Abstract-value Slicing

Sunae Seo[1], Hongseok Yang[2], and Kwangkeun Yi[3]

[1] Dept. of Computer Science, Korea Advanced Institute of Science and Technology
saseo@ropas.kaist.ac.kr
[2] ERC-ACI, Seoul National University
hyang@ropas.snu.ac.kr
[3] School of Computer Science and Engineering, Seoul National University
kwang@ropas.snu.ac.kr

Abstract. In our previous work[16] we have shown that by combining the abstract interpretation and Hoare logic, we can automatically construct program proofs, to be used inside the Proof-Carrying framework. The next problem of our proof-construction technique is how to minimize the generated proof size. We observed that when an abstract interpreter is used to verify a specific safety property, it often computes invariants that are not needed for the verification. The reason is that the abstract interpreter does not know what the safety property is, so it simply tries to find as strong invariants as possible. These useless invariants result in an unnecessarily big proof. Unless the proof-construction phase is notified which invariants are useless, it cannot help but proving all the computed invariants.

In this paper, we present an algorithm that slices out useless invariants from the abstract interpretation results. This algorithm works as a post-processor to an abstract interpreter in the whole proof-construction process, and notifies to the next proof-construction phase which invariants it does not have to prove. In our experiment with Miné's abstract interpretation, the algorithm identified 63% − 84% of the computed invariants as useless, and resulted in 56% − 88% reduction in the size of constructed proofs.

1 Introduction

The Proof-Carrying Code(PCC) technologies [15, 14, 1, 6] have been a convincing approach for certifying the safety of mobile code, yet how to achieve the code's safety proof is still open for alternatives. The existing proof construction process is either not fully automatic, assuming that the program invariants should be provided by the programmer [12–14], or limited to a class of properties that are automatically inferable by the current type system technologies [6, 2, 11].

In our previous work [16], we have shown that by combining the abstract interpretation [4, 3] and Hoare logic [7], we can automatically construct proofs for a wider class of program properties. The main challenge for automatically constructing Hoare proofs of programs is to find the programs' invariants, while an abstract interpreter can automatically compute such invariants, though they are approximate. An abstract interpreter automatically estimates program properties (approximate invariants) of our interests, and our proof-construction method constructs a Hoare proof for those approximate invariants. The proof-checking side (code consumer's side) is insensitive to a specific abstract interpretation; the assertions in the Hoare proofs are always first-order logic formulas for integers, into which we first compile the abstract interpreter's results.

1.1 Problem

When a Hoare proof of a property is constructed by our proof-construction method, the substantial parts of the proof are often useless; they do not contribute to the verification of the property. As a result, the constructed Hoare proof is unnecessarily big.

The main reason for this problem is that the abstract interpreter computes approximate invariants stronger than what are needed for verification. When we use an abstract interpreter to prove a specific safety property of a program, the interpreter does not know what the property is; it simply tries to discover as strong invariant as possible. As a result, it usually finds some (approximate) program invariants that are not needed to prove the safety property. Note that this existence of useless invariants becomes a bottleneck for all the efforts to reduce the size of a proof; if the proof construction does not know which invariants are of no use, it cannot help but proving all the invariants that the abstract interpreter found.

To see this problem more clearly, let's consider the following insertion sort program annotated with the results of Miné's abstract interpretation [9]:

```
sort(int n, int A[1..n])
{  int i,j,pivot;
   // true
   i:=2; j:=0;
   // inv: (2≤i) ∧ (0≤j≤i+2)
   while (i<=n) {
      // (2≤i≤n) ∧ (0≤j≤i+2)
      pivot:=A[i]; j:=i-1;
      // inv: (2≤i≤n) ∧ (0≤j≤n−1) ∧ (2≤n) ∧ (j≤i−1)
      while (j>=1 and A[j]>pivot) {
         // (2≤i≤n) ∧ (1≤j≤n−1) ∧ (2≤n) ∧ (j≤i−1)
         A[j+1]:=A[j]; j:=j-1;
      }
      // (2≤i≤n) ∧ (0≤j≤n−1) ∧ (2≤n) ∧ (j≤i−1)          --- *
      A[j+1]:=pivot; i:=i+1;
   }
}
```

This program takes an array A and the size n of the array as an input, and sorts the array. Suppose that we ran the abstract interpreter in order to verify the absence of array index errors in the program. The annotations in the program prove this safety property,[1] but they also contain unnecessary information. For instance, $i \le n$ in the annotation marked by * neither shows that the following array access A[j+1] is within bounds, nor is used to imply the loop invariant $(2 \le i) \wedge (0 \le j \le i+2)$. Thus, it can be eliminated without breaking the proof. In fact, half of the annotations in the program are not needed. If all such useless invariants are eliminated, the program becomes:

```
// true
i:=2; j:=0;
// inv: 2≤i
while (i<=n) {
   // 2≤i≤n
   pivot:=A[i]; j:=i-1;
   // inv: (2≤i) ∧ (0≤j≤n−1)
   while (j>=1 and A[j]>pivot) {
      // (2≤i) ∧ (1≤j≤n−1)
      A[j+1]:=A[j]; j:=j-1;
   }
   // (2≤i) ∧ (0≤j≤n−1)
   A[j+1]:=pivot; i:=i+1;
}
```

1.2 Our Solution

In this paper, we present an algorithm that slices out such useless invariants from the results of an abstract interpreter. Our algorithm works as a post-processor to the abstract interpreter. Given an annotated program and a property of interests, the algorithm approximates all the annotations further, until all the information in each annotation contributes to the verification of the property.

The main merit of our solution is that it does not require any changes from an abstract interpreter. An alternative approach for this problem of useless invariant is to modify an abstract interpreter, so that the interpreter becomes the goal-oriented backward one; the modified interpreter computes an under-approximation of "safe" initial states, the states from which a program always achieves the given goal. However, such an approach needs a complete redesign of the analysis, because for most forward abstract interpreters, it is not straightforward to make their backward versions, and some of them cannot be used to model under-approximation. For example, Miné's abstract domain $\mathcal{M}$ [9] does not have an "under-approximation" map δ: there are no monotone functions $\delta\colon \mathcal{M} \to \mathcal{P}(\mathsf{States})$ such that

$$\forall S \in \mathcal{P}(\mathsf{States}), \forall m \in \mathcal{M}.\ m \sqsubseteq_{\mathcal{M}} \delta(S) \iff \gamma(m) \subseteq S.$$

where γ is the concretization map for $\mathcal{M}$. The reason is that the existence of such an under-approximation implies that γ preserves all joins, but there are some joins that γ does not preserve.

[1] Here we assume that in "B_1 and B_2", B_2 is evaluated only when B_1 is true.

In this paper, we first explain our solution with Miné's abstract domain[9]. In Section 2 and 3, we review his abstract interpretation, and propose an abstract-value slicing algorithm that eliminates useless invariants from the abstract interpretation results. In Section 4, we explain the experimental results from our prototype implementation. The results show that our algorithm sliced out the 63%−84% of the computed invariants, so that it induced from 56% to 88% reduction in the size of the constructed proofs. Then, we propose a general framework for abstract-value slicing in Section 5, and conclude the paper in Section 6.

Due to the space limitation, we omit the proofs of lemmas and theorems in this paper. The omitted proofs appear in the technical report [17].

