
if $a_1 = \bot$ then return $\bot$
Let $\Psi \subseteq \Phi$ be the set of formulas such that $\widehat{\gamma}(a_1) = \bigwedge \Psi$
foreach $\psi \in \Psi$ do
    $a \leftarrow \mu\widehat{\alpha}(\psi)$
    if $a \not\sqsupseteq a_2$ then return $a$
```

Alg. 5 presents the parametric bilateral algorithm $\widetilde{\alpha}^{\updownarrow}\langle\mathcal{L}, \mathcal{A}\rangle(\varphi)$, which performs symbolic abstraction of $\varphi \in \mathcal{L}$ for abstract domain $\mathcal{A}$. The differences between Alg. 5 and Alg. 4 are highlighted in gray.

The assumptions placed on the logic and the abstract domain are as follows:

1. There is a Galois connection $\mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$ between $\mathcal{A}$ and concrete domain $\mathcal{C}$.
2. Given $a_1, a_2 \in \mathcal{A}$, there are algorithms to evaluate $a_1 \sqcup a_2$ and $a_1 \sqcap a_2$, and to check $a_1 = a_2$.
3. There is a symbolic-concretization operation $\widehat{\gamma}$ that maps an abstract value $a \in \mathcal{A}$ to a formula $\widehat{\gamma}(a)$ in $\mathcal{L}$.
4. There is a decision procedure for the logic $\mathcal{L}$ that is also capable of returning a model satisfying a formula in $\mathcal{L}$.
5. The logic $\mathcal{L}$ is closed under conjunction and negation.
6. There is an acceptable abstract-consequence operation for $\mathcal{A}$ (Defn. 1).

The abstract value returned by `AbstractConsequence` (line 4 of Alg. 5) is used to generate the decision-procedure query (line 5).

Theorem 1. [Correctness of Alg. 5] *Suppose that $\mathcal{L}$ and $\mathcal{A}$ satisfy requirements 1–6, and $\varphi \in \mathcal{L}$. Let $a \in \mathcal{A}$ be the value returned by $\widetilde{\alpha}^{\updownarrow}\langle\mathcal{L}, \mathcal{A}\rangle(\varphi)$. Then*

1. *a over-approximates $\widehat{\alpha}(\varphi)$; i.e., $\widehat{\alpha}(\varphi) \sqsubseteq a$.*
2. *If $\mathcal{A}$ has neither infinite ascending nor infinite descending chains and $\widetilde{\alpha}^{\updownarrow}\langle\mathcal{L}, \mathcal{A}\rangle(\varphi)$ returns with no timeout, then $a = \widehat{\alpha}(\varphi)$.*

Proof. See App. B. □

Defn. 1 allows `AbstractConsequence`(a_1, a_2) to return any $a \in \mathcal{A}$ as long as a satisfies $a_1 \sqsubseteq a$ and $a \not\sqsupseteq a_2$. Thus, for a given abstract domain $\mathcal{A}$ there could be multiple implementations of the `AbstractConsequence` operation. In particular, `AbstractConsequence`(a_1, a_2) can return a_1, because $a_1 \sqsubseteq a_1$ and $a_1 \not\sqsupseteq a_2$. If this particular implementation of `AbstractConsequence` is used, then Alg. 5 reduces to the RSY algorithm (Alg. 1). However, as illustrated in §2, the decision-procedure queries issued by the RSY algorithm can be very expensive.

Conjunctive domains. We now define a class of *conjunctive domains*, for which `AbstractConsequence` can be implemented by the method presented as Alg. 6. The benefit of Alg. 6 is that it causes Alg. 5 to issue the kind of inexpensive queries that we see in $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$. Let Φ be a given set of formulas expressed in $\mathcal{L}$. A *conjunctive domain* over Φ is an abstract domain $\mathcal{A}$ such that:

– For any $a \in \mathcal{A}$, there exists a finite subset $\Psi \subseteq \Phi$ such that $\widehat{\gamma}(a) = \bigwedge \Psi$.

- For any finite $\Psi \subseteq \Phi$, there exists an $a \in \mathcal{A}$ such that $\gamma(a) = [\![\bigwedge \Psi]\!]$.
- There is an algorithm $\mu\widehat{\alpha}(\varphi)$ ("micro-$\widehat{\alpha}$") that, for each singleton formula $\varphi \in \Phi$, returns $a_\varphi \in \mathcal{A}$ such that $\widehat{\alpha}(\varphi) = a_\varphi$.
- There is an algorithm that, for all $a_1, a_2 \in \mathcal{A}$, checks $a_1 \sqsubseteq a_2$.[3]

Many common domains are conjunctive domains. For example, using v, v_i for program variables and c, c_i for constants:

Domain	Φ
Interval domain	inequalities of the form $c_1 \leq v$ and $v \leq c_2$
Octagon domain [20]	inequalities of the form $\pm v_1 \pm v_2 \leq c$
Polyhedral domain [7]	linear inequalities over reals or rationals
KS domain [16, 11]	linear equalities over integers mod 2^w

Theorem 2. *When $\mathcal{A}$ is a conjunctive domain over Φ, Alg. 6 is an acceptable abstract-consequence operation.*

Proof. See App. B. □

If there are also algorithms for join and meet in $\mathcal{A}$, and a decision procedure for the logic $\mathcal{L}$ that supplies models for satisfiable formulas, then $\mathcal{A}$ satisfies the bilateral framework, and therefore supports the $\widetilde{\alpha}^{\updownarrow}$ algorithm.

Discussion. We can weaken part 2 of Thm. 1 to allow $\mathcal{A}$ to have infinite descending chains by modifying Alg. 5 slightly. The modified algorithm has to ensure that it does not get trapped updating *upper* along an infinite descending chain, and that it exits when *lower* has converged to $\widehat{\alpha}(\varphi)$. Suppose that, for some fixed N, N consecutive iterations of the loop on lines 3–11 update *upper* (line 9) *without updating lower* (line 11). If this situation occurs, in the next iteration the algorithm can set p to *lower* so that the decision-procedure query at line 5 becomes $\texttt{Model}(\varphi \wedge \neg\widehat{\gamma}(\mathit{lower}))$—i.e., we force the algorithm to perform the basic iteration-step from the RSY algorithm. In this way, we force *lower* to be updated at least once every N iterations. Moreover, if on such an RSY-step the model S returned from the decision procedure is None, then we know that *lower* has converged to $\widehat{\alpha}(\varphi)$ and the algorithm can return. A version of Alg. 5 that implements this strategy is presented as Alg. 7 (see App. C).

As presented, Alg. 5 exits and returns the value of *upper* the first time the decision procedure times out. We can improve the precision of Alg. 5 by not exiting after the first timeout, and instead trying other abstract consequences. The algorithm will exit and return *upper* only if it cannot find an abstract consequence for which the decision-procedure terminates within the time bound. For conjunctive domains, Alg. 5 can be modified to enumerate all conjuncts of *lower* that are abstract conse-

[3] Note that $a_1 \sqcup a_2 = a_2$ iff $a_1 \sqsubseteq a_2$ iff $a_1 \sqcap a_2 = a_1$, so by Assumption 2 of the bilateral framework, a comparison test is always available in a conjunctive domain that satisfies the requirements of the bilateral framework.

quences; to implement this strategy, lines 4–7 of Alg. 5 are replaced with

```
$progress \leftarrow$ false                                   // Initialize $progress$
foreach $p$ such that $p =$ AbstractConsequence($lower, upper$) do
  $S \leftarrow$ Model($\varphi \wedge \neg\widehat{\gamma}(p)$)
  if $S$ is not TimeOut then
    $progress \leftarrow$ true                                // Can make progress
    break
if $\neg progress$ then return $upper$                         // Could not make progress
```

Henceforth, when we refer to $\widetilde{\alpha}^{\updownarrow}$, we mean Alg. 5 with the above two changes.

