Goal-directed Weakening of Abstract Interpretation Results

SUNAE SEO
Korea Advanced Institute of Science and Technology
HONGSEOK YANG and KWANGKEUN YI
Seoul National University
and
TAISOOK HAN
Korea Advanced Institute of Science and Technology

One proposal for automatic construction of proofs about programs is to combine Hoare logic and abstract interpretation. Constructing proofs is in Hoare logic. Discovering programs' invariants is done by abstract interpreters.

One problem of this approach is that abstract interpreters often compute invariants that are not needed for the proof goal. The reason is that the abstract interpreter does not know what the proof goal is, so it simply tries to find as strong invariants as possible. These unnecessary invariants increase the size of the constructed proofs. Unless the proof-construction phase is notified which invariants are not needed, it blindly proves all the computed invariants.

In this paper, we present a framework for designing algorithms, called *abstract-value slicers*, that slice out unnecessary invariants from the results of forward abstract interpretation. The framework provides a generic abstract-value slicer that can be instantiated into a slicer for a particular abstract interpretation. Such an instantiated abstract-value slicer works as a post-processor to an abstract interpretation in the whole proof-construction process, and notifies to the next proof-construction phase which invariants it does not have to prove. Using the framework, we designed an abstract-value slicer for an existing relational analysis and applied it on programs. In this experiment, the slicer identified $62\% - 81\%$ of the computed invariants as unnecessary, and resulted in $52\% - 84\%$ reduction in the size of constructed proofs.

Categories and Subject Descriptors: F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Mechanical verification; D.2.4 [**Software/Program Verification**]: Correctness proofs

General Terms: Algorithms, Design, Languages, Verification

Additional Key Words and Phrases: Program verification, Static analysis, Abstract interpretation, Backward analysis

Authors' addresses: S. Seo and T. Han, Department of Computer Science, Korea Advanced Institute of Science and Technology, 373-1 Guseong-dong Yuseong-gu Daejeon 305-701, Korea; email: {saseo,han}@pllab.kaist.ac.kr; H. Yang, Department of Computer Science, Queen Mary, University of London, Mile End Road, London, E1 4NS, UK; email: hyang@dcs.qmul.ac.uk; K. Yi, School of Computer Science and Engineering, Seoul National University, San 56-1 Shilim-dong Gwanak-gu Seoul 151-744, Korea; email: kwang@ropas.snu.ac.kr.

1. INTRODUCTION

Though Proof-Carrying Code(PCC) technologies [Necula and Schneck 2002; Necula and Rahul 2001; Appel 2001; Hamid et al. 2002] have been a convincing approach for certifying the safety of code, how to achieve the code's safety proof is still a matter for investigation. The existing proof construction process is either not fully automatic, assuming that the program invariants should be provided by the programmer [Necula 1997; Necula and Lee 1997; Necula and Rahul 2001], or limited to a class of properties that are automatically inferable by the current type system technologies [Hamid et al. 2002; Appel and Felty 2000; Morrisett et al. 1998].

One proposal [Seo et al. 2003] for automatic construction of proofs for a wide class of program properties is to combine abstract interpretation [Cousot and Cousot 1977; Cousot 1999] and Hoare logic [Hoare 1969]. Constructing proofs is in Hoare logic. Discovering program's invariants, which is the main challenge in automatically constructing Hoare proofs, is done by abstract interpreters [Cousot and Cousot 1977; Cousot 1999]. An abstract interpreter estimates program properties (i.e., approximate invariants) of interest, and the proof-construction method constructs a Hoare proof for those approximate invariants. The gap between the estimated invariants of an abstract interpreter and the preconditions "computed by Hoare-logic proof rules" is filled by the soundness of the abstract interpreter only, without involving any theorem provers. For instance, when the abstract interpreter's results (i.e., approximate invariants at program points – boxed properties here) are $\boxed{\mathtt{p}}\ x{:=}E\ \boxed{\mathtt{q}}$, the soundness proofs of the abstract interpreter are used to produce a proof that $\mathtt{p}$ implies the weakest precondition of $x{:=}E$ for $\mathtt{q}$.

This proof-construction method for PCC has several appealing features. An once designed abstract interpreter can be used repeatedly for all programs of the target language, as long as their properties to verify and check remain the same. Furthermore, the proof-checking side (code consumer's side) is insensitive to a specific abstract interpreter. The code consumer does not have to know which analysis technique has been used to generate the proof. The assertion language in Hoare logic is fixed to first-order logic for integers, into which we have to translate abstract interpretation results. This translation procedure is defined by referencing the concretization formulas of the used abstract interpreter. Lastly, the proof-checking side is simple. Checking the Hoare proofs is simply by pattern-matching the proof tree nodes against the corresponding Hoare logic rules. Checking if the proofs are about the accompanied code is straightforward, because the program texts are embedded in the Hoare proofs.

1.1 Problem

This work is motivated by one problem in the proof-construction method: abstract interpretation results are often unnecessarily informative for intended Hoare proofs. A (forward) abstract interpreter is usually designed to compute (approximate) program invariants that are as strong as possible, so that the computed invariants can verify a wide class of safety properties. Thus, when the abstract interpreter is used to verify one *specific* safety property, its results usually contain some (approximate) program invariants that are not necessary to prove the safety property of interest, although those invariants might be needed for some other safety prop-

erties. For instance, in our experiment with an existing relational analysis [Miné 2001], $62\% - 81\%$ of the analysis results were not needed for the intended verification.[1]

The existence of such unnecessary invariants among the results of an abstract interpretation becomes a bottleneck for all the efforts to reduce the proof size. When a Hoare proof of a safety property is constructed from the abstract interpretation results, it consists of two kinds of subproofs: the ones that the abstract interpretation results are indeed (approximate) invariants, and the others that those approximate invariants imply the safety property. The unnecessarily informative analysis results mainly cause the first kind of subproofs to "explode"; they increase the number of such subproofs, by adding useless proof subgoals.

Without addressing this problem, the proof-construction method often produces unnecessarily big Hoare proofs, hence is likely to be impractical for PCC. Big proofs accompanying mobile code degrades the code mobility in a network that usually has a limited bandwidth, or are impractical for code consumers that usually are small embedded systems with a limited memory. Note that no techniques for representing subproofs compactly by some clever encoding can solve the problem, because they assume that all subproofs are necessary; it is not the purpose of such techniques to identify the useless subproofs.

Example 1 As an example where abstract interpretation results are stronger than necessary, consider the following assignment sequence with the parity abstract interpretation, which estimates whether each program variable contains an even integer or an odd integer:

$$\texttt{x:=4x; x:=2x}$$

The estimated invariants from the abstract interpretation for variable `x` are:

$$\boxed{\top}\ \texttt{x:=4x;}\ \boxed{\textsf{even}}\ \texttt{x:=2x}\ \boxed{\textsf{even}}$$

Suppose we are interested in constructing a proof that variable `x` at the end is an even integer. Then the invariant "even" after the first assignment, which means `x` is an even integer, is stronger than needed; just $\top$ is enough. This is because for the second assignment, Hoare triple $\{true\}\texttt{x:=2x}\{\exists n.x = 2n\}$ can be derived

$$\frac{\overline{true \Rightarrow \exists n.\, 2x = 2n} \quad \overline{\{\exists n.\, 2x = 2n\}\texttt{x:=2x}\{\exists n.\, x = 2n\}}}{\{true\}\texttt{x:=2x}\{\exists n.x = 2n\}}$$

and this triple is enough to construct the intended proof:

$$\frac{\overline{\{true\}\texttt{x:=4x}\{true\}} \quad \overline{\{true\}\texttt{x:=2x}\{\exists n.\, x = 2n\}}}{\{true\}\texttt{x:=4x; x:=2x}\{\exists n.\, x = 2n\}}$$

That is, the following invariants, weaker than the original results, are just enough for our proof goal:

$$\boxed{\top}\ \texttt{x:=4x;}\ \boxed{\top}\ \texttt{x:=2x}\ \boxed{\textsf{even}}$$

□

[1] We explain this further in Section 5.

Example 2 Similarly, as another example where useless invariants occur in the results of an abstract interpretation, consider the following program, again with the parity abstract interpretation.

```
x:=1; y:=2x
```

The estimated invariants from the abstract interpretation for each variable at each program point are as follows:

$$\boxed{x\mapsto\top,\ y\mapsto\top}\ \texttt{x:=1;}\ \boxed{x\mapsto\mathsf{odd},\ y\mapsto\top}\ \texttt{y:=2x}\ \boxed{x\mapsto\mathsf{odd},\ y\mapsto\mathsf{even}}$$

Suppose we are interested in constructing a proof that variable `y` at the end is an even integer. Then, all invariants about `x` are useless. Just $\top$ for `x` is enough to reach the proof goal:

$$\dfrac{\dfrac{}{\{true\}\texttt{x:=1}\{true\}}\quad\dfrac{\overline{true\Rightarrow(\exists n.\,2x=2n)}\quad\overline{\{\exists n.\,2x=2n\}\texttt{y:=2x}\{\exists n.\,y=2n\}}}{\{true\}\texttt{y:=2x}\{\exists n.\,y=2n\}}}{\{true\}\texttt{x:=1; y:=2x}\{\exists n.\,y=2n\}}$$

Thus, the original analysis results can be weakened to the following, while still proving that `y` is even at the end:

$$\boxed{x\mapsto\top,y\mapsto\top}\ \texttt{x:=1;}\ \boxed{x\mapsto\top,y\mapsto\top}\ \texttt{y:=2x;}\ \boxed{x\mapsto\top,y\mapsto\mathsf{even}}$$

This example illustrates that the conventional program slicing technique does not immediately provide a satisfactory solution for our problem. One naive idea to eliminate the unnecessary information from the abstract interpretation result is to apply first the program slicing and then an abstract interpretation. However, for this example, this approach cannot identify any useless information from the abstract interpretation result. When the program slicing is applied to the example program with the slicing criterion "the value of variable `y` after `y:=2x`", it cannot slice out any parts of the program, because both `x:=1` and `y:=2x` affect the slicing criterion. As a result, the following parity abstract interpretation is given the unmodified original program, thus getting no help from the program slicing. Another idea might be to use dependency analysis in program slicing; to compute the dependency relationship between variables at different program points, and then to use this relationship to slice the abstract interpretation result. When this idea is applied to our example, it finds out that $x\mapsto\mathsf{odd}$ after `y:=2x` is not necessary, but it fails to discover that $x\mapsto\mathsf{odd}$ after `x:=1` is not needed for verification. Given the slicing criterion "the value of variable `y` after `y:=2x`", dependency analysis finds out that the value of variable `y` after `y:=2x` is dependent upon that of variable `x` after `x:=1`. Thus, the following slicing step does not delete $x\mapsto\mathsf{odd}$ after `x:=1`. □

Example 3 To see the problem in a "real world", we consider a slightly more realistic program — the insertion sort. Figure 1(a) shows the insertion sort, which is annotated with results of an abstract interpreter named "zone analysis" [Miné 2001]. Zone analysis estimates the upper and lower bounds of expressions x and $x-y$, for all program variables x and y. The insertion sort program takes an array `A` and the size `n` of the array as an input, and sorts the array.