2 Miné's Abstract Interpreter

Miné's abstract interpreter[9] estimates, at each program point, the upper and lower bounds of the "distance" between two program variables (i.e., $x_i - x_j$). In this section, we review this abstract interpreter.

2.1 Programming Language

We assume that the abstract interpreter is given programs in a simple imperative language with integer variables. The syntax of the language is given below:

$$
\begin{array}{ll}
\textit{Boolean Expressions} & B ::= x{-}y \leq c \mid * \mid B \wedge B \mid B \vee B \\
\textit{Commands} & C ::= x := y{+}c \mid \texttt{skip} \\
 & \quad\;\; \mid \;\; C; C \mid \texttt{if}\, B\, \texttt{then}\, C\, \texttt{else}\, C \mid \texttt{while}\, B\, \texttt{do}\, C
\end{array}
$$

Note that the language allows only very limited forms of assignments and boolean expressions. The right-hand side of an assignment should be the c increment of a variable y; and the atomic boolean expression should have the form of $x - y \leq c$. We consider only these special forms of assignments and boolean expressions in order to focus on the key aspects of Miné's abstract interpreter; his interpreter can handle such assignments and boolean expressions well, but for other more general forms of assignments and boolean expressions, it merely uses the standard interval domain [4].

Another feature of the language is that it includes a boolean expression $*$ that nondeterministically evaluates true or false. This $*$ expression normally fills in the place where an expression has been sliced out before the analysis is applied.

We frequently use a syntactic sugar $\neg B$. It is defined inductively using the usual de-Morgan laws[2] and the equivalence: $\neg(x - y \leq c) \;\Leftrightarrow\; y - x \leq -c{-}1$.

2.2 Abstract Domain

The abstract domain for Miné's abstract interpreter consists of *difference-bound* matrices (in short, DBMs). Let N be the number of the program variables in

[2] $\neg(B_1 \wedge B_2) \Leftrightarrow (\neg B_1 \vee \neg B_2)$ and $\neg(B_1 \vee B_2) \Leftrightarrow (\neg B_1 \wedge \neg B_2)$.

a given program, and $x_1, \ldots, x_N$ an enumeration of all those variables. A DBM m for this program is an $(N+1) \times (N+1)$ matrix with integer values, $-\infty$ or ∞. Intuitively, each m_{ij} entry denotes the upper bound of $x_j - x_i$ (that is, $x_j - x_i \le m_{ij}$). A DBM m means the conjunction of all these constraints m_{ij} and $x_0 = 0$; because of this additional constraint about x_0, the range of a variable can be expressed by a DBM. For example, a DBM

$$\begin{array}{c|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & \infty \\ x_1 & \infty & \infty & -1 \\ x_2 & 0 & \infty & \infty \end{array}$$

means $x_0 - x_2 \le 0 \ \wedge \ x_2 - x_1 \le -1 \ \wedge \ x_0 = 0$, which is equivalent to $-x_2 \le 0 \ \wedge \ x_2 - x_1 \le -1$.

Let States be the set of states (i.e., $\mathsf{States} = \mathsf{Ints}^{\{x_1,\ldots,x_N\}}$). Formally, the abstract domain is defined by the lattice $\mathcal{M} = (M, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ of DBMs, concretization map $\gamma\colon M \to \mathcal{P}(\mathsf{States})$, and abstraction map $\alpha\colon \mathcal{P}(\mathsf{States}) \to M$:

$$\begin{array}{rcl} M & \triangleq & \{m \mid m \text{ is a DBM}\} \qquad m \sqsubseteq m' \overset{\Delta}{\Longleftrightarrow} \forall ij.\ m_{ij} \le m'_{ij} \\ \top_{ij} & \triangleq & \infty \qquad\qquad\qquad\qquad\quad \bot_{ij} \ \triangleq \ -\infty \\ {[m \sqcup m']}_{ij} & \triangleq & \max(m_{ij}, m'_{ij}) \qquad\quad [m \sqcap m']_{ij} \ \triangleq \ \min(m_{ij}, m'_{ij}) \\ \gamma(m) & \triangleq & \{s \in \mathsf{States} \mid \forall ij.\ s[x_0 \to 0](x_i) - s[x_0 \to 0](x_j) \le m_{ij}\} \\ \alpha(S)_{ij} & \triangleq & \max(\{s[x_0 \to 0](x_j) - s[x_0 \to 0](x_i) \mid s \in S\}) \end{array}$$

In the definition of γ and α, we used $s[x_0 \to 0]$, the extension of state s with an additional component for x_0: $s[x_0 \to 0](x_i) \triangleq \text{if } (i = 0) \text{ then } 0 \text{ else } s(x_i)$.

One special feature of this abstract domain is that it has a lower closure operator $-^*\colon M \to M$. Miné's domain is slightly different from other common abstract domains in that the concretization is not injective: two different DBMs might mean the same set of states. Moreover, among such DBMs with the same meaning, there is the "best representation" n: for all m, if $\gamma(n) = \gamma(m)$, then $n \sqsubseteq m$. The closure operator $-^*$ transforms each DBM m to this best n. That is, $m^* \sqsubseteq m$, $\gamma(m^*) = \gamma(m)$, and m^* is the smallest such DBM. More explicitly, the closure operator works as follows. Given a DBM m, it first checks whether m has a "negative cycle": a nonempty sequence $i_1 \ldots i_n$ such that $i_1 = i_n$, $0 \le i_1 \ldots i_n \le N$, and $\Sigma_{k=1}^{n-1} m_{i_k i_{k+1}} < 0$. If so, the closure m^* is $\bot$. Otherwise, each (i,j) entry is updated by the "length" of a shortest path from i to j in m: when $i = j$, the (i,j) entry of m^* is 0; and when $i \ne j$, the (i,j) entry is

$$\min\left\{ \Sigma_{k=1}^{n-1} m_{i_k i_{k+1}} \ \middle| \ 0 \le i_1 \ldots i_n \le N \wedge i_1 = i \wedge i_n = j \right\}.$$

For example,

$$\left(\begin{array}{c|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & \infty \\ x_1 & \infty & \infty & -1 \\ x_2 & 0 & \infty & \infty \end{array} \right)^* = \left(\begin{array}{c|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & 0 & \infty & \infty \\ x_1 & -1 & 0 & -1 \\ x_2 & 0 & \infty & 0 \end{array} \right).$$

Note that in the closed DBM, all the diagonal entries became 0 by definition ($i = j$ case); and the value of the x_1x_0 entry has changed to -1, because there is a path $x_1x_2x_0$ from x_1 to x_0 whose "length" is -1.

In the abstract interpreter, we use the closure operator $-^*$ in order to improve the accuracy of the analysis.