Relationship of `AbstractConsequence` to interpolation. To avoid the potential for confusion, we now discuss how the notion of abstract consequence differs from the well-known concept of *interpolation* [8]:

> A logic $\mathcal{L}$ *supports interpolation* if for all $\varphi_1, \varphi_2 \in \mathcal{L}$ such that $\varphi_1 \Rightarrow \varphi_2$, there exists a formula I such that (i) $\varphi_1 \Rightarrow I$, (ii) $I \Rightarrow \varphi_2$, and (iii) I uses only symbols in the shared vocabulary of φ_1 and φ_2.

Although condition (i) is part of Defn. 1, the restrictions imposed by conditions (ii) and (iii) are not part of Defn. 1. To highlight the differences, we restate Defn. 1 in terms of formulas.

> An operation `AbstractConsequence`$(\cdot,\cdot)$ is an acceptable abstract-consequence operation iff for all $a_1, a_2 \in \mathcal{A}$ such that $\widehat{\gamma}(a_1) \Rightarrow \widehat{\gamma}(a_2)$ and $\widehat{\gamma}(a_1) \not\Leftarrow \widehat{\gamma}(a_2)$, $a =$ `AbstractConsequence`(a_1, a_2) implies $\widehat{\gamma}(a_1) \Rightarrow \widehat{\gamma}(a)$ and $\widehat{\gamma}(a) \not\Leftarrow \widehat{\gamma}(a_2)$.

From an operational standpoint, condition (iii) in the definition of interpolation serves as a heuristic that generally allows interpolants to be expressed as small formulas. In the context of $\widetilde{\alpha}^{\updownarrow}$, we are interested in obtaining small formulas to use in the decision-procedure query (line 5 of Alg. 5). Thus, given $a_1, a_2 \in \mathcal{A}$, it might appear plausible to use an interpolant I of $\widehat{\gamma}(a_1)$ and $\widehat{\gamma}(a_2)$ in $\widetilde{\alpha}^{\updownarrow}$ instead of the abstract consequence of a_1 and a_2. However, there are a few problems with such an approach:

- There is no guarantee that I will indeed be simple; for instance, if the vocabulary of $\widehat{\gamma}(a_1)$ is a subset of the vocabulary of $\widehat{\gamma}(a_2)$, then I could be $\widehat{\gamma}(a_1)$ itself, in which case Alg. 5 performs the more expensive RSY iteration step.
- Converting the formula I into an abstract value $p \in \mathcal{A}$ for use in line 9 of Alg. 5 itself requires performing $\widehat{\alpha}$ on I.

As discussed above, many domains are conjunctive domains, and for conjunctive domains is it always possible to find a *single conjunct* that is an abstract consequence (see Thm. 2). Moreover, such a conjunct is not necessarily an interpolant.

4 Instantiations

In this section, we describe instantiations of the bilateral framework for several abstract domains.

4.1 Herbrand-Equalities Domain

Herbrand equalities are used in analyses for partial redundancy elimination, loop-invariant code motion [28], and strength reduction [29]. In these analyses, arithmetic operations (e.g., + and *) are treated as term constructors. Two program variables are known to hold equal values if the analyzer determines that the variables hold equal terms. Herbrand equalities can also be used to analyze programs whose types are user-defined algebraic data-types.

Basic definitions. Let $\mathcal{F}$ be a set of function symbols. The function $\mathit{arity}\colon \mathcal{F} \to \mathbb{N}$ yields the number of parameters of each function symbol. *Terms* over $\mathcal{F}$ are defined in the usual way; each function symbol f always requires $\mathit{arity}(f)$ parameters. Let $\mathcal{T}(\mathcal{F}, X)$ denote the set of finite terms generated by $\mathcal{F}$ and variable set X. The *Herbrand universe* of $\mathcal{F}$ is $\mathcal{T}(\mathcal{F}, \emptyset)$, the set of *ground terms* over $\mathcal{F}$.

A *Herbrand state* is a mapping from program variables $\mathcal{V}$ to ground terms (i.e., a function in $\mathcal{V} \to \mathcal{T}(\mathcal{F}, \emptyset)$). The concrete domain consists of all sets of Herbrand states: $\mathcal{C} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{V} \to \mathcal{T}(\mathcal{F}, \emptyset))$. We can apply a Herbrand state σ to a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ as follows:

$$\sigma[t] \stackrel{\text{def}}{=} \begin{cases} \sigma(t) & \text{if } t \in \mathcal{V} \\ f(\sigma[t_1], \ldots, \sigma[t_k]) & \text{if } t = f(t_1, \ldots, t_k) \end{cases}$$

The Herbrand-equalities domain. Sets of Herbrand states can be abstracted in several ways. One way is to use conjunctions of equations among terms (whence the name "Herbrand-equalities domain"). Such systems of equations can be represented using Equivalence DAGs [28]. A different, but equivalent, approach is to use a representation based on *idempotent substitutions*: $\mathcal{A} = (\mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V}))_\perp$. Idempotence means that for each $\sigma \neq \perp$ and $v \in \mathcal{V}$, $\sigma[\sigma(v)] = \sigma(v)$. The meaning of an idempotent substitution $\sigma \in \mathcal{A}$ is given by its concretization, $\gamma\colon \mathcal{A} \to \mathcal{C}$, where $\gamma(\perp) = \emptyset$, and otherwise

$$\gamma(\sigma) = \{\rho\colon \mathcal{V} \to \mathcal{T}(\mathcal{F}, \emptyset) \,|\, \forall v \in \mathcal{V}\colon \rho(v) = \rho[\sigma(v)]\}\,. \tag{2}$$

We now show that the Herbrand-equalities domain satisfies the requirements of the bilateral framework. We will assume that the logical language $\mathcal{L}$ has all the function symbols and constant symbols from $\mathcal{F}$, equality, and a constant symbol for each element from $\mathcal{V}$. (In a minor abuse of notation, the set of such constant symbols will also be denoted by $\mathcal{V}$.) The logic's universe is the Herbrand universe of $\mathcal{F}$ (i.e., $\mathcal{T}(\mathcal{F}, \emptyset)$). An interpretation maps the constants in $\mathcal{V}$ to terms in $\mathcal{T}(\mathcal{F}, \emptyset)$. To be able to express $\widehat{\gamma}(p)$ and $\neg\widehat{\gamma}(p)$ (see item 3 below), we assume that $\mathcal{L}$ contains at least the following productions:

$$\begin{aligned} F &::= F \wedge F \mid \neg F \mid v = T \text{ for } v \in \mathcal{V} \mid \ \text{false} \\ T &::= v \in \mathcal{V} \mid f(T_1, \ldots, T_k) \text{ when } \mathit{arity}(f) = k \end{aligned} \tag{3}$$

1. There is a Galois connection $\mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$:
   - The ordering on $\mathcal{C}$ is the subset relation on sets of Herbrand states.
   - $\gamma(\sigma)$ is given in Eqn. (2).
   - $\alpha(S) = \bigsqcap \{a \,|\, \gamma(a) \supseteq S\}$.

   - For $a, b \in \mathcal{A}$, $a \sqsubseteq b$ iff $\gamma(a) \subseteq \gamma(b)$.
2. Join is anti-unification of substitutions, meet is unification of substitutions, and equality checking is described by Lassez et al. [19].
3. $\widehat{\gamma}$: $\widehat{\gamma}(\bot) = \text{false}$; otherwise, $\widehat{\gamma}(\sigma)$ is $\bigwedge_{v \in \mathcal{V}} v = \sigma(v)$.
4. A decision procedure for $\mathcal{L}$, with models, is given by Lassez et al. [19]. In practice, one can obtain a decision procedure for $\mathcal{L}$ formulas using the built-in datatype mechanism of, e.g., Z3 [9] or Yices [10], and obtain the necessary decision procedure using an existing SMT solver.