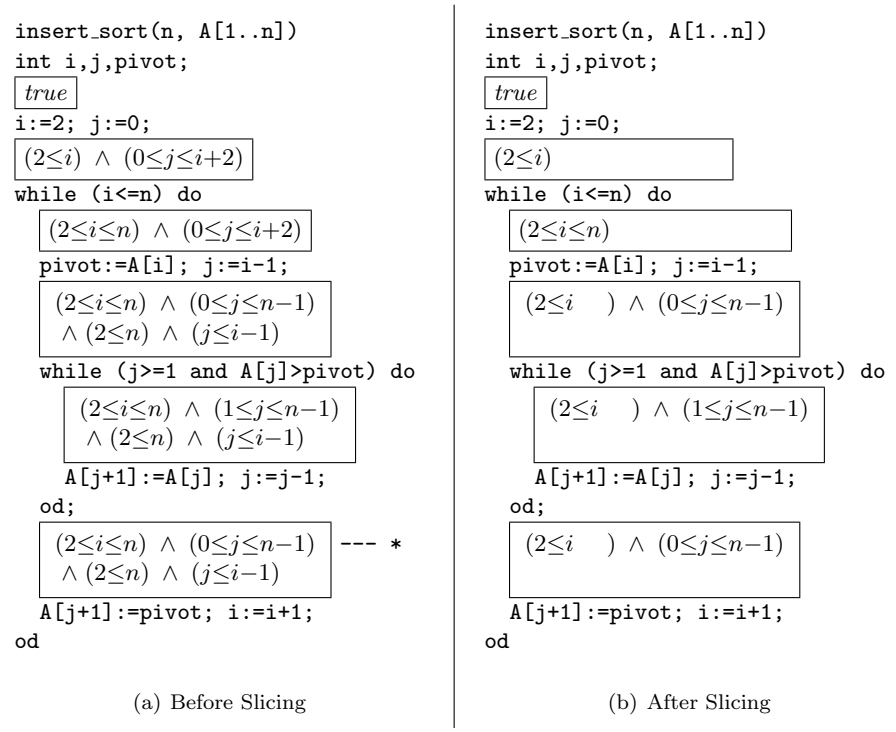


Fig. 1. Annotated Insertion Sort Before and After Slicing

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.[2]

However, note that the annotations also contain unnecessary information. For instance, $i \leq n$ in the annotation marked by `*` neither is helpful for showing that the subsequent array access `A[j+1]` is within bounds, nor is used to imply the loop invariant $(2 \leq i) \wedge (0 \leq j \leq i+2)$. Thus, it can be eliminated without breaking the proof. In fact, half of the annotations in the program are not needed. Figure 1(b) shows the program where all such useless invariants are eliminated. □

1.2 Our Solution

In this paper, we present an algorithm, called *abstract-value slicer*, that filters out unnecessary invariants from the results of a forward abstract interpreter. The abstract-value slicer works as a post-processor to the abstract interpreter. Given an annotated program and a property of interest, the slicer approximates all the annotations further, until all the information in each annotation contributes to the verification of the property.

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

The main idea of the abstract-value slicer is to view an abstract interpretation result at each program point as conjunction of formulas, and to find out which formulas in the conjunction are not necessary for verification. For instance, suppose that an abstract interpreter analyzed the assignment $x{:=}E$ for an input abstract value that means $p_1 \wedge p_2 \wedge p_3$, and that it produced an output abstract value that means $p'_1 \wedge p'_2$. That is, the abstract interpreter verified that if a pre state satisfies $p_1 \wedge p_2 \wedge p_3$, then the post state after the assignment satisfies $p'_1 \wedge p'_2$. When the abstract-value slicer is given this analysis result and it is notified that only p'_1 is used for verification, the slicer computes a subset $P \subseteq \{p_1, p_2, p_3\}$ such that (1) $\bigwedge P$ can be represented by some abstract value and (2) although $\bigwedge P$ is weaker than the original $p_1 \wedge p_2 \wedge p_3$, it is still strong enough to ensure that the assignment can achieve the goal p'_1: if the pre state satisfies $\bigwedge P$, then the post state of the assignment satisfies p'_1. Then, the slicer filters out the formulas in $\{p_1, p_2, p_3\} - P$ that are not necessary for verification: the slicer replaces the original input abstract value "$\{p_1, p_2, p_3\}$" by the abstract state that means $\bigwedge P$.

A reader might feel that a better alternative approach for solving the problem of unnecessary invariants is to use "on-line", goal-oriented backward abstract interpreters that compute the under-approximation of the weakest precondition, i.e., a set of pre states from which a program always achieves the given goal. Note that our approach is, on the other hand, in the reverse direction and "off-line" yet achieving the same effect. Given the results of forward abstract interpreters, which are already under-approximations of the weakest preconditions, we weaken the under-approximate results and make them closer from below to the weakest preconditions. Please be reminded that our problem is to *under-approximate* the weakest preconditions under which a program must always satisfy the given goal property.

Although designing such an abstract interpreter (an under-approximate backward precondition analyzer) is plausible, we pursue the idea of designing an abstract-value slicer over an existing forward abstract interpreter. That is, such abstract-value slicer is not a new analysis for estimating goal-directed invariants but an "off-line" method that reuses the results of existing abstract interpreters to achieve goal-directed invariants.

This separation (or, modularization) of the slicing from the analysis is meaningful for the reuse of the analysis. First, the analysis results can be reused. Computed analysis results for a program can be reused to achieve different slices for different slicing criteria (proof goals). The analysis itself can be reused too. For example, once an abstract interpreter is designed originally for detecting buffer-overrun errors by estimating the ranges of buffer-accessing indexes, it can be reused now to provide our slicer with invariant candidates for being used in buffer-overrun non-existence proofs. We consider forward abstract interpreters because they are most common in design and practice, having a number of realistic instances. Our method can be seen as a method for achieving goal-directed analysis results from the results of a usual, goal-independent, forward abstract interpretation.

The contributions of the paper are:

—We present a general framework for designing correct abstract-value slicers. The framework defines the generic abstract-value slicer, which we can instantiate into

a specific slicer for a particular abstract interpreter, by providing parameters. The framework also specifies the soundness conditions for those parameters of the generic slicer; if the parameters satisfy these conditions, the resulting slicer filters out only the unnecessary parts from abstract interpretation results.

—We present methods for constructing parameters of the generic abstract-value slicer, and show when all the constructed parameters by these methods satisfy the soundness conditions.

—Using our framework, we build a specific abstract-value slicer for zone analysis [Miné 2001], and demonstrate its effectiveness in the context of proof construction. In our experiment, the slicer identified that $62\%-81\%$ of the abstract interpretation results are not necessary for the verification, and resulted in $52\%-84\%$ reduction in the size of constructed program proofs.

1.3 Related Work

Our abstract-value slicer is closely related to program slicing [Tip 1995; Rival 2005a] and cone of influences [Clarke et al. 1999] in model checking. All these techniques, including ours, identify the irrelevant parts for achieving a given goal, and slice them out. The objects that get sliced are, however, different: the abstract-value slicer works only on the abstract interpretation results, while program slicing and cone of influence modify a program or a Kripke structure that models the behavior of a program.

Another important difference lies in the computation of the irrelevant parts for the goal. In order to detect the irrelevant parts, both program slicing techniques and cone of influences compute (an over-approximation of) the dependency between program variables at different program points. Intuitively, the computed dependency of y at l_1 on x at l_0 means that some *concrete* computation uses the value of x at l_0 to compute the value of y at l_1, so that the different values of x at l_0 will make y have different values at l_1. Recently, Rival [Rival 2005a] generalized this dependency in his abstract program slicing, so that the dependency is now between facts about *one variable*, such as $x > 3$ and $y < 9$, but it is still about the concrete computations. The abstract-value slicer, on the other hand, computes the dependency in the *abstract* computation between *general* facts which might involve multiple variables such as $x \leq z{+}3$ at l_0 and $z \leq y \leq z{+}9$ at l_1. For instance, suppose that an abstract interpretation result of the assignment $\mathtt{x} := \mathtt{y} - \mathtt{z}$ is $\boxed{(y{\leq}z{+}1 \wedge v{\leq}y) \wedge (y{\leq}v \wedge v{\leq}z{+}1)}\ \mathtt{x} := \mathtt{y} - \mathtt{z}\ \boxed{x{\leq}1 \wedge v{\leq}z{+}1}$, and the abstract-value slicer is asked to check whether $x{\leq}1$ in the post abstract value depends on the first conjunct $y{\leq}z{+}1 \wedge v{\leq}y$ of the pre abstract value. If by the same abstract interpretation the other conjunct can result in the same conclusion $x{\leq}1$, i.e., $\boxed{y{\leq}v \wedge v{\leq}z{+}1}\ \mathtt{x} := \mathtt{y} - \mathtt{z}\ \boxed{x{\leq}1}$, then the abstract-value slicer reports that $x{\leq}1$ does not depend on the first conjunct $y{\leq}z{+}1 \wedge v{\leq}y$. Otherwise, i.e., if the abstract interpreter approximates too much that its result from $y{\leq}v \wedge v{\leq}z{+}1$ does not imply $x{\leq}1$, then the slicer decides that $x{\leq}1$ depends on $y{\leq}z{+}1 \wedge v{\leq}y$. This is so, although the Hoare triple $\{y{\leq}v \wedge v{\leq}z{+}1\}\ \mathtt{x} := \mathtt{y} - \mathtt{z}\ \{x{\leq}1\}$ holds in the concrete semantics and thus there is no such dependency in the concrete semantics.

The dependency between general facts is also considered in the work on the abstract non-interference [Giacobazzi and Mastroeni 2004]. However, unlike our

abstract-value slicers, the dependency in the abstract non-interference is about the concrete computations, not about the abstract computations. Moreover, the existing work on the abstract non-interference does not consider the algorithm for computing the dependency relation, while the main focus of our work is to find such an algorithm.

Another related line of research is the work on abstraction refinement and predicate abstraction [Graf and Saïdi 1997; Clarke et al. 2000; Ball and Rajamani 2001; Ball et al. 2001; Henzinger et al. 2003; 2002]. The analyzers [Ball and Rajamani 2001; Henzinger et al. 2003] based on these two techniques usually find an abstract domain that is as abstract as possible but still precise enough for verifying a particular property. However, for the problem of identifying unnecessary invariants, abstraction refinement and predicate abstraction are not widely applicable, because many existing abstract interpretations are not based on those techniques. Our abstract-value slicers, on the other hand, can be applied for such abstract interpretations. We note that the analyzers based on abstraction refinement and the abstract-value slicers work in opposite "directions." Such analyzers start with naive analysis results and keep strengthening the results until they are sufficient to prove a property of interest. On the other hand, the slicers start with precise analysis results, which already prove the property of interest, and weaken the results, while maintaining the proof.