2.3 Abstract Interpreter

Given a program, the abstract interpreter computes an invariant DBM at each program point, and outputs an annotated program A of the following form:

$$\begin{aligned} A &::= [m]R[m] \\ R &::= x_i{:=}x_j{+}c \mid A;A \mid \texttt{if}\ B\ \texttt{then}\ A\ \texttt{else}\ A \mid [\mathsf{inv}\ m]\texttt{while}\ B\ \texttt{do}\ A \end{aligned}$$

The computed DBM annotations in A satisfy a set of constraints that imply the correctness of the annotations: if the constraints hold, then the annotations in A approximate program invariants. Figure 1 shows the algorithm $\mathcal{C}$ that generates such constraints. We note the two cases: one for the assignment of the form $x_i{:=}x_j{+}c$ with $i \neq j$, and the other for a `while` loop. Given an assignment of the form $[m]x_i{:=}x_j{+}c[n]$, the algorithm $\mathcal{C}$ first abstractly executes the assignment $x_i{:=}x_j{+}c$: it first transforms m so that the DBM has the best representation m^*; then, it eliminates all the information in m^* involving the old value of x_i; finally, it adds a fact that $x_i{-}x_j{\le}c$ and $x_j{-}x_i{\le}{-c}$. Once finishing this abstract execution, $\mathcal{C}$ outputs a constraint which means that the result of the abstract execution should be approximated by n. For a `while` loop

$$[m]\big([\mathsf{inv}\ \iota]\texttt{while}\ B\ \texttt{do}\ ([m_1]R[n_1])\big)[n],$$

the algorithm produces the four constraints besides those for the loop body:

$$m^* \sqsubseteq \iota,\ (n_1)^* \sqsubseteq \iota,\ [\![B]\!]\iota \sqsubseteq m_1,\ \text{and}\ [\![\neg B]\!]\iota \sqsubseteq n.$$

The first two constraints express that ι abstracts the loop invariant. The DBM ι should approximate all the initial states (i.e., $m^* \sqsubseteq \iota$) as well as all the states at the end of the loop (i.e., $(n_1)^* \sqsubseteq \iota$). The other constraints are about the states that pass or fail the test B. The third constraint prunes some states in $\gamma(\iota)$ that do not pass B, and asks that m_1 should approximate all the remaining states in $\gamma(\iota)$ (i.e., $[\![B]\!]\iota \sqsubseteq m_1$). Note that this constraint implies that $\gamma(m_1)$ includes all the states in $\gamma(\iota)$ that pass the test B. Similarly, the fourth constraint implies that n approximates all the states in $\gamma(\iota)$ that fail the test B.

Figure 2.(a) shows an annotated program A whose DBMs satisfy the constraints generated by $\mathcal{C}$. Among those generated constraints, we note the following two for the loop invariant DBM:

$$\left(\begin{array}{c|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & 0 & \infty \\ x_1 & 0 & \infty & \infty \\ x_2 & \infty & \infty & \infty \end{array}\right)^* \sqsubseteq \left(\begin{array}{c|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & \infty \\ x_1 & 0 & \infty & \infty \\ x_2 & \infty & \infty & \infty \end{array}\right) \qquad \left(\begin{array}{c|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & \infty \\ x_1 & \infty & \infty & -1 \\ x_2 & 0 & 1 & \infty \end{array}\right)^* \sqsubseteq \left(\begin{array}{c|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & \infty \\ x_1 & 0 & \infty & \infty \\ x_2 & \infty & \infty & \infty \end{array}\right)$$

Pruning Algorithm $[\![B]\!]\colon M \to M$

$[\![x_i - x_j \leq c]\!]m = m[(j,i) \to \min(m_{ji}, c)]$ $\qquad$ $[\![*]\!]m = m$

$[\![B_0 \wedge B_1]\!]m = [\![B_0]\!]m \sqcap [\![B_1]\!]m$ $\qquad$ $[\![B_0 \vee B_1]\!]m = [\![B_0]\!]m \sqcup [\![B_1]\!]m$

Constraint Generation Algorithm $\mathcal{C}$

$$\begin{aligned}&\mathcal{C}([m]x_i{:=}x_j{+}c[n])\\ &= \text{if } (i=j) \text{ then } \{\Big(m\big[(k,i)\to m_{ki}+c, (i,k)\to m_{ik}+(-c)\big]_{0\leq k(\neq i)\leq N}\Big) \sqsubseteq n\}\\ &\qquad\qquad \text{else } \{\Big(m^*\big([(k,i)\to\infty, (i,k)\to\infty]_{0\leq k(\neq i)\leq N}\big)[(j,i)\to c, (i,j)\to -c]\Big) \sqsubseteq n\}\end{aligned}$$

$$\mathcal{C}([m]\mathtt{skip}[n]) = \{m \sqsubseteq n\}$$

$$\begin{aligned}&\mathcal{C}\big([m]\big(\mathtt{if}\ B\ \mathtt{then}\ ([m_1]R_1[n_1])\ \mathtt{else}\ ([m_2]R_2[n_2])\big)[n]\big)\\ &= \big\{([\![B]\!]m) \sqsubseteq m_1,\ ([\![\neg B]\!]m) \sqsubseteq m_2,\ (n_1)^* \sqsubseteq n,\ (n_2)^* \sqsubseteq n\big\} \cup \mathcal{C}([m_1]R_1[n_1])\\ &\qquad \cup \mathcal{C}([m_2]R_2[n_2])\end{aligned}$$

$$\begin{aligned}&\mathcal{C}\big([m]\big([\mathsf{inv}\ \iota]\mathtt{while}\ B\ \mathtt{do}\ ([m_1]R[n_1])\big)[n]\big)\\ &= \big\{m^* \sqsubseteq \iota,\ (n_1)^* \sqsubseteq \iota,\ ([\![B]\!]\iota) \sqsubseteq m_1,\ ([\![\neg B]\!]\iota) \sqsubseteq n\big\} \cup \mathcal{C}([m_1]R[n_1])\end{aligned}$$

$$\begin{aligned}&\mathcal{C}([m]([m_1]R_1[n_1];[m_2]R_2[n_2])[n])\\ &= \big\{m \sqsubseteq m_1,\ n_1 \sqsubseteq m_2,\ n_2 \sqsubseteq n\big\} \cup \mathcal{C}([m_1]R_1[n_1]) \cup \mathcal{C}([m_2]R_2[n_2])\end{aligned}$$

Fig. 1. Constraints for Abstract Interpretation Results

The same DBM on the right-hand side of both constraints is a (approximate) loop invariant of the program, and it means that $-x_1 \leq 0$. The first constraint ensures that the invariant holds before the loop gets executed, and the second that it holds right after the loop body. Both of these constraints hold; whenever an entry of the DBM on the left is bigger than the corresponding entry of the other DBM, the closure operator changes it to a smaller value. For instance, in the second constraint, the x_1x_0 entry of the DBM on the left is ∞, which is bigger than the corresponding entry of the DBM on the right. The closure $-^*$ fixes this "problem", by computing a shortest path $x_1x_2x_0$, and replacing the value ∞ of the x_1x_0 entry by -1, the "distance" of this path.

In this paper, we will not specify a particular strategy to obtain annotations satisfying the constraints. It is because the correctness of the abstract-value slicer in the paper does not depend on such a strategy; it only needs that the computed DBM annotations satisfy the constraints. If a reader is interested in a particular strategy, we refer to Miné's original paper [9], where he computes the correct annotations with specific widening and narrowing operators.