5. $\mathcal{L}$ is closed under conjunction and negation.
6. `AbstractConsequence`: The domain is a conjunctive domain, as can be seen from the definition of $\widehat{\gamma}$.

Thm. 1 ensures that Alg. 5 returns $\widehat{\alpha}(\varphi)$ when abstract domain $\mathcal{A}$ has neither infinite ascending nor infinite descending chains. The Herbrand-equalities domain has no infinite ascending chains [19, Lem. 3.15]. The domain described here also has no infinite descending chains, essentially because every right-hand term in every Herbrand state has no variables but those in $\mathcal{V}$.[4]

A pair of worked examples of $\widetilde{\alpha}^{\updownarrow}$ (Alg. 5) for the Herbrand-equalities domain is given in App. A.

4.2 Polyhedral Domain

An element of the polyhedral domain [7] is a convex polyhedron, bounded by hyperplanes. It may be unbounded in some directions. The symbolic concretization of a polyhedron is a conjunction of linear inequalities. The polyhedral domain is a conjunctive domain:

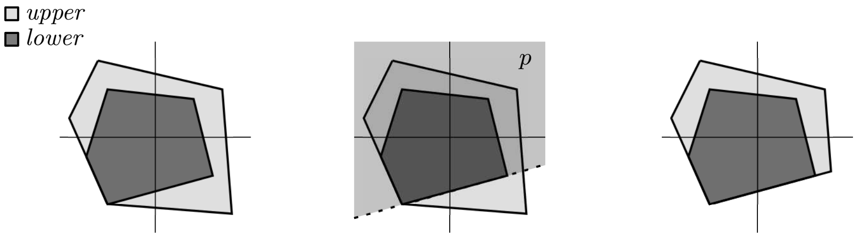
- Each polyhedron can be expressed as some conjunction of linear inequalities ("half-spaces") from the set $\mathcal{F} = \left\{ \sum_{v \in \mathcal{V}} c_v v \geq c \,\middle|\, c, c_v \text{ are constants} \right\}$.
- Every finite conjunction of facts from $\mathcal{F}$ can be represented as a polyhedron.
- $\mu\widehat{\alpha}$: Each formula in $\mathcal{F}$ corresponds to a simple, one-constraint polyhedron.
- There is an algorithm for comparing two polyhedra [7].

In addition, there are algorithms for join, meet, and checking equality.

The logic QF_LRA (quantifier-free linear real arithmetic) supported by SMT solvers provides a decision procedure for the fragment of logic that is required to express negation, conjunction, and $\widehat{\gamma}$ of a polyhedron. Consequently, the polyhedral domain satisfies the bilateral framework, and therefore supports the $\widetilde{\alpha}^{\updownarrow}$ algorithm. The polyhedral domain has both infinite ascending chains and infinite descending chains, and hence Alg. 5 is only guaranteed to compute an over-approximation of $\widehat{\alpha}(\varphi)$.

Because the polyhedral domain is a conjunctive domain, if *lower* $\sqsubsetneq$ *upper*, then some single constraint p of *lower* satisfies $p \not\sqsupseteq$ *upper*. For instance, if *lower*

[4] One can instead define a domain of Herbrand equalities in which fresh variables may occur in the terms of an idempotent substitution's range. Everything said in this section remains true of this alternative domain, except that it has infinite descending chains, and so Alg. 5 is only guaranteed to return an over-approximation of $\widehat{\alpha}(\varphi)$. However, as discussed in §3, Alg. 7, presented in App. C, is a version of Alg. 5 that converges to $\widehat{\alpha}(\varphi)$, even when the abstract domain has infinite descending chains.



(a) (b) (c)

Fig. 4. Abstract consequence on polyhedra. (a) Two polyhedra: *lower* $\sqsubseteq$ *upper*. (b) $p = \texttt{AbstractConsequence}(\mathit{lower}, \mathit{upper})$. (c) Result of $\mathit{upper} \leftarrow \mathit{upper} \sqcap p$.

and *upper* are the polyhedra shown in Fig. 4(a), then the region p above the dotted line in Fig. 4(b) is an acceptable abstract consequence. Fig. 4(c) shows the result after $\mathit{upper} \leftarrow \mathit{upper} \sqcap p$ is performed at line 9 of Alg. 5.

5 Experiments

In this section, we compare two algorithms for performing symbolic abstraction for the affine-relations (AR) domain [16, 11]:

- the $\widehat{\alpha}^{\uparrow}_{\text{KS}}$ procedure of Alg. 2 [11].
- the $\widetilde{\alpha}^{\updownarrow}\langle\text{AR}\rangle$ procedure that is the instantiation of Alg. 5 for the affine-relations (AR) domain and QFBV logic.

Although the bilateral algorithm $\widetilde{\alpha}^{\updownarrow}\langle\text{AR}\rangle$ benefits from being resilient to timeouts, it maintains *both* an over-approximation and an under-approximation. Thus, the experiments were designed to understand the trade-off between performance and precision. In particular, the experiments were designed to answer the following questions:

1. How does the speed of $\widetilde{\alpha}^{\updownarrow}\langle\text{AR}\rangle$ compare with that of $\widehat{\alpha}^{\uparrow}_{\text{KS}}$?
2. How does the precision of $\widetilde{\alpha}^{\updownarrow}\langle\text{AR}\rangle$ compare with that of $\widehat{\alpha}^{\uparrow}_{\text{KS}}$?

To address these questions, we performed affine-relations analysis (ARA) on x86 machine code, computing affine relations over the x86 registers. Our experiments were run on a single core of a quad-core 3.0 GHz Xeon computer running 64-bit Windows XP (SP2), configured so that a user process has 4GB of memory. We analyzed a corpus of Windows utilities using the WALi [15] system for weighted pushdown systems (WPDSs). For the $\widehat{\alpha}^{\uparrow}_{\text{KS}}$-based ($\widetilde{\alpha}^{\updownarrow}\langle\text{AR}\rangle$-based) analysis we used a weight domain of $\widehat{\alpha}^{\uparrow}$-generated ($\widetilde{\alpha}^{\updownarrow}\langle\text{AR}\rangle$-generated) ARA transformers. The weight on each WPDS rule encodes the ARA transformer for a basic block B of the program, including a jump or branch to a successor block. A formula φ_B is created that captures the concrete semantics of B, and then the ARA weight for B is obtained by performing $\widehat{\alpha}(\varphi_B)$. We used EWPDS merge functions [18] to preserve caller-save and callee-save registers across call sites. The post* query used the FWPDS algorithm [17].

	Measures of size				Performance (x86)			Better
Prog.					$\widehat{\alpha}^{\uparrow}_{\text{KS}}$		$\widetilde{\alpha}^{\updownarrow}\langle\text{AR}\rangle$	$\widetilde{\alpha}^{\updownarrow}\langle\text{AR}\rangle$
name	instrs	procs	BBs	brs	WPDS	t/o	WPDS	precision
finger	532	18	298	48	104.0	4	138.9	**6.3%**
subst	1093	16	609	74	196.7	4	214.6	0%
label	1167	16	573	103	146.1	2	171.6	0%
chkdsk	1468	18	787	119	377.2	16	417.9	0%
convert	1927	38	1013	161	287.1	10	310.5	0%
route	1982	40	931	243	618.4	14	589.9	**2.5%**
logoff	2470	46	1145	306	611.2	16	644.6	**15.0%**
setup	4751	67	1862	589	1499	60	1576	**1.0%**

Fig. 5. WPDS experiments. The columns show the number of instructions (instrs); the number of procedures (procs); the number of basic blocks (BBs); the number of branch instructions (brs); the times, in seconds, for $\widehat{\alpha}^{\uparrow}_{\text{KS}}$ and $\widetilde{\alpha}^{\updownarrow}\langle\text{AR}\rangle$ WPDS construction; the number of invocations of $\widehat{\alpha}^{\uparrow}_{\text{KS}}$ that had a decision procedure timeout (t/o); and the degree of improvement gained by using $\widetilde{\alpha}^{\updownarrow}\langle\text{AR}\rangle$-generated ARA weights rather than $\widehat{\alpha}^{\uparrow}_{\text{KS}}$ weights (measured as the percentage of control points whose inferred one-vocabulary affine relation was strictly more precise under $\widetilde{\alpha}^{\updownarrow}\langle\text{AR}\rangle$-based analysis).