From the view point that our off-line backward abstract-value slicing after an over-approximate forward post-condition analyzer can be simulated by a single under-approximate backward precondition analyzer, related works are backward abstract interpreters that under-approximate the weakest preconditions. Please note that, however, backward abstract interpreters in [Rival 2005b; Cousot 2005; Cousot and Cousot 1999; Massé 2001; Hughes and Launchbury 1992; Duesterwald et al. 1995; Bourdoncle 1993], *over-approximate* the weakest preconditions.[3] Their backward abstract interpreters discover a superset of the pre-states where a program might generate an error. Thus such backward abstract interpreters are used in program debugging [Bourdoncle 1993] and alarm inspection [Rival 2005b]. On the other hand, abstract model checkers [Dams et al. 1997] can be seen as backward abstract interpreters that under-approximate the weakest preconditions [Cousot 1981].

Projection analysis [Wadler and Hughes 1987; Hughes 1988; Davis and Wadler 1990] in functional programs and mode analyses [King and Lu 2002; Howe et al. 2004] in logic programs both under-approximate the weakest preconditions sharing the same goal as our abstract-value slicer, but their techniques are not directly applicable to the problem of this paper. The projection analysis estimates a function that transforms the demand for the output to the one for the input. The demand for the output specifies which part of the output will be needed by the environment of the program (i.e., continuation), and the demand for the input expresses which

[3]This statement is true only for terminating deterministic programs, because those backward abstract interpreters over-approximate so called "pre state-sets." The pre state-set of a command C for a postcondition p is the set of pre states from which C can output some state satisfying p. For terminating deterministic programs, the pre state-set of C and p is the same as the weakest precondition of C and p.

property of the input is sufficient for the program to produce the necessary part of the output. Mode analysis in logic programs is similar. It estimates context properties that, if satisfied by the initial query, guarantee that the program with the query never generate any moding error. However, the domains used in these backward analyses are not as general as the ones used in our framework. Our abstract-value slicer works with complex domains that are infinite or relational, such as interval domain and zone domain [Cousot and Cousot 1977; Miné 2001], can have nontrivial domain operations (e.g., closure operation in zone domain), or can require an acceleration method (e.g., widening) for quick convergence.

1.4 Organization

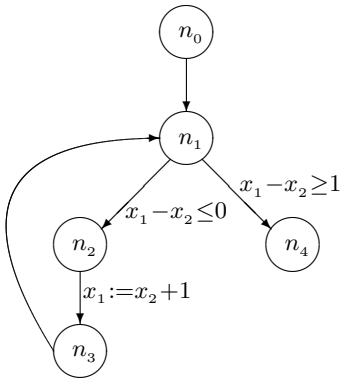
We start the paper by reviewing the basics of the abstract interpretations in Section 2. In Section 3, we define a generic abstract-value slicer parameterized by *extractor domain* and *back-tracers for assignments and boolean expressions*. Intuitively, the extractor domain specifies the working space of the slicer, and the back-tracers describe how the slicer treats each assignments and boolean expressions. In that section, we specify the soundness conditions for these two parameters, and prove that the conditions ensure the correctness of the instantiated slicer. In the next section, we present methods for constructing parameters to the slicer, which satisfy the soundness requirements. There we describe two techniques for constructing correct back-tracers. In Section 5, we explain the experimental results about one specific abstract-value slicer in the context of proof construction. Finally, we conclude the paper in Section 6.

2. ABSTRACT INTERPRETATION

We consider programs represented by control flow graphs [Cousot and Cousot 1977]. Let ATerm be the set of *atomic terms*, that is, all the inequalities $E \leq E'$, assignments $x := E$, and command `skip`. A *program* (V, E, n_i, n_f, L) is a finite graph with nodes in V and edges in E, together with two special nodes n_i and n_f, and a labeling function $L\colon E \to \mathsf{ATerm}$. A node in V represents a program point, and an edge in E a flow of control between program points; with this flow, an inequality or assignment is associated, and the labeling function L expresses this association. The special nodes n_i and n_f, respectively, denote the entry and exit of the program. We assume that in the program, no edges go into the entry node n_i, and no edges come out of the exit node n_f. Figure 2(a) shows a program that represents code with a single while loop. In this program, we label each edge with an atomic term, except when the atomic term is `skip`. Another thing to note is that the condition for exiting the loop, $\neg(x_1 - x_2 \leq 0)$, is expressed by an equivalent condition $x_1 - x_2 \geq 1$; these two conditions are equivalent since variables range over integers.

In the paper, we consider abstract interpretations that consist of three components: a join semilattice $\mathcal{A} = (A, \sqsubseteq, \bot, \sqcup)$, the abstract semantics[4] $[\![-]\!]\colon \mathsf{ATerm} \to (\mathcal{A} \to_m \mathcal{A})$ of atomic terms, and a strategy for computing post fixpoints. Given a program (V, E, n_i, n_f, L) and an initial abstract state $a_0 \in \mathcal{A}$, the abstract inter-

[4]We use the subscript $_m$ to express the monotonicity of functions. Thus, for all posets $(C, \sqsubseteq)$ and $(C', \sqsubseteq')$, $C \to_m C'$ is the poset of monotone functions from C to C'.




```
n0
n1      while (x1 - x2 <= 0) do
n2        x1 := x2 + 1
n3      od
n4
```

(a) Program

$$n_0 = n_i,\ n_4 = n_f,\ a_0 = \left[\begin{array}{c|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & 4 & 3 \\ x_1 & -1 & \infty & \infty \\ x_2 & -1 & 0 & \infty \end{array}\right]$$

(b) Entry, Exit Node and Initial Abstract State

	n_0	n_1	n_2	n_3	n_4
Analysis result as DBMs	$\begin{array}{c\|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & 4 & 3 \\ x_1 & -1 & \infty & \infty \\ x_2 & -1 & 0 & \infty \end{array}$	$\begin{array}{c\|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & 3 \\ x_1 & \infty & \infty & \infty \\ x_2 & -1 & 1 & \infty \end{array}$	$\begin{array}{c\|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & 3 \\ x_1 & \infty & \infty & \infty \\ x_2 & -1 & 0 & \infty \end{array}$	$\begin{array}{c\|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & 3 \\ x_1 & \infty & \infty & -1 \\ x_2 & -1 & 1 & \infty \end{array}$	$\begin{array}{c\|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & 3 \\ x_1 & \infty & \infty & -1 \\ x_2 & -1 & 1 & \infty \end{array}$
Analysis result as constraints	$1 \leq x_1 \leq 4$ $\wedge\ 1 \leq x_2 \leq 3$ $\wedge\ x_1 \leq x_2$	$1 \leq x_2 \leq 3$ $\wedge\ x_1 \leq x_2 + 1$	$1 \leq x_2 \leq 3$ $\wedge\ x_1 \leq x_2$	$1 \leq x_2 \leq 3$ $\wedge\ x_1 = x_2 + 1$	$1 \leq x_2 \leq 3$ $\wedge\ x_1 = x_2 + 1$

(c) Analysis Result

Fig. 2. An Example Program and Its Abstract Interpretation Result from Zone Analysis

pretation first uses $\mathcal{A}$ and $[\![-]\!]$ to define "abstract step function" F:

$$F \ : \ \prod_{n \in V} \mathcal{A} \to_m \prod_{n \in V} \mathcal{A}$$

$$F(f)(n) \stackrel{\text{def}}{=} \begin{cases} a_0 & \text{if } n = n_i \\ \bigsqcup \{ [\![L(mn)]\!](f(m)) \mid mn \in E \} & \text{otherwise} \end{cases}$$

Here $\prod_{n \in V} \mathcal{A}$ is the product join semilattice, which consists of tuples f of elements in $\mathcal{A}$ and inherits the order structure from $\mathcal{A}$ pointwise.[5] The first two components $\mathcal{A}$ and $[\![-]\!]$ of the abstract interpretation are designed so as to ensure that all the post fixpoints of this function F correctly approximate concrete program invariants. The next step of the abstract interpretation is to compute a post fixpoint of F (i.e., some f with $F(f) \sqsubseteq f$) using the post-fixpoint-computation strategy. This post fixpoint becomes the result of the abstract interpretation.

[5]For all f, g in $\prod_{n \in V} \mathcal{A}$, $f \sqsubseteq g$ iff $\forall n \in V. f(n) \sqsubseteq g(n)$. The least element $\bot$ and join $f \sqcup g$ in this join semilattice are, respectively, defined by $\lambda n. \bot$ and $\lambda n. f(n) \sqcup g(n)$.

Throughout the paper, we will use two abstract interpretations to explain the main ideas. The first one, an evenness analysis, will be mainly used to help the reader to understand the ideas themselves. The second, zone analysis, will be used to illustrate the significance and subtlety of the ideas.

Example 4 (Evenness Analysis) The goal of the evenness analysis is to discover (at each program point) the variables that always store even numbers. Let EV be a poset $\{\bot_e, \mathsf{even}, \top_e\}$ ordered by

$$\bot_e \sqsubseteq_e \mathsf{even} \sqsubseteq_e \top_e.$$

Each element in EV means a set of integers: $\bot_e$ denotes the empty set, even the set of all even integers, and $\top_e$ the set of all integers. Note that the poset EV is a join semilattice; the least element is $\bot_e$ and the join operation $\sqcup_e$ picks the bigger element among its arguments. The abstract domain $\mathcal{P} = (P, \bot, \sqsubseteq)$ of the evenness analysis is given below:

$$\begin{array}{ll} P \stackrel{\text{def}}{=} [\mathsf{Vars} \to \mathsf{EV}] & a \sqsubseteq a' \stackrel{\text{def}}{\Leftrightarrow} \forall x \in \mathsf{Vars}.\ a(x) \sqsubseteq_e a'(x) \\ \bot \stackrel{\text{def}}{=} \lambda x.\ \bot_e & a \sqcup a' \stackrel{\text{def}}{=} \lambda x.\ a(x) \sqcup_e a'(x) \end{array}$$

Intuitively, each abstract value a in P specifies which variables should have even numbers. Formally, the meaning of a is given by the following concretization map γ from P to $\mathsf{States} = [\mathsf{Vars} \to \mathsf{Ints}]$:

$$\gamma(a) \stackrel{\text{def}}{=} \begin{cases} \{\sigma \mid \forall x.\ (a(x){=}\mathsf{even} \Rightarrow \sigma(x) \text{ is even})\} & \text{if } (\forall x.\ a(x){=}\mathsf{even} \vee a(x){=}\top_e) \\ \{\} & \text{otherwise} \end{cases}$$