(a) Abstract Interpretation Results

$$\left[\begin{array}{c|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & 0 & \infty \\ x_1 & 0 & \infty & \infty \\ x_2 & \infty & \infty & \infty \end{array}\right] : -x_1{\le}0 \wedge x_1{\le}0$$

$$\left[\text{inv} : \begin{array}{c|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & \infty \\ x_1 & 0 & \infty & \infty \\ x_2 & \infty & \infty & \infty \end{array}\right] : -x_1{\le}0$$

`while` $(x_1{-}x_2 \le 0)$ `do`

$$\left[\begin{array}{c|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & \infty \\ x_1 & 0 & \infty & \infty \\ x_2 & \infty & 0 & \infty \end{array}\right] : \begin{pmatrix} -x_1{\le}0 \\ \wedge\, x_1{-}x_2{\le}0 \end{pmatrix}$$

$x_1{:=}x_2{+}1$

$$\left[\begin{array}{c|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & \infty \\ x_1 & \infty & \infty & -1 \\ x_2 & 0 & 1 & \infty \end{array}\right] : \begin{pmatrix} -x_2{\le}0 \\ \wedge\, x_1{-}x_2{\le}1 \\ \wedge\, x_2{-}x_1{\le}{-1} \end{pmatrix}$$

$$\left[\begin{array}{c|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & \infty \\ x_1 & -1 & \infty & -1 \\ x_2 & \infty & \infty & \infty \end{array}\right] : \begin{pmatrix} -x_1{\le}0 \\ \wedge\, x_2{-}x_1{\le}{-1} \end{pmatrix}$$

(b) Sliced Results

$$\left[\begin{array}{c|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & \infty \\ x_1 & 0 & \infty & \infty \\ x_2 & \infty & \infty & \infty \end{array}\right] : \begin{array}{c} -x_1{\le}0 \\ \{(1,0)\} \end{array}$$

$$\left[\text{inv} : \begin{array}{c|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & \infty \\ x_1 & 0 & \infty & \infty \\ x_2 & \infty & \infty & \infty \end{array}\right] : \begin{array}{c} -x_1{\le}0 \\ \{(1,0)\} \end{array}$$

`while` $(x_1{-}x_2 \le 0)$ `do`

$$\left[\begin{array}{c|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & \infty \\ x_1 & 0 & \infty & \infty \\ x_2 & \infty & 0 & \infty \end{array}\right] : \begin{array}{c} \begin{pmatrix} -x_1 \le 0 \\ \wedge\, x_1{-}x_2{\le}0 \end{pmatrix} \\ \{(2,1),(1,0)\} \end{array}$$

$x_1{:=}x_2{+}1$

$$\left[\begin{array}{c|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & \infty \\ x_1 & \infty & \infty & -1 \\ x_2 & 0 & \infty & \infty \end{array}\right] : \begin{array}{c} \begin{pmatrix} -x_2{\le}0 \\ \wedge\, x_2{-}x_1{\le}{-1} \end{pmatrix} \\ \{(2,0),(1,2)\} \end{array}$$

$$\left[\begin{array}{c|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & \infty \\ x_1 & 0 & \infty & \infty \\ x_2 & \infty & \infty & \infty \end{array}\right] : \begin{array}{c} -x_1{\le}0 \\ \{(1,0)\} \end{array}$$

Fig. 2. Results of Abstract Interpretation and Abstract-value Slicing

3 Abstract-value Slicer

When the abstract interpreter in Section 2 is used for verification, it usually computes stronger invariants than what are needed. The abstract interpreter does not know what properties we want to verify, so it simply tries to find as stronger invariants as possible. In this section, we present the abstract-value slicer that weakens the computed invariants until all the information in the invariants is necessary for verification.

The abstract-value slicer takes the output A of the abstract interpreter, and information about "crucial DBM entries": the entries of the DBMs in A that are used for verification. More precisely, the second input component is a program T annotated with index sets:

$$T ::= [I]S[I]$$
$$S ::= x_i{:=}x_j{+}c \mid T;T \mid \texttt{if}\ B\ \texttt{then}\ T\ \texttt{else}\ T \mid [\mathsf{inv}\ I]\texttt{while}\ B\ \texttt{do}\ T$$

Here I denotes a set of DBM indices. We assume that the two components A and T in an input are about the same program, so that their underlying programs are identical. Each index set I in T indicates that among all the entries in the corresponding DBM m, only those in I are directly used to verify a safety property. For example, suppose that we have used the annotated program A in Figure 2.(a) to prove that $x_1 \geq 0$ holds at the end of the program. In this case, only the $x_1 x_0$ entry of the last DBM annotation is crucial, so we use the following T_{ex} as an input to the slicer:

$$[\emptyset]\Big([\mathtt{inv}:\emptyset]\mathtt{while}\ (x_1{-}x_2{\leq}0)\ \mathtt{do}\ [\emptyset]x_1 := x_2 + 1[\emptyset]\Big)[\{(1,0)\}]$$

The result of the abstract-value slicer is an index-set annotation T' that records which DBM entries are used to compute the "crucial DBM entries." Suppose that the slicer is given a DBM annotation A and an index-set annotation T. The slicer returns an index-set annotation T' that satisfies the following three properties:

1. The underlying program T' is identical to the underlying program T.
2. Every index set I in T is included in the corresponding index set I' in T'.
3. For an index set I and a DBM m, let $\mathsf{prj}(m, I)$ be the projection of m by I:

$$\mathsf{prj}(m, I)_{ij} = \begin{cases} m_{ij} & \text{if } (i,j) \in I \\ \infty & \text{otherwise} \end{cases}$$

   If we project every DBM m in A by the corresponding index set I' in T', then the annotations in the resulting program satisfy the constraints generated by $\mathcal{C}$.

The third property ensures that we can safely "eliminate" all the matrix entries not in T', and the second property means that even when we eliminate such entries, the remaining ones are still strong enough to obtain all the "crucial" information. For instance, suppose that we ran the slicer with the annotated program A in Figure 2.(a) and the index-set annotation T_{ex} in the previous paragraph. Then, the slicer computes, at each program point, which DBM entries are used to prove that $x_1 \geq 0$ holds at the end of the program. Figure 2.(b) shows the computed index-set annotation T', and the projection of each DBM in A by the corresponding index set in T'. Note that these results identify three invariants in the original annotation A as "useless": $x_1 \leq 0$ in the first DBM, $x_1{-}x_2 \leq 1$ in the DBM right after $x_1{:=}x_2{+}1$, and $x_2{-}x_1 \leq -1$ in the last DBM. The first $x_1 \leq 0$ is not necessary, because the other invariant $-x_1 \leq 0$ in the first DBM is strong enough to imply the approximate loop invariant $-x_1 \leq 0$. By the same reason, the second is not needed either. Note that the arguments for these two invariants do not depend on the input T_{ex}. On the other hand, the reason that we can eliminate $x_2{-}x_1 \leq -1$ comes from T_{ex}. Since $x_2{-}x_1 \leq -1$ is the property of the final state, but it does not "belong" to T_{ex}, we can eliminate $x_2{-}x_1 \leq -1$.