Fig. 5 lists several size parameters of the examples (number of instructions, procedures, basic blocks, and branches).[5] Prior research [11] shows that the calls to $\widehat{\alpha}$ during WPDS construction dominate the total time for ARA. Although the overall time taken by $\widehat{\alpha}$ is not limited by a timeout, we use a 3-second timeout for each invocation of the decision procedure (as in Elder et al. [11]). Column 7 of Fig. 5 lists the number invocations of $\widehat{\alpha}^{\uparrow}_{\text{KS}}$ that had a decision-procedure timeout, and hence returned $\top$.

Columns 6 and 8 of Fig. 5 list the time taken, in seconds, for $\widehat{\alpha}^{\uparrow}_{\text{KS}}$ and $\widetilde{\alpha}^{\updownarrow}\langle\text{AR}\rangle$ WPDS construction. We observe that on average $\widetilde{\alpha}^{\updownarrow}\langle\text{AR}\rangle$ is about 10% slower than $\widehat{\alpha}^{\uparrow}_{\text{KS}}$ (computed as the geometric mean), which answers question 1.

To answer question 2 we compared the precision of the WPDS analysis when using $\widehat{\alpha}^{\uparrow}_{\text{KS}}$ with the precision obtained using $\widetilde{\alpha}^{\updownarrow}\langle\text{AR}\rangle$. In particular, we compare the affine-relation invariants computed by the $\widehat{\alpha}^{\uparrow}_{\text{KS}}$-based and $\widetilde{\alpha}^{\updownarrow}\langle\text{AR}\rangle$-based analyses for each *control point*—i.e., the beginning of a basic block that ends with a branch. The last column of Fig. 5 shows the percentage of control points for which the $\widetilde{\alpha}^{\updownarrow}\langle\text{AR}\rangle$-based analysis computed a strictly more precise affine relation. We see that the $\widetilde{\alpha}^{\updownarrow}\langle\text{AR}\rangle$-based analysis improves precision at up to 15% of control points, and, on average, the $\widetilde{\alpha}^{\updownarrow}\langle\text{AR}\rangle$-based analysis is more precise for 3.1% of the control points (computed as the arithmetic mean), which answers question 2.

6 Related Work

6.1 Related Work on Symbolic Abstraction

Previous work on symbolic abstraction falls into three categories:

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

1. algorithms for specific domains [23, 3, 2, 16, 11]
2. algorithms for parameterized abstract domains [12, 32, 26, 22]
3. abstract-domain frameworks [24, 31].

What distinguishes category 3 from category 2 is that each of the results cited in category 2 applies to a specific *family* of abstract domains, defined by a *parameterized Galois connection* (e.g., with an abstraction function equipped with a readily identifiable parameter for controlling the abstraction). In contrast, the results in category 3 are defined by an *interface*; for any abstract domain that satisfies the requirements of the interface, one has a method for symbolic abstraction. The approach presented in this paper falls into category 3.

Algorithms for specific domains. Regehr and Reid [23] present a method that constructs abstract transformers for machine instructions, for interval and bitwise abstract domains. Their method does not call a SAT solver, but instead uses the physical processor (or a simulator of a processor) as a black box. To compute the abstract post-state for an abstract value a, the approach recursively divides a until an abstract value is obtained whose concretization is a singleton set. The concrete semantics are then used to derive the post-state value. The results of each division are joined as the recursion unwinds to derive the abstract post-state value.

Brauer and King [3] developed a method that works from below to derive abstract transformers for the interval domain. Their method is based on an approach due to Monniaux [22] (see below), but they changed two aspects:

1. They express the concrete semantics with a Boolean formula (via "bit-blasting"), which allows a formula equivalent to $\forall x.\varphi$ to be obtained from φ (in CNF) by removing the x and $\neg x$ literals from all of the clauses of φ.
2. Whereas Monniaux's method performs abstraction and then quantifier elimination, Brauer and King's method performs quantifier elimination on the concrete specification, and then performs abstraction.

The abstract transformer derived from the Boolean formula that results is a guarded update: the guard is expressed as an element of the octagon domain [20]; the update operation is expressed as an element of the abstract domain of rational affine equalities [14]. The abstractions performed to create the guard and the update are optimal for their respective domains. The algorithm they use to create the abstract value for the update operation is essentially the King-Søndergaard algorithm for $\widehat{\alpha}$ [16, Fig. 2], which works from below (see Alg. 2). Brauer and King show that optimal evaluation of such transfer functions requires linear programming. They give an example that demonstrates that an octagon-closure operation on a combination of the guard's octagon and the update's affine equality is sub-optimal.

Barrett and King [2] describe a method for generating range and set abstractions for bit-vectors that are constrained by Boolean formulas. For range analysis, the algorithm separately computes the minimum and maximum value of the range for an n-bit bit-vector using $2n$ calls to a SAT solver, with each SAT query determining a single bit of the output. The result is the best over-approximation of the value that an integer variable can take on (i.e., $\widehat{\alpha}$).

Algorithms for parameterized abstract domains. Graf and Saïdi [12] showed that decision procedures can be used to generate best abstract transformers for predicate-abstraction domains. Other work has investigated more efficient methods to generate approximate transformers that are not best transformers, but approach the precision of best transformers [1, 4].

Yorsh et al. [32] developed a method that works from above to perform $\widetilde{\alpha}(\varphi)$ for the kind of abstract domains used in shape analysis (i.e., "canonical abstraction" of logical structures [25]).

Template Constraint Matrices (TCMs) are a parametrized family of linear-inequality domains for expressing invariants in linear real arithmetic. Sankaranarayanan et al. [26] gave a parametrized meet, join, and set of abstract transformers for all TCM domains. Monniaux [22] gave an algorithm that finds the best transformer in a TCM domain across a straight-line block (assuming that concrete operations consist of piecewise linear functions), and good transformers across more complicated control flow. However, the algorithm uses quantifier elimination, and no polynomial-time elimination algorithm is known for piecewise-linear systems.

Abstract-domain frameworks. As discussed in §3, the bilateral framework reduces to the RSY framework when using a particular (trivial) implementation of `AbstractConsequence`. Unlike the RSY framework, to compute $\widetilde{\alpha}(\varphi)$ the bilateral framework does not impose the requirement that the abstract domain have no infinite ascending chains. As shown in part 1 of Thm. 1, even when there are infinite ascending chains, the bilateral framework can return a *non-trivial* over-approximation of $\widehat{\alpha}(\varphi)$. In contrast, RSY gives no such guarantees. Consequently, compared to the RSY framework, the bilateral framework is applicable to a larger class of abstract domains.

Thakur and Reps [31] recently discovered a new framework for performing symbolic abstraction from "above": $\widetilde{\alpha}^{\downarrow}$. The $\widetilde{\alpha}^{\downarrow}$ framework builds upon the insight that Stålmarck's algorithm for propositional validity checking [27] can be explained using abstract-interpretation terminology [30]. The $\widetilde{\alpha}^{\downarrow}$ framework adapts the same algorithmic components of this generalization to perform symbolic abstraction. Because $\widetilde{\alpha}^{\downarrow}$ maintains an over-approximation of $\widehat{\alpha}$, it is resilient to timeouts.