For the abstract semantics of each atomic term, the analysis uses the following definition:

$$\begin{array}{rll} [\![x := 2E]\!]a & \stackrel{\text{def}}{=} & a[x{\mapsto}\mathsf{even}] \\ [\![x := y]\!]a & \stackrel{\text{def}}{=} & a[x{\mapsto}a(y)] \\ [\![x := E]\!]a & \stackrel{\text{def}}{=} & a[x{\mapsto}\top_e] \quad \text{(for all the other assignments)} \\ [\![\texttt{skip}]\!]a & \stackrel{\text{def}}{=} & a \\ [\![E \le E']\!]a & \stackrel{\text{def}}{=} & a \end{array}$$

In addition, we consider a special atomic term $\texttt{even}?(x)$ only for evenness analysis, which tests whether variable x is even. Its abstract semantics is defined as follows:

$$[\![\texttt{even}?(x)]\!]a \stackrel{\text{def}}{=} \textbf{if } (\mathsf{even} \sqsubseteq_e a(x)) \textbf{ then } a[x{\mapsto}\mathsf{even}] \textbf{ else } a$$

Note that in the semantics, the information "evenness" is created by $x := 2E$, propagated by $x := y$, and removed by all the other assignments. Thus, when the analysis is given (the control flow graph of) the code "`x:=2x; y:=x; x:=1`", it returns the following annotation for the code:

$$\boxed{\begin{array}{l} x \mapsto \top_e \\ y \mapsto \top_e \end{array}}\ \texttt{x:=2x;}\ \boxed{\begin{array}{l} x \mapsto \mathsf{even} \\ y \mapsto \top_e \end{array}}\ \texttt{y:=x;}\ \boxed{\begin{array}{l} x \mapsto \mathsf{even} \\ y \mapsto \mathsf{even} \end{array}}\ \texttt{x:=1}\ \boxed{\begin{array}{l} x \mapsto \top_e \\ y \mapsto \mathsf{even} \end{array}}$$

□

Example 5 (Zone Analysis) Zone analysis [Miné 2001] estimates the upper and lower bounds on the difference $x - y$ between two program variables, using so-called difference-bound matrices (in short, DBMs). In this paper, we will use a

simplified version of zone analysis to illustrate our technique for the relational abstract interpretations. Although we use a simplified version, all the definitions and algorithms in this example are essentially the ones by Miné [Miné 2001]. Let N be the number of the program variables in a given program, and let $x_1, \ldots, x_N$ be an enumeration of all those variables. A DBM a for this program is an $(N+1)\times(N+1)$ matrix with integer values, $-\infty$ or ∞. Intuitively, each a_{ij} entry denotes the upper bound of $x_j - x_i$ (that is, $x_j - x_i \leq a_{ij}$). The row and column of a DBM include an entry for an "auxiliary variable" x_0 that never appears in the program, and is assumed to have a fixed value 0. The main role of x_0 is to allow each DBM to express the range of all the other program variables. For instance, a DBM a can store l in the $i0$-th entry (i.e., $a_{i0} = l$) for each $i \neq 0$, to record that $-l \leq x_i$. A DBM a means the conjunction of $x_0 = 0$ and all the constraints $x_j - x_i \leq a_{ij}$. Formally, the abstract domain is defined by the following join semilattice $\mathcal{M} = (M, \sqsubseteq, \bot, \sqcup)$ of the DBMs:

$$\begin{array}{llll} M & \stackrel{\text{def}}{=} & \{a \mid a \text{ is a DBM}\} & a \sqsubseteq a' \stackrel{\text{def}}{\Leftrightarrow} \forall ij.\ a_{ij} \leq a'_{ij} \\ \bot_{ij} & \stackrel{\text{def}}{=} & -\infty & [a \sqcup a']_{ij} \stackrel{\text{def}}{=} \max(a_{ij}, a'_{ij}) \end{array}$$

The formal meaning of each DBM is given by a concretization map (i.e., meaning function) γ from M to the powerset of states:

$$\begin{array}{ll} \mathsf{States} & \stackrel{\text{def}}{=} [\{x_1, \ldots, x_N\} \to \mathsf{Ints}] \\ \gamma(a) & \stackrel{\text{def}}{=} \{\sigma \in \mathsf{States} \mid \forall ij.\ \sigma[x_0 \mapsto 0](x_j) - \sigma[x_0 \mapsto 0](x_i) \leq a_{ij}\} \end{array}$$

where $\sigma[x_0 \mapsto 0]$ means the extension of state σ with an additional component for x_0: $\sigma[x_0 \mapsto 0](x_i) \stackrel{\text{def}}{=} \textbf{if } (i = 0) \textbf{ then } 0 \textbf{ else } \sigma(x_i)$. For instance, a_0 in Figure 2(b) means the conjunction of five constraints for variables x_1 and x_2; these constraints say that x_1 and x_2 are, respectively, in the intervals $[1, 4]$ and $[1, 3]$, and that x_1 is at most as big as x_2. Note that all the diagonal entries of a_0 are ∞, while those entries, meaning the upper bounds for $x_i - x_i$, could be tighter bound 0. In this paper, we decide to use ∞, rather than 0, for diagonal entries of DBMs, because both ∞ and 0 at the diagonal positions provide no information about concrete states and this is clarified by ∞.

The analysis classifies atomic terms into two groups, and defines the abstract semantics of the terms in each group in a different style. The first group includes atomic terms whose execution can be precisely modelled by DBM transformations. It consists of inequalities of the form $x_i \leq c$, $x_i \geq c$, $x_i - x_j \leq c$, assignments of the form $x_i := x_i + c$, $x_i := x_j + c$, $x_i := c$, and command `skip`. For each inequality $E \leq E'$ in the group, $[\![E \leq E']\!]$ calculates the conjunction of $E \leq E'$ and the constraints denoted by the input DBM; for each assignment $x := E$ in this group, $[\![x := E]\!]$ computes its strongest postcondition for the input DBM; and $[\![\texttt{skip}]\!]$ is defined to be the identity function. The semantics of these atomic terms is shown in Figure 3. The abstract semantics of $x_i - x_j \leq c$ in Figure 3 implements the pruning of states, by updating the ji-th entry of the input DBM a by $\min(a_{ji}, c)$. Note that the updated DBM means precisely the conjunction of $x_i - x_j \leq c$ and $\gamma(a)$. Thus, among the states in (the concretization of) the input DBM, $[\![x_i - x_j \leq c]\!]$ filters out the states that violate the condition $x_i - x_j \leq c$, and returns a DBM for the remaining states. The abstract semantics $[\![x_i := x_i + c]\!]$ models the increment of

$$
\begin{array}{lcl}
[\![x_i \le c]\!]a & \stackrel{\text{def}}{=} & a[0i \mapsto \min(a_{0i}, c)] \\
[\![x_i \ge c]\!]a & \stackrel{\text{def}}{=} & a[i0 \mapsto \min(a_{i0}, -c)] \\
[\![x_i - x_j \le c]\!]a & \stackrel{\text{def}}{=} & a[ji \mapsto \min(a_{ji}, c)] \\
[\![E \le E']\!]a & \stackrel{\text{def}}{=} & a \quad \text{(for all other inequalities)} \\
[\![x_i := x_i + c]\!]a & \stackrel{\text{def}}{=} & a\big[ki \mapsto (a_{ki}+c),\ ik \mapsto (a_{ik}+(-c))\big]_{0 \le k(\ne i) \le N} \\
[\![x_i := x_j + c]\!]a & \stackrel{\text{def}}{=} & a^*\big([ki \mapsto \infty,\ ik \mapsto \infty]_{0 \le k(\ne i) \le N}\big)[ji \mapsto c,\ ij \mapsto (-c)] \\
[\![x_i := c]\!]a & \stackrel{\text{def}}{=} & a^*\big([ki \mapsto \infty,\ ik \mapsto \infty]_{0 \le k(\ne i) \le N}\big)[0i \mapsto c,\ i0 \mapsto (-c)] \\
[\![x_i := E]\!]a & \stackrel{\text{def}}{=} & a^*\big([ki \mapsto \infty,\ ik \mapsto \infty]_{0 \le k(\ne i) \le N}\big) \quad \text{(for all other assignments)} \\
[\![\texttt{skip}]\!]a & \stackrel{\text{def}}{=} & a
\end{array}
$$

Fig. 3. Abstract Semantics of Atomic Terms in the Zone Analysis

x_i by c. For every kl-th entry of a, if the column index l is i, so the entry means the constraint involving x_i, $[\![x_i := x_i + c]\!]$ increments the entry by c; and if the row index k is i, so the entry now means the constraint involving $-x_i$, not x_i, $[\![x_i := x_i + c]\!]$ decrements the entry by c. The case $[\![x_i := x_j + c]\!]$ in the figure is the most complex and interesting. Given a DBM a, the semantic function $[\![x_i := x_j + c]\!]$ first transforms a, so that the DBM has the smallest element a^* among the ones that mean the same state set as a: a^* satisfies $\gamma(a) = \gamma(a^*)$, and for all other such DBMs a' (i.e., $\gamma(a') = \gamma(a)$), $a^* \sqsubseteq a'$. We call a^* the *closure* of a. Zone analysis computes this closure using the *Floyd-Warshall* shortest path algorithm.[6] Next, $[\![x_i := x_j + c]\!]$ eliminates all the information in a^* involving the old value of x_i. Finally, it adds two facts, $x_i - x_j \le c$ and $x_j - x_i \le -c$.

The atomic terms in the other group are interpreted "syntactically": the semantics of an assignment $x_i := E$ in this group does not consider the expression E, and transforms an input DBM a to the following x_i-deleted DBM:

$$a^*\big([ki \mapsto \infty,\ ik \mapsto \infty]_{0 \le k(\ne i) \le N}\big)$$

and the semantics of $E \le E'$ in the group prunes nothing, and means the identity function on the DBMs. A better alternative is to use interval analysis to give the semantics of atomic terms in the second group as shown by Miné [Miné 2001]. In Section 5, we will discuss this better semantics and other improvements used in the original zone analysis [Miné 2001].

Figure 2(c) shows a result of the (simplified) zone analysis in the form of DBMs and constraints. The input to zone analysis is the program in Figure 2(a) and the DBM a_0 in Figure 2(b). The result implies that when the program terminates, x_2 is in the interval $[1, 3]$ and it is equal to $x_1 - 1$. □

3. ABSTRACT-VALUE SLICER

An abstract-value slicer is an algorithm that filters out unnecessary information from the result of an abstract interpretation. When an abstract interpretation is

[6] When the shortest path algorithm is applied, program variables are considered nodes in the graph and each DBM entry a_{ji} is regarded as the weight of the edge from node x_j to node x_i.

used for verification, it usually computes stronger invariants than needed. This situation commonly happens, because an abstract interpretation is usually designed and implemented to blindly estimate best possible invariants at each program point without considering global goal of the intended verification, normally assuming that every aspect of a program potentially contributes to the properties of interest. However, in the verification of a *specific* safety property, only *some* aspects of the program are usually needed. As a result, the abstract interpretation results are likely to contain unnecessary information for such verification. Actually, this situation results from a design choice too, because we want one abstract interpretation to estimate invariants once for multiple verifications of different safety properties.