Figure 3 shows the definition of the abstract-value slicer slice in an imperative style. The second input component to slice is a program annotated with variables

$$\mathsf{clsr}(m,n,I) = \text{if } (m^* {=} \bot) \text{ then } \emptyset \text{ else } \bigcup_{(i,j)\in I}\Big(\text{if } m_{ij} \le n_{ij} \text{ then } \{(i,j)\} \text{ else } \mathsf{mPath}(m,(i,j))\Big)$$

$$\mathsf{bslice}(x_i{-}x_j{\le}c,\ m,\ n,\ I) = \begin{cases} I - \{(j,i)\} & \text{if } (n_{ji} \ge c) \\ I & \text{otherwise} \end{cases}$$

$$\mathsf{bslice}(B_1 \vee B_2,\ m,\ n,\ I) = \mathsf{bslice}(B_1,\ m,\ n,\ I) \cup \mathsf{bslice}(B_2,\ m,\ n,\ I)$$

$$\begin{aligned}\mathsf{bslice}(B_1 \wedge B_2,\ m,\ n,\ I) = \text{let } & I_1 = \{(i,j) \in I \mid ([\![B_1]\!]m)_{ij} \le n_{ij}\} \\ & I_2 = \{(i,j) \in I \mid ([\![B_2]\!]m)_{ij} \le n_{ij}\} \\ \text{in } & \mathsf{bslice}(B_1,\ m,\ n \sqcup [\![B_1]\!]m,\ I_1) \\ & \cup\ \mathsf{bslice}(B_2,\ m,\ n \sqcup [\![B_2]\!]m,\ I_2)\end{aligned}$$

$$\begin{aligned}&\mathsf{slice}([m]\texttt{skip}[n],\ [X]\texttt{skip}[Y]) \\ &\triangleq X := X \cup Y\end{aligned}$$

$$\begin{aligned}&\mathsf{slice}([m]x_i{:=}x_j{+}c[n],\ [X]x_i{:=}x_j{+}c[Y]) \\ &\triangleq \text{if } (i = j) \\ &\qquad \text{then } X := X \cup Y \\ &\qquad \text{else } X := X \cup \mathsf{clsr}\big(m,\ (n[(i,k){\to}\infty,(k,i){\to}\infty]_{0\le k\le N}),\ Y - \{(i,k),(k,i) \mid 0{\le}k{\le}N\}\big)\end{aligned}$$

$$\begin{aligned}&\mathsf{slice}([m]([m_1]R_1[n_1];[m_2]R_2[n_2])[n],\ [X]([X_1]S_1[Y_1];[X_2]S_2[Y_2])[Y]) \\ &\triangleq (Y_2 := Y \cup Y_2);\ \ \mathsf{slice}([m_2]R_2[n_2],\ [X_2]S_2[Y_2]); \\ &\quad (Y_1 := Y_1 \cup X_2);\ \ \mathsf{slice}([m_1]R_2[n_1],\ [X_1]S_1[Y_1]); \\ &\quad X := X \cup X_1\end{aligned}$$

$$\begin{aligned}&\mathsf{slice}\begin{pmatrix}[m]\big(\texttt{if } B \texttt{ then } [m_1]R_1[n_1] \texttt{ else } [m_2]R_2[n_2]\big)[n], \\ [X]\big(\texttt{if } B \texttt{ then } [X_1]S_1[Y_1] \texttt{ else } [X_2]S_2[Y_2]\big)[Y]\end{pmatrix} \\ &\triangleq (Y_1 := Y_1 \cup \mathsf{clsr}(n_1,n,Y));\ \ \mathsf{slice}\big([m_1]R_1[n_1],\ [X_1]S_1[Y_1]\big); \\ &\quad (Y_2 := Y_2 \cup \mathsf{clsr}(n_2,n,Y));\ \ \mathsf{slice}\big([m_2]R_2[n_2],\ [X_2]S_2[Y_2]\big); \\ &\quad X := X \cup \mathsf{bslice}(B,m,m_1,X_1) \cup \mathsf{bslice}(\neg B,m,m_2,X_2)\end{aligned}$$

$$\begin{aligned}&\mathsf{slice}\big([m]\big([\mathsf{inv}\ \iota]\texttt{while } B \texttt{ do } [m_1]R[n_1]\big)[n],\ [X]\big([\mathsf{inv}\ K]\texttt{while } B \texttt{ do } [X_1]S[Y_1]\big)[Y]\big) \\ &\triangleq \text{do } \{\, K := K \cup \mathsf{bslice}(\neg B,\ \iota,\ n,\ Y) \cup \mathsf{bslice}(B,\ \iota,\ m_1,\ X_1); \\ &\qquad\quad (Y_1 := Y_1 \cup \mathsf{clsr}(n_1,\ \iota,\ K));\ \ \mathsf{slice}([m_1]R[n_1],\ [X_1]S[Y_1]) \\ &\quad \}\text{ until nothing changes;} \\ &\quad X := X \cup \mathsf{clsr}(m,\iota,K)\end{aligned}$$

Fig. 3. Abstract-value Slicing

for index sets. The algorithm slice updates these index-set variables directly, so that when slice terminates, the variables contain which DBM entries are used to compute crucial DBM entries (i.e., the initial values of the variables).

Algorithm slice uses two subroutines clsr and bslice. For an index set I and a pair of DBM matrices m and n such that $m^* \sqsubseteq n$, the first subroutine $\mathsf{clsr}(m, n, I)$ discovers which entries of m are needed to get the I entries of n by the closure and the order $\sqsubseteq$. Let (i, j) be an index in I. If $m_{ij} \leq n_{ij}$, the (i, j) entry of m (i.e., $x_j - x_i \leq m_{ij}$) alone implies the (i, j) entry of n (i.e., $x_j - x_i \leq n_{ij}$). The subroutine, therefore, simply puts (i, j) to the resulting index set. On the other hand, if $m_{ij} > n_{ij}$, the (i, j) entry of m is not strong enough; in order to get the (i, j) entry of n, we need to chain several entries in a "shortest path" from i to j in m. Thus, in this case, the subroutine calls mPath to obtain a shortest path from i to j, and adds all the entries in the path to the result index set.

The other subroutine bslice is about the case pruning. Given a boolean expression B, DBMs m, n, and an index set I such that $[\![B]\!]m \sqsubseteq n$, it computes which entries of m are needed to get the I entries of n. For instance, when B is $x_i - x_j \leq c$, subroutine bslice checks whether B "implies" the $x_j x_i$ entry of n (i.e., $x_i - x_j \leq c \Rightarrow x_i - x_j \leq n_{ji}$). If so, bslice removes (j, i) from I. Otherwise, it keeps the (j, i) entry in I. Note that this case exploits the value of n_{ji}, which is available only after the abstract interpreter computes approximate invariants.

The definition of slice is given inductively on the structure of the arguments. We note the case for the assignment $x_i{:=}x_j{+}c$ with $(i \neq j)$. Given $[m]x_i{:=}x_j{+}c[n]$ and $[X]x_i{:=}x_j{+}c[Y]$, algorithm slice backtracks the abstract execution of the assignment. Recall that the abstract execution of $x_i{:=}x_j{+}c$ first replaces m by a better representation m^*, and then modifies $(i, -)$ and $(-, i)$ entries of m^*: it eliminates $(i, -)$ and $(-, i)$ entries, and updates (j, i) to c, and (i, j) to $-c$. The first step of the slicer slice is to roll-back the last updates. For all k, the slicer slice removes (k, i) and (i, k) from Y, and replaces the values of (k, i) and (i, k) entries of n by ∞. This backtracking of the updates reflects that the abstract interpreter computed (k, i) and (i, k) entries without reading any entries of the initial DBM m. The next step of the slicer is to backtrack the closure operator. Let (n_0, I_0) be the result of the first slicing step. The slicer calls the subroutine $\mathsf{clsr}(m, n_0, I_0)$ to find out how the closure of m is used to show that the I_0 entries of n_0 are implied by m. For instance, suppose that slice is given annotated loop body in Figure 2.(a) and $[X]x_1{:=}x_2{+}1[Y]$ with $X = \emptyset$ and $Y = \{(2, 0), (1, 2)\}$. Then, it first eliminates $(1, 2)$ from $\{(2, 0), (1, 2)\}$, and modifies the $x_1 x_2$ entry of the "post-DBM" by ∞. Then slice calls clsr with the pre-DBM, the modified post-DBM and the index-set $\{(2, 0)\}$. The $x_2 x_0$ entry of the modified post-DBM has value 0, while the corresponding entry of the pre-DBM has value ∞. Thus, clsr concludes that the closure has been used in this implication, so it finds a shortest path $x_2 x_1 x_0$ from x_2 to x_0 in the pre-DBM and returns the set $\{(2, 1), (1, 0)\}$ of entries in this path.