The $\widetilde{\alpha}^{\downarrow}$ framework is based on much different principles from the RSY and bilateral frameworks. The latter frameworks use an *inductive-learning approach* to learn from examples, while the $\widetilde{\alpha}^{\downarrow}$ framework uses a *deductive approach* by using inference rules to deduce the answer. Thus, they represent two different classes of frameworks, with different requirements for the abstract domain. In contrast to the bilateral framework, which uses a decision procedure as a black box, the $\widetilde{\alpha}^{\downarrow}$ framework adopts (and adapts) some principles from Stålmarck's decision procedure. However, the instantiation of the $\widetilde{\alpha}^{\downarrow}$ framework for affine-relations over integers mod 2^w is not guaranteed to compute $\widehat{\alpha}$—unlike the bilateral framework, which (by part 2 of Thm. 1) *is* guaranteed to compute $\widehat{\alpha}$.

6.2 Other Related Work

Cover algorithms. Gulwani and Musuvathi [13] defined what they termed the "cover problem", which addresses *approximate existential quantifier elimination*:

> Given a formula φ in logic $\mathcal{L}$, and a set of variables V, find the strongest quantifier-free formula $\overline{\varphi}$ in $\mathcal{L}$ such that $[\![\exists V : \varphi]\!] \subseteq [\![\overline{\varphi}]\!]$.

They presented cover algorithms for the theories of uninterpreted functions and linear arithmetic, and showed that covers exist in some theories that do not support quantifier elimination.

The notion of a cover has similarities to the notion of symbolic abstraction, but the two notions are distinct. Although we defined symbolic abstraction as mapping between a logic $\mathcal{L}$ and an abstract domain $\mathcal{A}$, it may help to think of $\mathcal{A}$ as a *logic fragment* $\mathcal{L}'$: $\mathcal{L}'$ is defined by the image of $\widehat{\gamma}$: $\mathcal{L}' = \{\widehat{\gamma}(a) \mid a \in \mathcal{A}\}$. $\mathcal{L}'$ is often an impoverished fragment of $\mathcal{L}$. Thus, in purely logical terms, symbolic abstraction addresses the following problem of performing an *over-approximating translation to an impoverished fragment*:

> Given a formula φ in logic $\mathcal{L}$, find the strongest formula ψ in logic-fragment $\mathcal{L}'$ such that $[\![\varphi]\!] \subseteq [\![\psi]\!]$.

For instance, in the programs considered in our ARA experiments (§5), each φ in $\mathcal{L}$ is written in (arbitrary) quantifier-free bit-vector arithmetic, whereas ψ is restricted to the fragment consisting of "conjunctions of literals in quantifier-free bit-vector *affine* arithmetic".

Both cover and symbolic abstraction (deliberately) lose information from a given formula φ, and hence both result in over-approximations of $[\![\varphi]\!]$. In general, however, they yield *different* over-approximations of $[\![\varphi]\!]$.

1. The information loss from the cover operation only involves the removal of variable set V from the vocabulary of φ. The resulting formula $\overline{\varphi}$ is still allowed to be an *arbitrarily complex* $\mathcal{L}$ formula; $\overline{\varphi}$ can use all of the (interpreted) operators and (interpreted) relation symbols of $\mathcal{L}$.
2. The information loss from symbolic abstraction involves finding a formula ψ in the fragment $\mathcal{L}'$: ψ must be a *restricted* $\mathcal{L}$ formula; it can only use the operators and relation symbols of $\mathcal{L}'$, and must be written using the syntactic restrictions of $\mathcal{L}'$.

One of the uses of 2 is to bridge the gap between the concrete semantics and an abstract domain. In particular, it may be necessary to use the full power of logic $\mathcal{L}$ to state the concrete semantics of a transformer τ. However, the corresponding abstract transformer $\tau^{\#}$ *must* be expressed in $\mathcal{L}'$. When $\mathcal{L}'$ is not just the restriction of $\mathcal{L}$ to a sub-vocabulary, cover is not guaranteed to return an answer in $\mathcal{L}'$, and thus does not yield a suitable *abstract* transformer.

Consider the machine-code example from §1. Note that $\widehat{\alpha}(\varphi_I)$ is a relation over the same set of variables that appear in φ_I: $\{\texttt{eax}, \texttt{ebx}, \texttt{eax}', \texttt{ebx}'\}$. The only formulas that can be obtained from φ_I via cover are ones in which one or more of the variables is quantified out. Only the trivial cover with respect to the *empty* set of variables would be an input/output relation that retains all the variables; however, that result—namely, φ_I itself—is not expressed in the desired logic fragment, and hence is not a suitable abstract transformer in the abstract domain of affine relations over integers mod 2^{32}.

Logical abstract domains. Cousot et al. [6] define a method of abstract interpretation based on using particular sets of logical formulas as abstract-domain elements (so-called *logical abstract domains*). They face the problems of (i) performing abstraction from unrestricted formulas to the elements of a logical abstract domain [6, §7.1], and (ii) creating abstract transformers that transform input elements of a logical abstract domain to output elements of the domain [6, §7.2]. Their problems are particular cases of $\widehat{\alpha}(\varphi)$. They present heuristic methods for creating over-approximations of $\widehat{\alpha}(\varphi)$.

Connections to machine-learning algorithms. In [24], a connection was made between symbolic abstraction (in abstract interpretation) and the problem of *concept learning* (in machine learning). In machine-learning terms, an abstract domain $\mathcal{A}$ is a *hypothesis space*; each domain element corresponds to a *concept*. A hypothesis space has an *inductive bias*, which means that it has a limited ability to express sets of concrete objects. In abstract-interpretation terms, inductive bias corresponds to the image of γ on $\mathcal{A}$ not being the full power set of the concrete objects—or, equivalently, the image of $\widehat{\gamma}$ on $\mathcal{A}$ being only a fragment of $\mathcal{L}$. Given a formula φ, the symbolic-abstraction problem is to find the most specific concept that explains the meaning of φ.

$\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}$ (Alg. 1) is related to the Find-S algorithm [21, Section 2.4] for concept learning. Both algorithms start with the most-specific hypothesis (i.e., $\bot$) and work bottom-up to find the most-specific hypothesis that is consistent with positive examples of the concept. Both algorithms generalize their current hypothesis each time they process a (positive) training example that is not explained by the current hypothesis. A major difference is that Find-S receives a sequence of positive and negative examples of the concept (e.g., from nature). It discards negative examples, and its generalization steps are based solely on the positive examples. In contrast, $\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}$ already starts with a precise statement of the concept in hand, namely, the formula φ, and on each iteration, calls a decision procedure to generate the next positive example; $\widehat{\alpha}^{\uparrow}_{\mathrm{RSY}}$ never sees a negative example.

A similar connection exists between $\widetilde{\alpha}^{\updownarrow}$ (Alg. 5) and a different concept-learning algorithm, called the Candidate-Elimination algorithm [21, Section 2.5]. Both algorithms maintain two approximations of the concept, one that is an over-approximation and one that is an under-approximation.

References

1. T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In *Tools and Algs. for the Construct. and Anal. of Syst.*, pages 268–283, 2001.
2. E. Barrett and A. King. Range and set abstraction using SAT. *Electr. Notes Theor. Comp. Sci.*, 267(1), 2010.
3. J. Brauer and A. King. Automatic abstraction for intervals using Boolean formulae. In *Static Analysis Symp.*, 2010.
4. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2–3), 2004.
5. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Princ. of Prog. Lang.*, pages 269–282, 1979.
6. P. Cousot, R. Cousot, and L. Mauborgne. Logical abstract domains and interpretations. In *The Future of Software Engineering*, 2011.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear constraints among variables of a program. In *Princ. of Prog. Lang.*, pages 84–96, 1978.