The goal of an abstract-value slicer is to weaken the computed invariants until no information in the invariants is unnecessary for a specific verification. Mathematically, the abstract-value slicer lifts the result f of an abstract interpretation: it computes a new post fixpoint f' of the abstract step function F in Section 2, such that $f \sqsubseteq f'$, but f' is still strong enough to prove the properties of interest. Intuitively, the "difference" between f and f' represents the information filtered out from f by the abstract-value slicer.

In this section, we define the abstract-value slicers, and prove their correctness. First, we introduce *extractor domain* and *back-tracers for atomic terms*, which are two main components of an abstract-value slicer. An extractor domain determines the working space of an abstract-value slicer, i.e., a poset where the abstract-value slicer does the fixpoint computation, and back-tracers specify how the abstract-value slicer treats atomic terms: they describe how the slicer filters out unnecessary information from the abstract interpretation results for atomic terms. Then, we define an abstract-value slicer, and prove its correctness. Throughout the section, we assume a fixed abstract interpretation, and denote its abstract domain and abstract semantics of atomic terms by $\mathcal{A} = (A, \sqsubseteq, \bot, \sqcup)$ and $[\![-]\!]$, respectively.

3.1 Extractor Domain

An *extractor domain* for the abstract interpretation $(\mathcal{A}, [\![-]\!])$ is a pair $(\mathcal{E}, \mathsf{ex})$ where $\mathcal{E}$ is a complete lattice $(\sqsubseteq_\mathcal{E}, \bot_\mathcal{E}, \top_\mathcal{E}, \sqcup_\mathcal{E}, \sqcap_\mathcal{E})$ with finite height[7] and ex is a monotone map from $\mathcal{E}$ to upper closure operators on $\mathcal{A}$.[8] Intuitively, each element e in $\mathcal{E}$ denotes an "information extractor" that selects some information from abstract values a in $\mathcal{A}$, which is to be saved/preserved, and $\mathsf{ex}(e)(a)$ extracts information (to be saved/preserved) from a based on the extractor e. Note that we require $\mathsf{ex}(e)$ to be an upper closure operator, i.e., a monotone function that satisfies extensiveness and idempotency requirements. The extensiveness requirement means that the extracting operation $\mathsf{ex}(e)(a)$ lifts the value a. When an extractor e is applied to the abstract value a, it does not insert any new information, but only selects some information from a; thus, $\mathsf{ex}(e)(a)$ should have less information than a (i.e., $a \sqsubseteq \mathsf{ex}(e)(a)$). The idempotency requirement formalizes that the extraction by $\mathsf{ex}(e)$ is done all at once. We also point out that ex should be monotone with respect

[7] We don't need to require the completeness of $\mathcal{E}$, since all lattices with finite height are complete. However, we put the completeness requirement explicitly here to simplify presentation.

[8] An upper closure operator ρ on $\mathcal{A}$ is a monotone function on $\mathcal{A}$ such that ρ is extensive (i.e., $\mathsf{id} \sqsubseteq \rho$) and idempotent (i.e., $\rho \circ \rho = \rho$).

to the order $\sqsubseteq_{\mathcal{E}}$ on extractors. This monotonicity condition ensures that the order $\sqsubseteq_{\mathcal{E}}$ means the "strength" of the information extractors in the reverse direction: if $e \sqsubseteq_{\mathcal{E}} e'$, then e' extracts less information than e.

We call $\mathcal{E}$ *extractor lattice* and ex *extractor application*. We often omit the subscript $-_{\mathcal{E}}$ in the lattice operators, $\sqsubseteq_{\mathcal{E}}, \bot_{\mathcal{E}}, \top_{\mathcal{E}}, \sqcup_{\mathcal{E}}, \sqcap_{\mathcal{E}}$, when the missing subscript can be recovered from context.

Example 6 We use the following extractor domain for the evenness analysis:

$$\mathcal{E} \stackrel{\text{def}}{=} \wp(\mathsf{Vars}) \ \text{(ordered by } \supseteq) \quad \text{and} \quad \mathsf{ex}(e)(a) \stackrel{\text{def}}{=} \lambda x.\, \textbf{if } x \in e \textbf{ then } a(x) \textbf{ else } \top_{e}.$$

In this extractor domain, each abstract value a is regarded as the conjunction of information "$x{\mapsto}a(x)$" for all $x \in \mathsf{Vars}$, and the extractors in $\mathcal{E}$ indicate which information should be selected from such conjunction. For instance, an abstract value $[x{\mapsto}\mathsf{even}, y{\mapsto}\mathsf{even}, z{\mapsto}\mathsf{even}]$ is regarded as $\mathit{even}(x) \wedge \mathit{even}(y) \wedge \mathit{even}(z)$, where the predicate $\mathit{even}(x)$ asserts that x is even, and the extractor $\{x, y\}$ expresses that only the first and second conjuncts, $\mathit{even}(x) \wedge \mathit{even}(y)$, should be selected. Note that the definition formalizes such selection using the top element $\top_{e}$ of EV: all the unselected information is replaced by $\top_{e}$, while the other selected information remains as it is. The extractor $\mathsf{ex}(e)$ for evenness analysis is an upper closure operator: it is extensive because $\forall a \in \mathcal{A}.\, \forall x \in \mathsf{Vars}.\, a(x) \sqsubseteq \mathsf{ex}(e)(a)(x)$, and idempotent because $\forall a \in \mathcal{A}.\, \mathsf{ex}(e)\Big(\mathsf{ex}(e)(a)\Big) = \mathsf{ex}(e)(a)$. □

Example 7 We construct an extractor domain $(\mathcal{E}, \mathsf{ex})$ for zone analysis, using a set of matrix indices as an information extractor. The idea is to use each index set e to specify which entries of the DBM matrices should be extracted. For each DBM matrix a, $\mathsf{ex}(e)(a)$ selects only the entries of a whose indices are in e, and it fills in the other missing entries by ∞. For example, when the extractor $\{(2,1)\}$ is applied to the DBM a_{0} in Figure 2(b), it filters out all entries except the $(2,1)$-th entry, and results in the below DBM, which means $x_{1} - x_{2} \leq 0$.

	x_0	x_1	x_2
x_0	∞	∞	∞
x_1	∞	∞	∞
x_2	∞	0	∞

Let N be the number of program variables, so that the domain $\mathcal{M}$ of DBMs consists of $(N+1) \times (N+1)$ matrices, and let I be the index set $(N+1) \times (N+1)$. The precise definition of $\mathcal{E}$ and ex is given below:

$$\mathcal{E} \stackrel{\text{def}}{=} \langle \wp(I), \supseteq, I, \emptyset, \cap, \cup \rangle \quad \text{and} \quad \big(\mathsf{ex}(e)(a)\big)_{ij} \stackrel{\text{def}}{=} \begin{cases} a_{ij} & \text{if } ij \in e \\ \infty & \text{otherwise.} \end{cases}$$

Note that the extractor lattice uses the superset order; thus, a smaller extractor selects more matrix entries from the input DBM than a bigger one. □

3.2 Back-tracers

Let $(\mathcal{E}, \mathsf{ex})$ be an extractor domain for the abstract interpretation $(\mathcal{A}, [\![-]\!])$. For each atomic term t, define $\mathsf{prepost}(t)$ by

$$\mathsf{prepost}(t) \stackrel{\text{def}}{=} \{(a, b) \mid a, b \in \mathcal{A} \wedge [\![t]\!]a \sqsubseteq b\},$$

which means the pre and post conditions of the triples $\{a\}t\{b\}$ for t that can be proved by the abstract interpretation.

Definition 3.1 (Back-tracer) A *back-tracer k for an atomic term t* is a function of type $\mathsf{prepost}(t) \to \mathcal{E} \to \mathcal{E}$ that satisfies the following soundness condition:

$$\forall (a,b) \in \mathsf{prepost}(t).\ \forall e, e' \in \mathcal{E}.\ \big(k_{ab}(e) = e'\big) \implies [\![t]\!]\Big(\mathsf{ex}(e')(a)\Big) \sqsubseteq \mathsf{ex}(e)(b).$$

The back-tracer k_{ab} at (a,b) transforms a post extractor e (for b) to a pre extractor e' (for a). The soundness condition ensures that the e'-part of a is sufficient to get the e-part of b in the abstract interpretation.

A back-tracer induces a map from $\{\mathsf{ex}(e)(b) \mid e \in \mathcal{E}\}$ to $\{\mathsf{ex}(e)(a) \mid e \in \mathcal{E}\}$, when it satisfies

$$\mathsf{ex}(e)(b) = \mathsf{ex}(e')(b) \implies \mathsf{ex}\big(k_{ab}(e)\big)(a) = \mathsf{ex}\big(k_{ab}(e')\big)(a).$$

The domain and codomain of this map represent pieces of information from b and a, respectively, so the map indicates that the back-tracer transforms a piece of information of b to another piece of information of a.

Note that the back-tracer k_{ab} for an atomic term t is not required to be monotone. Thus, it is relatively easy to design one correct back-tracer. For example, suppose that $\mathsf{ex}(\bot)$ is the identity function.[9] Then, for every $e \in \mathcal{E}$, there exists $e' \in \mathcal{E}$ such that

$$[\![t]\!]\Big(\mathsf{ex}(e')(a)\Big) \sqsubseteq \mathsf{ex}(e)(b),$$

because $\bot$ could be e'. We can now define $k_{ab}(e)$ to be one such e'. However, designing a good back-tracer, not just correct one, is a nontrivial problem, and requires insights about the abstract interpreter and the extractor domain.

Our next result, Proposition 3.4, is concerned with this issue of designing good back-tracers. It gives a sufficient and necessary condition for the *existence* of the best back-tracers. In Section 4, we will discuss this issue further, and provide general techniques for *implementing* good back-tracers, including the best ones.

Definition 3.2 (Best Back-tracer) A back-tracer k for an atomic term t is *best* if and only if it is the greatest back-tracer for t[10]: for all back-tracers k' for t,

$$\forall (a,b) \in \mathsf{prepost}(t).\ \forall e \in \mathcal{E}.\quad k'_{ab}(e) \sqsubseteq k_{ab}(e).$$

Lemma 3.3 A function $k : \mathsf{prepost}(t) \to \mathcal{E} \to \mathcal{E}$ is the best back-tracer for t if and only if for all $(a,b) \in \mathsf{prepost}(t)$ and all $e, e' \in \mathcal{E}$,[11]

$$e' \sqsubseteq k_{ab}(e) \iff [\![t]\!]\Big(\mathsf{ex}(e')(a)\Big) \sqsubseteq \mathsf{ex}(e)(b).$$

[9] This supposition holds for all the examples in this paper.

[10] This definition can be rephrased as follows. A function k is the best back-tracer for t if and only if k is the least upper bound of all back-tracers for t and k itself is a back-tracer for t.