Suppose that $\mathsf{mPath}(m, (i, j))$ correctly calculates a shortest path from i to j in m: when m does not contain a negative cycle, $\mathsf{mPath}(m, (i, j))$ returns a sequence $i_1 \ldots i_n$ such that $i_1 = i$, $i_n = j$, $0 \leq i_1 \ldots i_n \leq N$, and

$$(\Sigma_{k=1}^{n-1} m_{i_k i_{k+1}}) = (m^*)_{ij}.$$

Under this supposition, all of clsr, bslice, and slice are correct.

Lemma 1. *For all DBMs m, n and index sets I, if $m^* \sqsubseteq n$, then*

$$\Big(\mathsf{prj}\big(m,\ \mathsf{clsr}(m, n, I)\big)\Big)^* \sqsubseteq \mathsf{prj}(n, I).$$

Lemma 2. *For all DBMs m and n, boolean expressions B, and index sets I, if $[\![B]\!]m \sqsubseteq n$, then $\Big([\![B]\!]\mathsf{prj}\big(m,\ \mathsf{bslice}(B, m, n, I)\big)\Big) \sqsubseteq \mathsf{prj}(n, I)$.*

Theorem 1 (Correctness). *Suppose that the abstract-value slicer is given (A, T) such that A satisfies the constraints generated by the algorithm $\mathcal{C}$, and T and A have the same underlying program. Then, $\mathsf{slice}(A, T)$ terminates, and the final value T_1 of T satisfies the following properties:*

1. *For each variable X in T, the final value of X includes the initial value of X.*
2. *When we project every annotation in A by the corresponding index set in T_1, the resulting annotated program satisfies the constraints generated by $\mathcal{C}$.*

4 Experimental Results

We tested the efficiency of the abstract-value slicer in the context of proof construction. We implemented Miné's abstract interpreter, the abstract-value slicer, and the proof construction algorithm in our previous work [16]. In our experiment, we first ran Miné's abstract interpreter with five array accessing programs, and obtained approximate invariants which are strong enough to show the absence of array bounds errors. Then, we ran the slicer for each of the computed abstract interpretation results, and measured how many invariants in the result have been eliminated. Finally, we applied the proof construction algorithm to both the original abstract interpretation results and their sliced versions, and measured how much the slicer reduces the size of the constructed proofs.

Table 1 shows how many invariants have been sliced out by the abstract-value slicer. The fourth column, labeled by "live", contains the number of all the nontrivial DBM entries in the result of the abstract interpreter, and the sixth column labeled by "useless" shows how many of those nontrivial entries the slicer found useless for verifying the absence of array bounds errors. The experimental result shows that about 63% to 84% of computed invariants are not needed for the verification.

The reduction in the size of constructed proofs is shown in Table 2. For each constructed proof, we counted the number of used Hoare logic rules, and that of used first-order logic rules. The abstract-value slicer did not reduce the number of used Hoare logic rules, because Hoare rules are applied as many times as

program	program spec. vars.[a]	labels[b]	number of DBM entries (1)live[c]	useful	(2)useless	(2)/(1)	slicing time
Insertionsort	3	22	92	22	70	76%	0.19
Partition[d]	3	27	120	45	75	63%	0.06
Bubblesort	4	32	217	42	175	81%	0.21
KMP[e]	6	45	463	133	330	72%	0.52
Heapsort	5	51	1250	203	1047	84%	0.77

[a] number of variables
[b] number of program labels; the labels are given to each command
[c] DBM entries with value m_{ij} such that $i \neq j$ and $m_{ij} \neq \infty, -\infty$
[d] Partition function in Quicksort
[e] Knuth-Morris-Pratt pattern matching algorithm

Table 1. Number of Sliced DBM Entries

program	number of Hoare rules	number of FOL rules[a] (1)abs. int.[b]	(2)value slicer[c]	(1)-(2)/(1)
Insertionsort	19	196	87	56%
Partition	24	218	94	57%
Bubblesort	32	504	137	73%
KMP	41	855	171	80%
Heapsort	53	3333	385	88%

[a] number of first-order logic rules
[b] results from Miné's abstract interpretation
[c] results from abstract-value slicer

Table 2. Reduction in the Proof Size

the number of command constructs in a program, and the abstract-value slicer does not change the program. However, the slicer reduced the number of used first-order logic rules. The experimental result shows that about 56% to 88% less rules are used for showing implications between first-order logic formulas.

5 General Framework

We now generalize the results in Section 3, and propose a framework for constructing a *correct* abstract-value slicer.

Consider an abstract interpreter $\mathcal{T}$ with the following data:

- An abstraction domain $(\mathcal{A}, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$, which is a distributive lattice.[3]

[3] A lattice $(\mathcal{A}, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is distributive if and only if

$$\forall a, b, d \in A.\, a \sqcup (b \sqcap d) = (a \sqcap b) \sqcup (a \sqcap d) \wedge a \sqcap (b \sqcup d) = (a \sqcup b) \sqcap (a \sqcup d).$$

– Concretization and abstraction maps, γ and α.
– An abstract semantics $[\![-]\!]$ for all atomic "terms": for all assignments $x := E$ and inequalities $E \leq E'$, both $[\![x := E]\!]$ and $[\![E \leq E']\!]$ are defined, and denote monotone functions on $\mathcal{A}$.

In order to construct an abstract-value slicer for $\mathcal{T}$ in our framework, we need to provide two kinds of data. First, for each $a \in \mathcal{A}$, we need to specify a *finite* sub-poset $\mathcal{A}_a$ of $\mathcal{A}$ and a monotone mapping $\alpha_a \colon \mathcal{A} \to \mathcal{A}_a$ such that

1. $\mathcal{A}_a$ is a lattice such that the least element of $\mathcal{A}_a$ is a, and all the other lattice operations of $\mathcal{A}_a$ are inherited from $\mathcal{A}$;
2. when inc is the inclusion from $\mathcal{A}_a$ to $\mathcal{A}$, the quadruple $(\mathcal{A}, \mathcal{A}_a, \alpha_a, \mathsf{inc})$ is a Galois injection;
3. α_a preserves the binary meet $\sqcap$; and
4. for all $a, b \in \mathcal{A}$, if $a \sqsubseteq b$, then for all $a' \in \mathcal{A}_a$,

$$a' \sqcup b \in \mathcal{A}_b \;\text{ and }\; a' \sqcup b = \alpha_b(a').$$

Intuitively, $\mathcal{A}_a$ denotes the set of all "sub-information" of a, and $\alpha_a(b)$ extracts the strongest sub-information of a that is implied by b. For example, in the abstract-value slicer in Section 3, the lattice $\mathcal{A}_m$ consists of all DBMs n that can be obtained by dropping some entries in m: $\forall ij.\, m_{ij} \neq n_{ij} \Rightarrow n_{ij} = \infty$. And $\alpha_m(n)$ records which entries of m are implied by the corresponding entries of n:

$$(\alpha_m(n))_{ij} = \begin{cases} \infty & \text{if } m_{ij} < n_{ij} \\ m_{ij} & \text{otherwise} \end{cases}$$