8. W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *J. Sym. Logic*, 22(3), Sept. 1957.
9. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Int. Conf. on Tools and Algs. for the Construction and Analysis of Systems*, 2008.
10. B. Dutertre and L. de Moura. Yices: An SMT solver, 2006. http://yices.csl.sri.com/.
11. M. Elder, J. Lim, T. Sharma, T. Andersen, and T. Reps. Abstract domains of affine relations. In *Static Analysis Symp.*, 2011.
12. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verif.*, volume 1254 of *Lec. Notes in Comp. Sci.*, pages 72–83, 1997.
13. S. Gulwani and M. Musuvathi. Cover algorithms and their combination. In *Prog. Lang. and Systems*, 2008.
14. M. Karr. Affine relationship among variables of a program. *Acta Inf.*, 6:133–151, 1976.
15. N. Kidd, A. Lal, and T. Reps. WALi: The Weighted Automaton Library, 2007. www.cs.wisc.edu/wpis/wpds/download.php.
16. A. King and H. Søndergaard. Automatic abstraction for congruences. In *Verif., Model Checking, and Abs. Interp.*, 2010.
17. A. Lal and T. Reps. Improving pushdown system model checking. In *Computer Aided Verif.*, 2006.
18. A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *Computer Aided Verif.*, 2005.
19. J. Lassez, M. Maher, and K. Marriott. Unification revisited. In *Foundations of Logic and Functional Programming*, volume 306, pages 67–113. Springer, 1988.
20. A. Miné. The octagon abstract domain. In *Working Conf. on Rev. Eng.*, pages 310–322, 2001.
21. T. Mitchell. *Machine Learning*. WCB/McGraw-Hill, Boston, MA, 1997.
22. D. Monniaux. Automatic modular abstractions for template numerical constraints. *Logical Methods in Comp. Sci.*, 6(3), 2010.
23. J. Regehr and A. Reid. HOIST: A system for automatically deriving static analyzers for embedded systems. In *Architectural Support for Prog. Lang. and Op. Syst.*, 2004.

24. T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *Verif., Model Checking, and Abs. Interp.*, pages 252–266, 2004.
25. 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.
26. S. Sankaranarayanan, H. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *Verif., Model Checking, and Abs. Interp.*, 2005.
27. M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods in System Design*, 16(1):23–58, 2000.
28. B. Steffen, J. Knoop, and O. Rüthing. The value flow graph: A program representation for optimal program transformations. In *ESOP*, 1990.
29. B. Steffen, J. Knoop, and O. Rüthing. Efficient code motion and an adaption to strength reduction. In *TAPSOFT '91*, pages 394–415, 1991.
30. A. Thakur and T. Reps. A Generalization of Stålmarck's Method. TR 1699, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, Oct. 2011.
31. A. Thakur and T. Reps. A method for symbolic computation of precise abstract operations. In *Computer Aided Verif.*, 2012.
32. G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Tools and Algs. for the Construct. and Anal. of Syst.*, pages 530–545, 2004.

A Worked Examples: $\widehat{\alpha}^{\updownarrow}$ For Herbrand Equalities

Example 1. Consider the following code fragment, which uses two different methods for checking the parity of `i` (`i&1` and `i << 31 == 0`):

```
int i; cons_tree x, y;
x = (i&1) ? nil : cons (y,x);
if ( i << 31 == 0 ) { // (*) ... }
```

Suppose that we want an element of the Herbrand-equalities domain that relates the values of x and y at the fragment's start to x' and y', the values of those variables at program point `(*)`. A straightforward abstract interpretation of this program in the Herbrand-equalities domain would yield no information about x', because $\{x' \mapsto \text{nil}\} \sqcup \{x' \mapsto \text{cons}(y, x)\} = \{x' \mapsto x'\}$.

Alg. 5 can do better. First, symbolic execution from the beginning of the code fragment to `(*)` yields the formula

$$\varphi \overset{\text{def}}{=} (x' = \mathit{ite}(i\&1, \text{nil}, \text{cons}(y, x))) \wedge (y' = y) \wedge (i \ll 31) = 0),$$

where $\mathit{ite}(\cdot, \cdot, \cdot)$ denotes the if-the-else operator. The values obtained just after line 5 during each iteration of Alg. 5 are shown in Fig. 6. Each row of Fig. 6 displays, for a given iteration of Alg. 5,

- the values of *lower* and *upper*,

lower	*upper*	$\widehat{\gamma}(p)$	model, or unsat
$\bot$	$\top$	false	$i \mapsto 0$ $x \mapsto \text{nil}$ $x' \mapsto \text{cons}(\text{nil}, \text{nil})$ $y \mapsto \text{nil}$ $y' \mapsto \text{nil}$
$x \mapsto \text{nil}$ $x' \mapsto \text{cons}(\text{nil}, \text{nil})$ $y \mapsto \text{nil}$ $y' \mapsto \text{nil}$	$\top$	$y = \text{nil}$	$i \mapsto 0$ $x \mapsto \text{nil}$ $x' \mapsto \text{cons}(\text{cons}(\text{nil}, \text{nil}), \text{nil})$ $y \mapsto \text{cons}(\text{nil}, \text{nil})$ $y' \mapsto \text{cons}(\text{nil}, \text{nil})$
$x \mapsto \text{nil}$ $x' \mapsto \text{cons}(y, \text{nil})$ $y \mapsto y'$	$\top$	$y = y'$	unsatisfiable
$x \mapsto \text{nil}$ $x' \mapsto \text{cons}(y, \text{nil})$ $y \mapsto y'$	$y \mapsto y'$	$x = \text{nil}$	$i \mapsto 0$ $x \mapsto \text{cons}(\text{nil}, \text{nil})$ $x' \mapsto \text{cons}(\text{nil}, \text{cons}(\text{nil}, \text{nil}))$ $y \mapsto \text{nil}$ $y' \mapsto \text{nil}$
$x' \mapsto \text{cons}(y, x)$ $y \mapsto y'$	$y \mapsto y'$	$x' = \text{cons}(y, x)$	unsatisfiable
$x' \mapsto \text{cons}(y, x)$ $y \mapsto y'$	$x' \mapsto \text{cons}(y, x)$ $y \mapsto y'$		

Fig. 6. Iterations of Alg. 5 in Ex. 1. Self-mappings, e.g., $y \mapsto y$, are omitted.

– the value of $\widehat{\gamma}(p)$ computed from `AbstractConsequence`(*lower*, *upper*), and
– the model, if any, of $\varphi \wedge \neg\widehat{\gamma}(p)$ that Z3 returned.

Iterations that find a model increase the next iteration's *lower*, when $\mathit{lower} \leftarrow \mathit{lower} \sqcup \beta(S)$. Iterations that return `None` decrease the next iteration's *upper*, when $\mathit{upper} \leftarrow \mathit{upper} \sqcap p$. Note that both meet and join in this domain are maximally precise. In particular, the anti-unification procedure used for join computes the most restrictive system of equations between terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$ that permits its input. In this trial run, each call to Z3 finished in 14 milliseconds or less. The final result is $\{x \mapsto x,\ y \mapsto y,\ x' \mapsto \mathrm{cons}(y, x),\ y' \mapsto y\}$. □

The next example shows that one can use the Herbrand-equalities domain without having to give *every* function symbol its Herbrand interpretation. This approach allows one to more faithfully model the language semantics—and still use the Herbrand-equalities domain—thereby increasing the set of equalities that the analysis is capable of detecting.

Example 2. Consider the following program fragment, which is in the same programming language as the fragment from Ex. 1, extended with selectors `car` and `cdr`:

```
bool a; cons_tree x, y;
x = a ? cons(y,x) : cons(x,y);
y = cdr(x);
x = car(x);
if (a) { // (**) ... }
```

Suppose that we would like an element of the Herbrand-equalities domain that relates the values of x and y at the fragment's start to x' and y' at program point `(**)`. To reach `(**)`, `a` must be true; in this case, the code swaps the values of `x` and `y`.