[11] The equivalence between the order relationships is reminiscent of Galois connection. We cannot use Galois connection directly here, because $k_{ab} : \mathcal{E} \to \mathcal{E}$ and $[\![t]\!] : \mathcal{A} \to \mathcal{A}$ are not type-correct for Galois connection. We resolve this type error using functions $\mathsf{ex}(-)(a)$ $\mathsf{ex}(-)(b)$ of type $\mathcal{E} \to \mathcal{A}$.

Proof: First, we prove the only-if direction. Suppose that k is the best back-tracer for t, and choose arbitrary $(a, b) \in \mathsf{prepost}(t)$ and $e, e' \in \mathcal{E}$. We need to show the equivalence:

$$e' \sqsubseteq k_{ab}(e) \iff [\![t]\!]\Big(\mathsf{ex}(e')(a)\Big) \sqsubseteq \mathsf{ex}(e)(b).$$

Suppose that the left hand side of the equivalence holds. Then,

$$\begin{array}{rcll} [\![t]\!]\Big(\mathsf{ex}(e')(a)\Big) & \sqsubseteq & [\![t]\!]\Big(\mathsf{ex}\Big(k_{ab}(e)\Big)(a)\Big) & \text{(by the monotonicity of } [\![t]\!] \text{ and } \mathsf{ex}(-)(a)) \\ & \sqsubseteq & \mathsf{ex}(e)(b) & \text{(by the soundness of the back-tracer } k). \end{array}$$

Thus, the right hand side of the equivalence holds as well. Now suppose that the right hand side of the equivalence holds. Define a function k' as follows:

$$\begin{array}{rcl} k' & : & \mathsf{prepost}(t) \to \mathcal{E} \to \mathcal{E} \\ k'_{cd}(e_0) & \stackrel{\text{def}}{=} & \textbf{if } ((c, d, e_0) = (a, b, e)) \textbf{ then } e' \textbf{ else } k_{cd}(e_0). \end{array}$$

k' is a back-tracer because it satisfies the condition for back-tracers: when the arguments are (a, b, e), $k'_{ab}(e) = e'$ holds and the right hand side of the equivalence directly means the condition for back-tracers, and when the arguments are not (a, b, e), k' is the same as k, which is already a back-tracer. Since k is a best back-tracer, $k' \sqsubseteq k$ holds, which implies the left hand side of the equivalence as follows:

$$e' = k'_{ab}(e) \sqsubseteq k_{ab}(e).$$

Next, we prove the if direction. Suppose that k satisfies the equivalence in the lemma. Then, for all back-tracers k' for t, and all $(a, b) \in \mathsf{prepost}(t)$ and $e \in \mathcal{E}$,

$$[\![t]\!]\Big(\mathsf{ex}\big(k'_{ab}(e)\big)(a)\Big) \sqsubseteq \mathsf{ex}(e)(b) \quad \text{by the soundness of back-tracer } k',$$

and so, by the equivalence in the lemma, $k'_{ab}(e) \sqsubseteq k_{ab}(e)$. We just have shown that k' is smaller than or equal to k. It remains to show that k is a back-tracer for t, i.e., it satisfies the soundness requirement for back-tracers for t. To show the requirement, consider arbitrary $(a, b) \in \mathsf{prepost}(t)$ and $e \in \mathcal{E}$. Let e' be $k_{ab}(e)$. Since $e' \sqsubseteq k_{ab}(e)$, the equivalence in the lemma gives

$$[\![t]\!]\Big(\mathsf{ex}(e')(a)\Big) \sqsubseteq \mathsf{ex}(e)(b).$$

Note that this order relationship is precisely the soundness requirement for back-tracers for t. □

Proposition 3.4 An atomic term t has the best back-tracer if for all $a \in \mathcal{A}$, the function $\lambda e.[\![t]\!](\mathsf{ex}(e)(a)) : \mathcal{E} \to_m \mathcal{A}$ preserves all finite joins. Moreover, when $\mathsf{ex}(\bot)$ is the identity function on $\mathcal{A}$, the converse holds as well.

Proof: First, we prove the if direction. Suppose that for all $a \in \mathcal{A}$, function $\lambda e.[\![t]\!](\mathsf{ex}(e)(a))$ of type $\mathcal{E} \to_m \mathcal{A}$ preserves all finite joins. Define a function k as follows:

$$\begin{array}{rcl} k & : & \mathsf{prepost}(t) \to \mathcal{E} \to \mathcal{E} \\ k_{ab}(e) & \stackrel{\text{def}}{=} & \bigsqcup\{e_0 \mid [\![t]\!](\mathsf{ex}(e_0)(a)) \sqsubseteq \mathsf{ex}(e)(b)\}. \end{array}$$

We will show that k satisfies the following equivalence in Lemma 3.3: for all $e, e' \in \mathcal{E}$,

$$e' \sqsubseteq k_{ab}(e) \iff [\![t]\!]\Big(\mathsf{ex}(e')(a)\Big) \sqsubseteq \mathsf{ex}(e)(b).$$

The right-to-left implication follows from the definition of k. For the left-to-right implication, it is sufficient (because $\lambda e.[\![t]\!](\mathsf{ex}(e)(a))$ is monotone) to prove the following condition:

$$e' = k_{ab}(e) \implies [\![t]\!]\Big(\mathsf{ex}(e')(a)\Big) \sqsubseteq \mathsf{ex}(e)(b).$$

Suppose that $e' = k_{ab}(e)$. Since the extractor domain has finite height, there exists a finite nonempty subset E_0 of $\{e_0 \mid [\![t]\!](\mathsf{ex}(e_0)(a)) \sqsubseteq \mathsf{ex}(e)(b)\}$ such that $e' = \bigsqcup E_0$. Note that by the assumption of the if direction, function $\lambda e.\,[\![t]\!](\mathsf{ex}(e)(a))$ preserves finite joins. From this join preservation and the choice of E_0, we derived the required implication:

$$\begin{array}{lll}
[\![t]\!](\mathsf{ex}(e')(a)) & = & [\![t]\!](\mathsf{ex}(\bigsqcup E_0)(a)) \qquad\qquad (\text{since } e' = \bigsqcup E_0) \\
& = & \bigsqcup_{e_0 \in E_0} [\![t]\!](\mathsf{ex}(e_0)(a)) \quad (\text{by the join preservation of } [\![t]\!](\mathsf{ex}(-)(a))) \\
& \sqsubseteq & \mathsf{ex}(e)(b) \qquad\qquad (\text{since } \forall e_0 \in E_0.\ [\![t]\!]\big(\mathsf{ex}(e_0)(a)\big) \sqsubseteq \mathsf{ex}(e)(b)).
\end{array}$$

Next, we show the only if direction, assuming that $\mathsf{ex}(\bot)$ is the identity on $\mathcal{A}$. Suppose that t has the best back-tracer k. Then, it satisfies the equivalence in Lemma 3.3. Consider a finite family $\{e_i\}_{i \in I}$ of extractors. Then, by the monotonicity of ex and $[\![t]\!]$, we have that

$$[\![t]\!]\Big(\mathsf{ex}\Big(\bigsqcup_{i \in I} e_i\Big)(a)\Big) \sqsupseteq \bigsqcup_{i \in I} [\![t]\!]\Big(\mathsf{ex}(e_i)(a)\Big).$$

Thus, to show that the join in $\bigsqcup_{i \in I} e_i$ is preserved, we only need to prove the other order relationship. Let b be $\bigsqcup_{i \in I} [\![t]\!]\Big(\mathsf{ex}(e_i)(a)\Big)$. Then,

$$\begin{array}{lll}
 & \forall i \in I.\ [\![t]\!]\Big(\mathsf{ex}(e_i)(a)\Big) \sqsubseteq b & (\text{by the definition of } b) \\
\iff & \forall i \in I.\ [\![t]\!]\Big(\mathsf{ex}(e_i)(a)\Big) \sqsubseteq \mathsf{ex}(\bot)(b) & (\text{since } \mathsf{ex}(\bot) \text{ is the identity}) \\
\iff & \forall i \in I.\ e_i \sqsubseteq k_{ab}(\bot) & (\text{by the equivalence in Lemma 3.3}) \\
\iff & \Big(\bigsqcup_{i \in I} e_i\Big) \sqsubseteq k_{ab}(\bot) & \\
\iff & [\![t]\!]\Big(\mathsf{ex}\Big(\bigsqcup_{i \in I} e_i\Big)(a)\Big) \sqsubseteq \mathsf{ex}(\bot)(b) & (\text{by the equivalence in Lemma 3.3}) \\
\iff & [\![t]\!]\Big(\mathsf{ex}\Big(\bigsqcup_{i \in I} e_i\Big)(a)\Big) \sqsubseteq b & (\text{since } \mathsf{ex}(\bot) \text{ is the identity}).
\end{array}$$

We have just shown the required order relationship. □

Example 8 We define a back-tracer for each atomic term for the evenness analysis. Recall the abstract domain $\mathcal{P}$ of the evenness analysis in Example 4, and the extractor domain $(\mathcal{E}, \mathsf{ex})$ for the analysis in Example 6. For a back-tracer for an atomic term t, we use $(\!|t|\!)$ for notational convenience. The back-tracer $(\!|t|\!)$ for each

atomic term t in this domain is defined as follows:

$$\begin{aligned}
(\!|x := 2E|\!)_{ab}(e) &\stackrel{\text{def}}{=} e{-}\{x\},\\
(\!|x := y|\!)_{ab}(e) &\stackrel{\text{def}}{=} \textbf{if } x \in e \textbf{ then } (e{-}\{x\}) \cup \{y\} \textbf{ else } e,\\
(\!|x := E|\!)_{ab}(e) &\stackrel{\text{def}}{=} e{-}\{x\} \quad \text{(for all the other assignments)},\\
(\!|\texttt{skip}|\!)_{ab}(e) &\stackrel{\text{def}}{=} e,\\
(\!|E \le E'|\!)_{ab}(e) &\stackrel{\text{def}}{=} e,\\
(\!|\texttt{even}?(x)|\!)_{ab}(e) &\stackrel{\text{def}}{=} \textbf{if } (\mathsf{even} \sqsubseteq_e b(x)) \textbf{ then } (e - \{x\}) \textbf{ else } e.
\end{aligned}$$