Second, we need to define a procedure btrack that calculates the "backtracking" of forward abstract computation. The procedure btrack takes three parameters. The first parameter is a monotone function f on $\mathcal{A}$, and it always has one of following three forms: $[\![x := E]\!]$, $[\![E \leq E']\!]$, and the lower closure $(\alpha \circ \gamma)$. The other two parameters are abstract elements a and b such that b approximates the result of f from a: $f(a) \sqsubseteq b$. Given such f, a and b, procedure btrack returns a monotone function of type $\mathcal{A}_b \to \mathcal{A}_a$ such that[4]

$$\forall b' \in \mathcal{A}_b.\; f(\mathsf{btrack}(f, a, b)(b')) \sqsubseteq b'.$$

Here the input b' to $\mathsf{btrack}(f, a, b)$ indicates which parts of b are used for verification, and $\mathsf{btrack}(f, a, b)$ denotes which parts of a are necessary to compute b'.

Figure 4 shows an abstract-value slicer gslice for $\mathcal{T}$ in an imperative style, and its two subroutines order and $\mathsf{gbslice}$. Note that algorithm $\mathsf{gbslice}$ and gslice generalize bslice and slice in Section 3, respectively; in bslice and slice, we use index sets to represent elements in $\mathcal{A}_a$. We state the correctness of order, $\mathsf{gbslice}$ and gslice:

[4] Here we omitted the applications of the inclusion maps.

$$\mathsf{order}(a, b, b') = \textstyle\bigsqcup_a \{a' \in \mathcal{A}_a \mid \alpha_b(a') \sqsubseteq b'\}$$

$$\mathsf{gbslice}(E{\leq}E',\, a,\, b,\, b') = \mathsf{btrack}([\![E \leq E']\!], a, b)(b')$$

$$\mathsf{gbslice}(B_1 \vee B_2,\, a,\, b,\, b') = \mathsf{gbslice}(B_1,\, a,\, b,\, b') \sqcap \mathsf{gbslice}(B_2,\, a,\, b,\, b')$$

$$\begin{aligned}\mathsf{gbslice}(B_1 \wedge B_2,\, a,\, b,\, b') = \text{let } & b_1 = [\![B_1]\!]a \text{ and } b_1' = \alpha_{b \sqcup b_1}(b' \sqcup \alpha_b(b_1)) \\ & b_2 = [\![B_2]\!]a \text{ and } b_2' = \alpha_{b \sqcup b_2}(b' \sqcup \alpha_b(b_2)) \\ \text{in } & \mathsf{gbslice}(B_1,\, a,\, b \sqcup b_1,\, b_1') \;\sqcap\; \mathsf{gbslice}(B_2,\, a,\, b \sqcup b_2,\, b_2')\end{aligned}$$

$$\mathsf{gslice}([a]\mathtt{skip}[b],\, [X]\mathtt{skip}[Y]) \triangleq X := X \sqcap \mathsf{order}(a, b, Y)$$

$$\mathsf{gslice}([a]x{:=}E[b],\, [X]x{:=}E[Y]) \triangleq X := X \sqcap \mathsf{btrack}([\![x{:=}E]\!], a, b)(Y)$$

$$\begin{aligned}&\mathsf{gslice}([a]([a_1]R_1[b_1]; [a_2]R_2[b_2])[b],\; [X]([X_1]S_1[Y_1]; [X_2]S_2[Y_2])[Y]) \\ &\triangleq (Y_2 := Y_2 \sqcap \mathsf{order}(b_2, b, Y));\;\; \mathsf{gslice}([a_2]R_2[b_2],\, [X_2]S_2[Y_2]); \\ &\quad (Y_1 := Y_1 \sqcap \mathsf{order}(b_1, a_2, X_2));\;\; \mathsf{gslice}([a_1]R_1[b_1],\, [X_1]S_1[Y_1]); \\ &\quad X := X \sqcap \mathsf{order}(a, a_1, X_1)\end{aligned}$$

$$\begin{aligned}&\mathsf{gslice}\begin{pmatrix}[a]\big(\mathtt{if}\, B\, \mathtt{then}\, [a_1]R_1[b_1]\, \mathtt{else}\, [a_2]R_2[b_2]\big)[b], \\ [X]\big(\mathtt{if}\, B\, \mathtt{then}\, [X_1]S_1[Y_1]\, \mathtt{else}\, [X_2]S_2[Y_2]\big)[Y]\end{pmatrix} \\ &\triangleq Y_1 := Y_1 \sqcap \mathsf{btrack}(\gamma \circ \alpha, b_1, b)(Y);\;\; \mathsf{gslice}\big([a_1]R_1[b_1],\, [X_1]S_1[Y_1]\big); \\ &\quad Y_2 := Y_2 \sqcap \mathsf{btrack}(\gamma \circ \alpha, b_2, b)(Y);\;\; \mathsf{gslice}\big([a_2]R_2[b_2],\, [X_2]S_2[Y_2]\big); \\ &\quad X := X \sqcap \mathsf{gbslice}(B, a, a_1, X_1) \sqcap \mathsf{gbslice}(\neg B, a, a_2, X_2)\end{aligned}$$

$$\begin{aligned}&\mathsf{gslice}\big([a]\big([\mathsf{inv}\ i]\mathtt{while}\ B\ \mathtt{do}\ [a_1]R[b_1]\big)[b],\; [X]\big([\mathsf{inv}\ K]\mathtt{while}\ B\ \mathtt{do}\ [X_1]S[Y_1]\big)[Y]\big) \\ &\triangleq \text{do } \{\;\; K := K \sqcap \mathsf{gbslice}(\neg B,\, i,\, b,\, Y) \sqcap \mathsf{gbslice}(B,\, i,\, b_1,\, X_1); \\ &\qquad\qquad Y_1 := Y_1 \sqcap \mathsf{btrack}(\gamma \circ \alpha,\, b_1,\, i)(K); \\ &\qquad\qquad \mathsf{gslice}([a_1]R[b_1],\, [X_1]S[Y_1]) \\ &\quad \} \text{ until nothing changes;} \\ &\quad X := X \sqcap \mathsf{btrack}(\gamma \circ \alpha,\, a,\, i)(K)\end{aligned}$$

Fig. 4. Generalized Abstract-value Slicing

Lemma 3. *For all abstract values a and b, if $a \sqsubseteq b$, then $\lambda b'.\mathsf{order}(a, b, b')$ is a well-defined monotone function from $\mathcal{A}_b$ to $\mathcal{A}_a$ such that*

$$\forall b' \in \mathcal{A}_b.\, \mathsf{order}(a, b, b') \sqsubseteq b'.$$

Lemma 4. *Let a, b be abstract elements and let B be a boolean expression such that $[\![B]\!]a \sqsubseteq b$. Then, $\lambda b'.\mathsf{gbslice}(B, a, b, b')$ is a well-defined monotone function from $\mathcal{A}_b$ to $\mathcal{A}_a$ such that $\forall b' \in \mathcal{A}_b.\ [\![B]\!](\mathsf{gbslice}(B, a, b, b')) \sqsubseteq b'$.*

Definition 1 (Slicing Domain). *Let A be a program annotated with abstract elements in $\mathcal{A}$. The* slicing domain *$\mathcal{A}_A$ for A consists of annotated programs A' such that*

1. *A and A' have the same underlying program; and*
2. *for all program point, if a and a' are, respectively, the annotations in A and A' at that point, then $a' \in \mathcal{A}_a$.*

The slicing domain $\mathcal{A}_A$ is ordered pointwise: for $A', A'' \in \mathcal{A}_A$, $A' \sqsubseteq A''$ iff for all program point, the annotation a' in A' at the point is less than or equal to the annotation a'' in A'' at the same program point.