lower	*upper*	$\widehat{\gamma}(p)$	model, or unsat	
$\bot$	$\top$	false	$a \mapsto \mathrm{true}$ $x \mapsto \mathrm{cons}(\mathrm{nil}, \mathrm{nil})$ $y \mapsto \mathrm{nil}$	$x' \mapsto \mathrm{nil}$ $y' \mapsto \mathrm{cons}(\mathrm{nil}, \mathrm{nil})$
$x \mapsto \mathrm{cons}(\mathrm{nil}, \mathrm{nil})$ $x' \mapsto \mathrm{nil}$ $y \mapsto \mathrm{nil}$ $y' \mapsto \mathrm{cons}(\mathrm{nil}, \mathrm{nil})$	$\top$	$y = \mathrm{nil}$	$a \mapsto \mathrm{true}$ $x \mapsto \mathrm{nil}$ $y \mapsto \mathrm{cons}(\mathrm{nil}, \mathrm{nil})$	$x' \mapsto \mathrm{cons}(\mathrm{nil}, \mathrm{nil})$ $y' \mapsto \mathrm{nil}$
$x' \mapsto y$ $x \mapsto y'$	$\top$	$x' = y$	unsatisfiable	
$x' \mapsto y$ $x \mapsto y'$	$x' \mapsto y$	$x = y'$	unsatisfiable	
$x' \mapsto y$ $x \mapsto y'$	$x' \mapsto y$ $x \mapsto y'$			

Fig. 7. Iterations of Alg. 5 in Ex. 2. Self-mappings, e.g., $y \mapsto y$, are omitted.

As in Ex. 1, a straightforward abstract interpretation of the path to `(**)` in the Herbrand-equalities domain yields $\top$. As shown in Fig. 7, symbolic abstraction, using the Herbrand-equalities domain, of the formula obtained from symbolic execution results in an abstract value that captures the swap effect.

In this example, the set of function symbols $\mathcal{F}$ over which we define the Herbrand universe $\mathcal{T}(\mathcal{F}, \emptyset)$ is $\mathcal{F} \stackrel{\text{def}}{=} \{\text{nil}, \text{cons}\}$, not $\mathcal{F} \stackrel{\text{def}}{=} \{\text{nil}, \text{cons}, \text{car}, \text{cdr}\}$. The functions car and cdr are not given their Herbrand interpretation; instead, they are interpreted as the deconstructors that select the first and second components, respectively, of a cons-term.

Symbolic execution from the beginning of the code fragment to `(**)` yields the following formula:

$$\begin{aligned}\varphi \stackrel{\text{def}}{=} x' &= \text{car}\left(\textit{ite}\left(a, \text{cons}(y,x), \text{cons}(x,y)\right)\right)\\ \wedge\, y' &= \text{cdr}\left(\textit{ite}\left(a, \text{cons}(y,x), \text{cons}(x,y)\right)\right)\\ \wedge\, a &\end{aligned}$$

As in Ex. 1, Fig. 7 shows the values obtained just after line 5 on each iteration of Alg. 5, applied to φ. Each `Model` query completes without evident delay; each invocation was 10 milliseconds or less. The final result is $x' = y$ and $x = y'$, which captures the fact that, when the program reaches `(**)`, it has swapped the values of `x` and `y`. □

B Proofs

Theorem 1. [Correctness of Alg. 5] *Suppose that $\mathcal{L}$ and $\mathcal{A}$ satisfy requirements 1–6, and $\varphi \in \mathcal{L}$. Let $a \in \mathcal{A}$ be the value returned by $\widetilde{\alpha}^{\updownarrow}\langle\mathcal{L}, \mathcal{A}\rangle(\varphi)$. Then*

1. *a over-approximates $\widehat{\alpha}(\varphi)$; i.e., $\widehat{\alpha}(\varphi) \sqsubseteq a$.*
2. *If $\mathcal{A}$ has neither infinite ascending nor infinite descending chains and $\widetilde{\alpha}^{\updownarrow}\langle\mathcal{L}, \mathcal{A}\rangle(\varphi)$ returns with no timeout, then $a = \widehat{\alpha}(\varphi)$.*

Proof. To prove part 1, we show that at each stage of Alg. 5 *lower* $\sqsubseteq \widehat{\alpha}(\varphi) \sqsubseteq$ *upper* holds. This invariant is trivially true after *upper* and *lower* are initialized in lines 1 and 2, respectively.

If control reaches line 4, then *lower* $\neq$ *upper* and *timeout* is false. Hence, at line 4, *lower* $\sqsubsetneq$ *upper*. Thus, the precondition for the call to `AbstractConsequence`(*lower*, *upper*) is satisfied, and the abstract value p returned is such that *lower* $\sqsubseteq p$ and $p \not\sqsupseteq$ *upper* (by Defn. 1).

A Galois connection $\mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$ obeys the adjointness condition

$$\text{for all } c \in \mathcal{C}, a \in \mathcal{A} : c \sqsubseteq \gamma(a) \text{ iff } \alpha(c) \sqsubseteq a. \tag{4}$$

The counterpart of Eqn. (4) for symbolic abstraction is

$$\text{for all } \varphi \in \mathcal{L}, a \in \mathcal{A} : \varphi \Rightarrow \widehat{\gamma}(a) \text{ iff } \widehat{\alpha}(\varphi) \sqsubseteq a. \tag{5}$$

If control reaches line 9, then $\varphi \wedge \neg\widehat{\gamma}(p)$ is unsatisfiable (Fig. 8(a)), which means that $\varphi \Rightarrow \widehat{\gamma}(p)$ holds. Consequently, by Eqn. (5), we know that $\widehat{\alpha}(\varphi) \sqsubseteq p$ holds. By properties of meet ($\sqcap$), we can combine the latter inequality with the invariant $\widehat{\alpha}(\varphi) \sqsubseteq \mathit{upper}$ to obtain $\widehat{\alpha}(\varphi) \sqsubseteq \mathit{upper} \sqcap p$. Hence it is safe to update *upper* by performing a meet with p; that is, after the assignment $\mathit{upper} \leftarrow \mathit{upper} \sqcap p$ on line 9, the invariant $\widehat{\alpha}(\varphi) \sqsubseteq \mathit{upper}$ still holds.

On the other hand, if $\varphi \wedge \neg\widehat{\gamma}(p)$ is satisfiable (Fig. 8(b)), then at line 11 $S \models \varphi$. Thus, $\beta(S) \sqsubseteq \widehat{\alpha}(\varphi)$. By properties of join ($\sqcup$), we can combine the latter inequality with the invariant $\mathit{lower} \sqsubseteq \widehat{\alpha}(\varphi)$ to obtain $\mathit{lower} \sqcup \beta(S) \sqsubseteq \widehat{\alpha}(\varphi)$. Hence it is safe to update *lower* by performing a join with $\beta(S)$; that is, after the assignment $\mathit{lower} \leftarrow \mathit{lower} \sqcup \beta(S)$ on line 11, the invariant $\mathit{lower} \sqsubseteq \widehat{\alpha}(\varphi)$ still holds.

In both cases, $\mathit{lower} \sqsubseteq \widehat{\alpha}(\varphi) \sqsubseteq \mathit{upper}$ holds, and thus $\mathit{lower} \sqsubseteq \widehat{\alpha}(\varphi) \sqsubseteq \mathit{upper}$ holds throughout the loop on lines 3–11.

On exiting the loop, we have $\widehat{\alpha}(\varphi) \sqsubseteq \mathit{upper}$. At line 12, *ans* is assigned the value of *upper*, which is the value returned by Alg. 5 at line 13. This finishes the proof of part 1.