The back-tracer for every assignment $x := E$ has the same pattern. Given an extractor e, it first deletes x from e, and then adds (to the resulting extractor) the variables used in the *abstract* semantics. Note that this is similar to the DEF-USE calculation in the conventional data-flow analysis. The main difference is that the back-tracer computes the DEF-USE for the abstract semantics, not for the concrete semantics. For instance, when the back-tracer $(\!|\mathtt{x} := \mathtt{y} + \mathtt{z}|\!)_{ab}$ is applied to the extractor $\{x, v\}$, it deletes x from the extractor and returns $\{v\}$. The variables y, z are not added to the extractor even though they are read by assignments in the concrete semantics. This is because y, z are not used by the abstract semantics of the assignments. The back-tracer for evenness test $\texttt{even}?(x)$ uses the analysis results critically. The function $[\![\texttt{even}?(x)]\!]$ refines the input a by replacing $a(x)$ by the minimum of $a(x)$ and even. Thus, for pre and post conditions $(a, b) \in \mathsf{prepost}(\texttt{even}?(x))$ (i.e., $[\![\texttt{even}?(x)]\!]a \sqsubseteq b$), if $\mathsf{even} \sqsubseteq_e b(x)$, then the x component of a is not necessary to obtain the x component of b. The back-tracer $(\!|\texttt{even}?(x)|\!)_{ab}$ correctly captures this using the analysis result b; it first tests whether $\mathsf{even} \sqsubseteq_e b(x)$, and if so, it deletes x from the given extractor e. □

Example 9 Recall the abstract domain $\mathcal{M}$ for zone analysis in Example 5 and the extractor domain $(\mathcal{E}, \mathsf{ex})$ for the analysis in Example 7. The back-tracer $(\!|t|\!)$ for an atomic term t in this extractor domain should be a parameterized index-set transformer that satisfies the following condition: for all pre and post conditions $(a, b) \in \mathsf{prepost}(t)$ (i.e., $[\![t]\!]a \sqsubseteq b$) and all extractors e for b, the computed index set $(\!|t|\!)_{ab}(e)$ contains (the indices of) all the entries of a that are necessary for obtaining the e entries of b. We define such a back-tracer $(\!|t|\!)$ as a two-step computation. First, $(\!|t|\!)_{ab}(e)$ deletes all the indices ij from e that satisfy $b_{ij} = \infty$ (i.e., $e{-}\{ij \mid b_{ij} = \infty\}$). Then, for each remaining ij-th entry of e, $(\!|t|\!)_{ab}(e)$ computes the entries of a that are needed for obtaining $([\![t]\!]a)_{ij}$, collects all the computed entries, and returns the set of the collected indices. Note that all the deleted indices ij in the first step select only the empty information from b: for all index sets e, we have that $\mathsf{ex}_b(e) = \mathsf{ex}_b(e{-}\{ij\})$. Thus, the first step only makes e have a better representation e' (i.e., $e' \subseteq e$), without changing its effect on b. Another thing to note is that the second step is concerned with only a and $[\![t]\!]a$, but not b. Here the second step exploits the fact that to get the e' entries of b, we need only the e' entries of $[\![t]\!]a$.

The actual implementation $(\!|t|\!)$ for each atomic term t optimizes the generic two-step computation, and it is shown in Figure 4. The most interesting part is the last case $x_i := E$. The back-tracer $(\!|x_i := E|\!)_{ab}(e)$ first checks whether the input matrix a has a negative cycle, that is, a sequence $k_0 k_1 \ldots k_n$ of integers in $[0, N]$ such that $k_0 = k_n$, $n \ge 1$, and $\sum_{m=0}^{n-1} a_{k_m k_{m+1}} < 0$. If a has a negative cycle, $(\!|x_i := E|\!)_{ab}(e)$

picks a shortest such cycle (i.e., one with smallest n), and returns the set of all the "edges" $k_m k_{m+1}$ in the cycle. If a does not have a negative cycle, $(\!|x_i := E|\!)_{ab}(e)$ eliminates all the indices kl from e such that $b_{kl} = \infty$ or $i = k \vee i = l$. Then, for each remaining index kl in e, $(\!|x_i := E|\!)_{ab}(e)$ selects a sequence $k_0 \ldots k_n$ of integers in $[0, N]$ such that $k_0 = k$, $k_n = l$, and

$$\Big(\sum_{m=0}^{n-1} a_{k_m k_{m+1}}\Big) \;\leq\; (b)_{kl}.$$

The formula on the left hand side of the above inequality computes an upper bound of $(a^*)_{kl}$, and the inequality means that this upper bound is still tight enough to prove that $(a^*)_{kl} \leq (b)_{kl}$. The set of all the "edges" $k_m k_{m+1}$ in these selected paths is the result of the back-tracer $(\!|x_i := E|\!)_{ab}(e)$.

When $(\!|x_i := E|\!)_{ab}(e)$ chooses a path from k to l in the second step, it usually picks one with the minimum weight, denoted $\mathsf{mPath}(a, k, l)$.[12] However, when $a_{kl} \leq b_{kl}$, $(\!|x_i := E|\!)_{ab}(e)$ selects a possibly different and shorter path kl. Note that the selected path for kl here might be different from the path that the abstract interpretation has used to compute the kl-th entry of b. This shows that $(\!|x_i := E|\!)_{ab}(e)$ does not necessarily denote the part of a that the abstract interpretation *has used* to obtain the e-part of b; instead it means the part of a that the abstract interpretation *can use* to get the e-part of b.

To see how the back-tracer works more clearly, consider the following pre and post conditions of $x_3 := 0$:[13]

$$[\![x_3 := 0]\!] \left(\begin{array}{c|cccc} & x_0 & x_1 & x_2 & x_3 \\ \hline x_0 & \infty & 4 & 3 & \infty \\ x_1 & -1 & \infty & 4 & \infty \\ x_2 & -1 & 0 & \infty & \infty \\ x_3 & \infty & \infty & \infty & \infty \end{array}\right) \;\sqsubseteq\; \left(\begin{array}{c|cccc} & x_0 & x_1 & x_2 & x_3 \\ \hline x_0 & \infty & 5 & \infty & 0 \\ x_1 & \infty & \infty & 2 & \infty \\ x_2 & \infty & \infty & \infty & \infty \\ x_3 & 0 & \infty & \infty & \infty \end{array}\right)$$

Let a and b be, respectively, the left and right DBMs of this relationship. When the back-tracer $(\!|x_3 := 0|\!)_{ab}$ is given a post extractor $e = \{(0,1),(1,2),(2,1),(3,0)\}$ it first gets rid of $(2,1)$ and $(3,0)$ from e, because the $(2,1)$-th entry of b has ∞ and the $(3,0)$-th entry of b is generated by the assignment $x_3 := 0$. Then, the back-tracer $(\!|x_3 := 0|\!)_{ab}$ computes paths for $(0,1)$ and $(1,2)$ separately; for $(0,1)$, it picks the path $0,1$, because $a_{01} \leq b_{01}$; and for the other index $(1,2)$, condition $a_{12} \leq b_{12}$ does not hold, and so the back-tracer computes a path from 1 to 2 with the minimum weight, which is the sequence $1,0,2$. Finally, it returns the index set $\{(0,1),(1,0),(0,2)\}$ that consists of all the edges in the computed two paths. □

[12] $\mathsf{mPath}(a,k,l)$ is a path $k_0 \ldots k_n$ such that $k_0 = k, k_n = l$ and

$$(\sum_{m=0}^{n-1} a_{k_m k_{m+1}}) = (a^*)_{kl}.$$

[13] The second DBM can arise, for instance, when the assignment $x_3 := 0$ is executed at the end of the false branch of a conditional statement and the analysis result of the true branch is a DBM whose $(0,2),(1,0),(2,0),(2,1)$ entries are ∞ and $(0,1)$ entry is 5.

$$
\begin{array}{rcl}
(\!|x_i \le c|\!)_{ab}(e) & \stackrel{\text{def}}{=} & \textbf{if } (b_{0i} \ge c) \textbf{ then } (e - \{kl \mid b_{kl} = \infty\} - \{0i\}) \\
& & \qquad \textbf{else } (e - \{kl \mid b_{kl} = \infty\}) \\
(\!|x_i \ge c|\!)_{ab}(e) & \stackrel{\text{def}}{=} & \textbf{if } (b_{i0} \ge -c) \textbf{ then } (e - \{kl \mid b_{kl} = \infty\} - \{i0\}) \\
& & \qquad \textbf{else } (e - \{kl \mid b_{kl} = \infty\}) \\
(\!|x_i - x_j \le c|\!)_{ab}(e) & \stackrel{\text{def}}{=} & \textbf{if } (b_{ji} \ge c) \textbf{ then } (e - \{kl \mid b_{kl} = \infty\} - \{ji\}) \\
& & \qquad \textbf{else } (e - \{kl \mid b_{kl} = \infty\}) \\
(\!|E \le E'|\!)_{ab}(e) & \stackrel{\text{def}}{=} & e - \{kl \mid b_{kl} = \infty\} \\
(\!|x_i := x_i + c|\!)_{ab}(e) & \stackrel{\text{def}}{=} & e - \{kl \mid b_{kl} = \infty\} \\
(\!|x_i := E|\!)_{ab}(e) & \stackrel{\text{def}}{=} & \textbf{if } (\mathsf{hasNegCycle}(a) = \mathit{true}) \\
& & \textbf{then } \mathsf{edges}(\mathsf{pickNegCycle}(a)) \\
& & \textbf{else let } e' = \big(e - \{kl \mid b_{kl} = \infty\} - \{ik, ki \mid 0 \le k \le N\}\big) \\
& & \qquad \textbf{in } \bigcup_{kl \in e'} \Big(\textbf{if } a_{kl} \le b_{kl} \textbf{ then } \{kl\} \textbf{ else } \mathsf{edges}(\mathsf{mPath}(a,k,l))\Big) \\
& & \quad \text{(where } E \text{ is either } x_j + c,\ c, \text{ or a general expression } E) \\
(\!|\texttt{skip}|\!)_{ab}(e) & \stackrel{\text{def}}{=} & e - \{kl \mid b_{kl} = \infty\}
\end{array}
$$

where edges is defined by $\mathsf{edges}(k_0 k_1 ... k_n) = \{k_0 k_1,\ k_1 k_2,\ \ldots,\ k_{n-1} k_n\}$.

Fig. 4. Back-tracers for Atomic Terms in the Zone Analysis

Back-tracers for all atomic terms induce a back-tracer for an entire program $P = (V, E, n_i, n_f, L)$. Assume that we are given back-tracers $(\!|t|\!)$ for all atomic terms t. Suppose that f and g are maps from program points of P to abstract values in $\mathcal{A}$ (i.e., $f, g \in \prod_{n \in V} \mathcal{A}$) such that g approximates the abstract one-step execution from f: $F(f) \sqsubseteq g$ for the abstract step function F for P. For such f and g, we define the *back-tracer* $(\!|P|\!)_{fg}$ *for* P to be the following function:

$$
\begin{array}{rcl}
(\!|P|\!)_{fg} & : & (\prod_{n \in V} \mathcal{E}) \to (\prod_{n \in V} \mathcal{E}) \\
(\!|P|\!)_{fg}(\epsilon)(n) & \stackrel{\text{def}}{=} & \bigsqcap \big\{ (\!|L(nm)|\!)_{f(n)g(m)}(\epsilon(m)) \mid nm \in E \big\}
\end{array}
$$

where $\prod_{n \in V} \mathcal{E}$ is the cartesian product of lattices $\mathcal{E}$, ordered pointwise. We call $\epsilon \in \prod_{n \in V} \mathcal{E}$ *extractor annotation*. The back-tracer $(\!|P|\!)_{fg}$ for P takes a post extractor annotation ϵ for g, and computes a pre extractor annotation ϵ' for f, by first running given $(\!|L(nm)|\!)_{f(n)g(m)}$, and then combining all the resulting extractors at each program node.