Theorem 2 (Correctness). *Suppose that the abstract-value slicer is given (A, T) such that A satisfies the constraints generated by an algorithm $\mathcal{C}(A)$ in Appendix A, and the initial value A' of T is in the slicing domain $\mathcal{A}_A$. Then, $\mathsf{gslice}(A, T)$ terminates, and the final value A'' of T satisfies the following properties:*

1. *A'' is in the slicing domain $\mathcal{A}_A$;*
2. *$A'' \sqsubseteq A'$; and*
3. *A'' satisfies the constraints in $\mathcal{C}(A'')$ in Appendix A.*

6 Conclusion

In this paper, we have presented an abstract-value slicer that removes the useless parts from the abstract interpretation results. The ultimate goal of the slicer is to improve our proof-construction method in [16], which takes the program invariants computed by an abstract interpretation, and produces a Hoare proof for these invariants. Since the slicer reduces the number of invariants to prove, it resulted in smaller proofs. In our experiment with Miné's abstract interpreter, the slicer identified $63\% - 84\%$ of the abstract interpretation results as useless, and resulted in $56\% - 88\%$ reduction in the proof size.

Our abstract-value slicer is similar to program slicing [18], because it identifies all the irrelevant parts for achieving a given goal, and removes them. The objects that get sliced are, however, different: the abstract-value slicer works only on the results of an abstract interpreter, while program slicing modifies a program.

Our work is also related to the proposals for combining forward and backward abstract interpretations [5, 8]. The abstract-value slicer works backward; it backtracks each command, and finds out what parts of the "pre-abstract-value" the abstract interpreter used to compute the "crucial" parts of the "post-abstract-value." Thus, adding the abstract-value slicer as a post-processor to an abstract interpreter can be seen roughly as a forward analysis followed by a backward analysis. However, our goal is different from existing proposals. The proposals mainly concern about improving the accuracy of the analysis, or speeding up the analysis time, whereas our goal is to obtain exactly the abstract interpretation results that contribute to our verification proofs.

We currently plan to extend the abstract-value slicing algorithm in Section 3 to handle more language features such as procedures, so that we can use the slicer for more realistic programs. Another plan is to test the generality of the framework in Section 5 with specific abstract interpreters, including those for non-relational domains and the octagon domain [10].

References

1. A. W. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, June 2001.
2. A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 243–253, January 2000.
3. P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.
4. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, January 1977.
5. P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6(1):69–95, 1999.
6. N. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 89–100, June 2002.
7. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
8. D. Masse. Combining backward and forward analyses of temporal properties. In O. Danvy and A. Filinski, editors, *Proceedings of the Second Symposium PADO'2001, Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Sciences*, pages 155–172, Arhus, Denmark, 21 – 23 May 2001. Springer-Verlag, Berlin, Germany.
9. A. Miné. A new numerical abstract domain based on difference-bound matrices. In *Program As Data Objects II*, volume 2053 of *Lecture Notes in Computer Science*, pages 155–172. Springer-Verlag, May 2001.
10. A. Miné. The octagon abstract domain. In *Analysis, Slicing and Transformation (part of Working Conference on Reverse Engineering)*, IEEE, pages 310–319. IEEE CS Press, October 2001.
11. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, January 1998.
12. G. C. Necula. Proof-carrying code. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, January 1997.
13. G. C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In *Special Issue on Mobile Agent Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer-Verlag, 1997.
14. G. C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 142–154, January 2001.
15. G. C. Necula and R. Schneck. Proof-carrying code with untrusted proof rules. In *Software Security – Theories and Systems*, volume 2609 of *Lecture Notes in Computer Science*, pages 283–298. Springer-Verlag, November 2002.
16. S. Seo, H. Yang, and K. Yi. Automatic construction of Hoare proofs from abstract interpretation results. In *The First Asian Symposium on Programming Languages and Systems*, volume 2895 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, November 2003.

17. S. Seo, H. Yang, and K. Yi. Abstract-value slicing. Technical Memorandum ROPAS-2004-22, Programming Research Laboratory, School of Computer Science & Engineering, Seoul National University, September 2004. Available at http://ropas.snu.ac.kr/lib/dock/SeYaYi2004a.pdf.
18. Frank Tip. A survey of program slicing techniques. Technical Report CS-R9438, Centrum voor Wiskunde en Informatica, 1994.

A Constraints for the Results of an Abstract Interpreter

PRUNING ALGORITHM $[\![B]\!]: M \to M$

$[\![E \leq E']\!]a$ is given $\qquad [\![*]\!]a = a$

$[\![B_0 \wedge B_1]\!]a = [\![B_0]\!]a \sqcap [\![B_1]\!]a \qquad [\![B_0 \vee B_1]\!]a = [\![B_0]\!]a \sqcup [\![B_1]\!]a$

CONSTRAINT GENERATION ALGORITHM $\mathcal{C}$

$$\mathcal{C}([a]x{:=}E[b]) = \{[\![x{:=}E]\!]a \sqsubseteq b\}$$

$$\mathcal{C}([a]\texttt{skip}[b]) = \{a \sqsubseteq b\}$$

$$\begin{aligned}&\mathcal{C}\big([a]\big(\texttt{if } B \texttt{ then } ([a_1]R_1[b_1]) \texttt{ else } ([a_2]R_2[b_2])\big)[b]\big)\\ &= \big\{[\![B]\!]a \sqsubseteq a_1,\, [\![\neg B]\!]a \sqsubseteq a_2,\, (\gamma \circ \alpha)(b_1) \sqsubseteq b,\, (\gamma \circ \alpha)(b_2) \sqsubseteq b\big\} \cup \mathcal{C}([a_1]R_1[b_1])\\ &\qquad \cup \mathcal{C}([a_2]R_2[b_2])\end{aligned}$$

$$\begin{aligned}&\mathcal{C}\big([a]\big([\mathsf{inv}\ i]\texttt{while } B \texttt{ do } ([a_1]R[b_1])\big)[b]\big)\\ &= \big\{(\gamma \circ \alpha)(a) \sqsubseteq i,\, (\gamma \circ \alpha)(b_1) \sqsubseteq i,\, [\![B]\!]i \sqsubseteq a_1,\, [\![\neg B]\!]i \sqsubseteq b\big\} \cup \mathcal{C}([a_1]R[b_1])\end{aligned}$$

$$\begin{aligned}&\mathcal{C}([a]([a_1]R_1[b_1]; [a_2]R_2[b_2])[b])\\ &= \big\{a \sqsubseteq a_1,\, b_1 \sqsubseteq a_2,\, b_2 \sqsubseteq b\big\} \cup \mathcal{C}([a_1]R_1[b_1]) \cup \mathcal{C}([a_2]R_2[b_2])\end{aligned}$$

Fig. 5. Constraints for the Results of an Abstract Interpreter $\mathcal{T}$