We now prove part 2 of the theorem.

- At line 9, because $p \not\sqsupseteq \mathit{upper}$, $\mathit{upper} \sqcap p$ does not equal *upper*; that is, $\mathit{upper} \sqcap p \sqsubsetneq \mathit{upper}$.
- At line 11, $S \not\models \widehat{\gamma}(p)$. Because $p \sqsupseteq \mathit{lower}$, $S \not\models \widehat{\gamma}(p)$ implies that $S \not\models \widehat{\gamma}(\mathit{lower})$, and hence $\beta(S) \not\sqsubseteq \mathit{lower}$. Therefore, $\mathit{lower} \sqcup \beta(S)$ is not equal to *lower*; that is, $\mathit{lower} \sqcup \beta(S) \sqsupsetneq \mathit{lower}$.

Consequently, progress is made no matter which branch of the if-then-else on lines 8–11 is taken, and hence Alg. 5 makes progress during each iteration of the while-loop.

By part 1, $\mathit{lower} \sqsubseteq \widehat{\alpha}(\varphi) \sqsubseteq \mathit{upper}$. Consequently, if $\mathcal{A}$ has neither infinite ascending nor infinite descending chains, then eventually *lower* will be equal to *upper*, and both *lower* and *upper* will have the value $\widehat{\alpha}(\varphi)$ (provided the loop exits without a timeout). Thus, for a run of Alg. 5 on which the loop exits without a timeout, the answer returned is $\widehat{\alpha}(\varphi)$. □

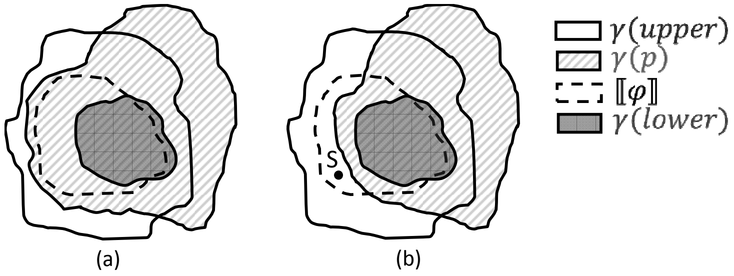


Fig. 8. The two cases arising in Alg. 5: $\varphi \wedge \neg\widehat{\gamma}(p)$ is either (a) unsatisfiable, or (b) satisfiable with $S \models \varphi$ and $S \not\models \widehat{\gamma}(p)$. (Note that although $\mathit{lower} \sqsubseteq \widehat{\alpha}(\varphi) \sqsubseteq \mathit{upper}$ and $[\![\varphi]\!] \subseteq \gamma(\mathit{upper})$ are invariants of Alg. 5, $\gamma(\mathit{lower}) \subseteq [\![\varphi]\!]$ does not necessarily hold, as depicted above.)

Theorem 2. *When $\mathcal{A}$ is a conjunctive domain over Φ, Alg. 6 is an acceptable abstract-consequence operation.*

Proof. Suppose that $a_1 \sqsubsetneq a_2$, and let $\widehat{\gamma}(a_1) = \bigwedge \Psi$, where $\Psi \subseteq \Phi$. If for each $\psi \in \Psi$ we have $\widehat{\gamma}(a_2) \Rightarrow \psi$, then $\widehat{\gamma}(a_2) \Rightarrow \bigwedge\{\psi\}$, or equivalently $\widehat{\gamma}(a_2) \Rightarrow \bigwedge \Psi$; i.e., $\widehat{\gamma}(a_2) \Rightarrow \widehat{\gamma}(a_1)$, or equivalently $a_2 \sqsubseteq a_1$, which contradicts $a_1 \sqsubsetneq a_2$. Thus, there must exist some $\psi \in \Psi$ such that $\widehat{\gamma}(a_2) \not\Rightarrow \psi$. The latter is equivalent to $\psi \not\Leftarrow \widehat{\gamma}(a_2)$, which can be written as $a \not\sqsupseteq a_2$ (where $a = \mu\widehat{\alpha}(\psi)$). Therefore, Alg. 6 will return some $a \in \mathcal{A}$ such that $a_1 \sqsubseteq a$ and $a \not\sqsupseteq a_2$. □

C A bilateral algorithm for abstract domains with infinite descending chains

Algorithm 7: $\widetilde{\alpha}^{\updownarrow}_{+}\langle \mathcal{L}, \mathcal{A} \rangle(\varphi)$

```
1  $upper \leftarrow \top$
2  $lower \leftarrow \bot$
3  $k \leftarrow 0$                                          // initialize $k$
4  while $lower \neq upper$ do
   // $lower \sqsubsetneq upper$
5    if $k < N$ then
6      $p \leftarrow$ AbstractConsequence($lower, upper$)
7    else
8      $p \leftarrow lower$
   // $p \sqsupseteq lower, p \not\sqsupseteq upper$
9    $S \leftarrow$ Model($\varphi \wedge \neg\widehat{\gamma}(p)$)
10   if $S$ is TimeOut then
11     return $upper$
12   else if $S$ is None then                              // $\varphi \Rightarrow \widehat{\gamma}(p)$
13     $upper \leftarrow upper \sqcap p$
14     $k \leftarrow k + 1$                                 // increment $k$
15   else                                                  // $S \not\models \widehat{\gamma}(p)$
16     $lower \leftarrow lower \sqcup \beta(S)$
17     $k \leftarrow 0$                                     // reset $k$
18 $ans \leftarrow upper$
19 return $ans$
```

Theorem 3. *If abstract domain $\mathcal{A}$ does not have infinite ascending chains, and Alg. 7 does not timeout, then Alg. 7 terminates and returns $\widehat{\alpha}(\varphi)$.*

Proof. The proof of Thm. 1 carries over, except that we must additionally argue that if $\mathcal{A}$ has infinite descending chains, Alg. 7 does not get trapped refining *upper* along an infinite descending chain, and that the algorithm returns $\widehat{\alpha}(\varphi)$ after *lower* has converged to $\widehat{\alpha}(\varphi)$.

Suppose that N consecutive iterations of the loop on lines 3–11 update *upper* (line 13) *without updating lower* (line 16). In the next iteration, the algorithm can set p to *lower* (line 8 so that the decision-procedure query at line 9 becomes

$\texttt{Model}(\varphi \wedge \neg\widehat{\gamma}(\mathit{lower}))$—i.e., we force the algorithm to perform the basic iteration-step from the RSY algorithm. In this way, we force *lower* to be updated at least once every N iterations.

Consequently, because $\mathcal{A}$ has no infinite ascending chains, *lower* must eventually be set to $\widehat{\alpha}(\varphi)$. After that happens, within the next N iterations of the loop body, Alg. 7 must execute line 8, which sets p to *lower* (i.e., $p = \widehat{\alpha}(\varphi)$). The model S returned from calling $\texttt{Model}(\varphi \wedge \neg\widehat{\gamma}(p))$ in line 9 must be None. As argued in the proof of Thm. 1, $\mathit{lower} \sqsubseteq \widehat{\alpha}(\varphi) \sqsubseteq \mathit{upper}$ holds on every iteration of the loop in lines 4–17. Because $\widehat{\alpha}(\varphi) \sqsubseteq \mathit{upper}$ and $p = \widehat{\alpha}(\varphi)$, the update $\mathit{upper} \leftarrow \mathit{upper} \sqcap p$ on line 13 assigns $\widehat{\alpha}(\varphi)$ to *upper*. Because *lower* is equal to *upper*, the loop exits with $\mathit{upper} = \widehat{\alpha}(\varphi)$. At line 18, *ans* is thus assigned $\widehat{\alpha}(\varphi)$, which is the value returned by Alg. 7 at line 19. □