Recall that back-tracers $(\!|L(nm)|\!)$ for atomic terms $L(nm)$ take only those subscripts ab that satisfy $[\![L(nm)]\!]a \sqsubseteq b$. We note that when $(\!|P|\!)$ calls $(\!|L(nm)|\!)_{f(n)g(m)}$, it always uses correct subscripts; for each $nm \in E$,

$$
\begin{array}{lcll}
[\![L(nm)]\!]f(n) & \sqsubseteq & \bigsqcup\{[\![L(n'm)]\!]f(n') \mid n'm \in E\} & \text{(since } nm \in E\text{)} \\
& = & F(f)(m) & \text{(by the definition of } F\text{)} \\
& \sqsubseteq & g(m) & \text{(since } F(f) \sqsubseteq g\text{).}
\end{array}
$$

We define exall to be the application of extractor annotations:

$$\begin{array}{rcl} \mathsf{exall} & : & (\prod_{n\in V}\mathcal{E}) \to_m \big((\prod_{n\in V}\mathcal{A}) \to_m (\prod_{n\in V}\mathcal{A})\big) \\ \mathsf{exall}(\epsilon)(f) & \stackrel{\text{def}}{=} & \lambda n{\in}V.\ \mathsf{ex}(\epsilon(n))(f(n)) \end{array}$$

The following lemma shows that the back-tracer $(\!|P|\!)_{fg}$ computes a correct pre extractor annotation.

Lemma 3.5 For all ϵ in $\prod_{n\in V}\mathcal{E}$, if $(\!|P|\!)_{fg}(\epsilon) = \epsilon'$,

$$F\Big(\mathsf{exall}(\epsilon')(f)\Big) \sqsubseteq \mathsf{exall}(\epsilon)(g).$$

Proof: To show the lemma, pick an arbitrary program point n from V. Then, for all m such that $mn \in E$,

$$\begin{array}{rcll} \Big([\![L(mn)]\!]f(m)\Big) & \sqsubseteq & \Big(\bigsqcup_{m'n\in E}([\![L(m'n)]\!]f(m'))\Big) & (\text{since } mn \in E) \\ & = & \Big(F(f)(n)\Big) & (\text{by the definition of } F) \\ & \sqsubseteq & g(n) & (\text{by the assumption that } F(f) \sqsubseteq g). \end{array}$$

Let e' be $(\!|L(mn)|\!)_{f(m)g(n)}(\epsilon(n))$. By what we have derived above and the definition of back-tracers for $L(mn)$, we have that

$$[\![L(mn)]\!]\Big(\mathsf{ex}(e')(f(m))\Big) \quad \sqsubseteq \quad \mathsf{ex}(\epsilon(n))(g(n)).$$

We now prove the required inequality as follows.

$$\begin{array}{ll} & F\Big(\mathsf{exall}\ \big((\!|P|\!)_{fg}(\epsilon)\big)(f)\Big)(n) \\ = & \bigsqcup_{mn\in E}\Big([\![L(mn)]\!]\Big(\big(\mathsf{exall}\ ((\!|P|\!)_{fg}(\epsilon))(f)\big)(m)\Big)\Big) \qquad (\text{by the definition of } F) \\ = & \bigsqcup_{mn\in E}\Big([\![L(mn)]\!]\Big(\mathsf{ex}\big((\!|P|\!)_{fg}(\epsilon)(m)\big)\big(f(m)\big)\Big)\Big) \\ & \qquad (\text{since } \mathsf{exall}(\epsilon')(f)(m) = \mathsf{ex}\big(\epsilon'(m)\big)\big(f(m)\big) \text{ for all } \epsilon') \\ = & \bigsqcup_{mn\in E}\Big([\![L(mn)]\!]\Big(\mathsf{ex}\Big(\bigsqcap\big\{(\!|L(mn')|\!)_{f(m)g(n')}(\epsilon(n')) \mid mn' \in E\big\}\Big)(f(m))\Big)\Big) \\ & \qquad (\text{by the definition of } (\!|P|\!)_{fg}) \\ \sqsubseteq & \bigsqcup_{mn\in E}\Big([\![L(mn)]\!]\Big(\mathsf{ex}\Big((\!|L(mn)|\!)_{f(m)g(n)}(\epsilon(n))\Big)(f(m))\Big)\Big) \\ & \qquad (\text{since } mn \in E, \text{ and } [\![L(mn)]\!] \text{ and } \mathsf{ex}(-)(f(m)) \text{ are monotone}) \\ \sqsubseteq & \bigsqcup_{mn\in E}\mathsf{ex}(\epsilon(n))(g(n)) \\ & \qquad (\text{since } \forall e \in \mathcal{E}.\ \mathsf{ex}(e)(g(n)) \sqsupseteq [\![L(mn)]\!]\Big(\mathsf{ex}\Big((\!|L(mn)|\!)_{f(m)g(n)}(e)\Big)(f(m))\Big) \\ = & \mathsf{ex}\big(\epsilon(n)\big)\big(g(n)\big). \end{array}$$

□

3.3 Abstract-value Slicer SL

We now define an abstract-value slicer, assuming that we are given two components of the slicer, namely, an extractor domain $(\mathcal{E}, \mathsf{ex})$ and back-tracers $(\!|-|\!)$ for all atomic terms in this domain. Suppose that we are given a program $P = (V, E, n_i, n_f, L)$.

Let F be the abstract one-step execution of P in the abstract interpretation, and let $\mathsf{postfix}(F)$ be $\{f \mid F(f) \sqsubseteq f\}$, the set of post fixpoints of F.

Definition 3.6 The *abstract-value slicer* SL *for the program* P is the function defined as follows:

$$\begin{array}{rcl} \mathsf{SL} & : & (\mathsf{postfix}(F) \times \prod_{n\in V} \mathcal{E}) \to (\prod_{n\in V} \mathcal{E}) \\ \mathsf{SL}(f,\epsilon) & \stackrel{\text{def}}{=} & \textbf{let } B_f = \lambda\epsilon'.(\epsilon' \sqcap (\!|P|\!)_{ff}\epsilon') \textbf{ and} \\ & & \quad k = \min\{n \mid n \geq 0 \wedge B_f^n(\epsilon) = B_f^{n+1}(\epsilon)\} \\ & & \textbf{in } B_f^k(\epsilon) \end{array}$$

Intuitively, the first input f to the slicer denotes the result of the abstract interpretation, and the second ϵ specifies the part of f that is used for verification; although $\mathsf{exall}(\epsilon)(f)$ is weaker than f, it is still strong enough to verify the property of interest. Given such f and ϵ, the slicer SL defines a reductive function[14] B_f on $\prod_{n\in V} \mathcal{E}$, and then computes its fixpoint ϵ' such that $\epsilon' \sqsubseteq \epsilon$, by repeatedly applying B_f from ϵ. Note that SL always succeeds in computing such ϵ', because the domain $\prod_{n\in V} \mathcal{E}$ of B_f has finite height. The result of the slicer $\mathsf{SL}(f,\epsilon)$ is this computed fixpoint ϵ'.[15]

The result $\mathsf{SL}(f,\epsilon)$ of the abstract-value slicer satisfies the following two important properties, which together ensure the correctness of the slicer:

(1) $\mathsf{SL}(f,\epsilon) \sqsubseteq \epsilon$, and

(2) $\mathsf{exall}(\mathsf{SL}(f,\epsilon))(f)$ is a post fixpoint of F.

The first property means that $\mathsf{SL}(f,\epsilon)$ extracts at least as much information as ϵ, so that if a property of P can be verified by $\mathsf{exall}(\epsilon)(f)$, it can also be verified by $\mathsf{exall}(\mathsf{SL}(f,\epsilon))(f)$. The second property means that $\mathsf{exall}(\mathsf{SL}(f,\epsilon))(f)$ is another possible solution of the abstract interpretation, which could have been obtained if the abstract interpretation used a different strategy for computing post fixpoints. Note that the first property holds because B_f is reductive and the fixpoint computation of the slicer starts from ϵ. For the second property, we prove a slightly more general lemma, by using the soundness of the back-tracer $(\!|P|\!)$ (Lemma 3.5).

Lemma 3.7 For all $f \in \mathsf{postfix}(F)$ and all $\epsilon' \in \prod_{n\in V} \mathcal{E}$,

$$B_f(\epsilon') = \epsilon' \implies F(\mathsf{exall}(\epsilon')(f)) \sqsubseteq \mathsf{exall}(\epsilon')(f).$$

Proof: To show the lemma, choose an arbitrary post fixpoint f of F and an extractor annotation ϵ' for f such that $B_f(\epsilon') = \epsilon'$. Then,

$$\begin{array}{rcll} (\!|P|\!)_{ff}\epsilon' & \sqsupseteq & B_f(\epsilon') & \text{(by the definition of } B_f\text{)} \\ & = & \epsilon' & \text{(by assumption).} \end{array}$$

[14] A function f on a poset C is reductive iff $f(x) \sqsubseteq x$ for all $x \in C$.

[15] In general, the result of $\mathsf{SL}(f,\epsilon)$ is not the greatest fixpoint ϵ' of B_f satisfying the condition $\epsilon' \sqsubseteq \epsilon$. In fact, such greatest fixpoints ϵ' might not even exist. However, if $(\!|t|\!)_{ab}$ is monotone for all atomic terms t, so that $(\!|P|\!)_{fg}$ is monotone, then the result of $\mathsf{SL}(f,\epsilon)$ is the greatest fixpoint ϵ' of B_f that satisfies the condition. In this case, the result of $\mathsf{SL}(f,\epsilon)$ is the greatest fixpoint of $\lambda\epsilon'.\, \epsilon \sqcap (\!|P|\!)_{ff}(\epsilon')$.

Using what we have just shown above, we prove the required inequality as follows:

$$\begin{array}{rll} \mathsf{exall}(\epsilon')(f) & \sqsupseteq & F\big(\mathsf{exall}\big(\langle\!|P|\!\rangle_{ff}\,\epsilon'\big)(f)\big) \qquad \text{(by Lemma 3.5)} \\ & \sqsupseteq & F(\mathsf{exall}(\epsilon')(f)) \;\; \text{(since } \epsilon' \sqsubseteq \langle\!|P|\!\rangle_{ff}\,\epsilon', \text{ and } F \text{ and } \mathsf{exall} \text{ are monotone).} \end{array}$$

□

We summarize what we have just proved in the following proposition.

Proposition 3.8 (Correctness) For all $f \in \mathsf{postfix}(F)$ and all $\epsilon \in \prod_{n\in V} \mathcal{E}$, the slicer $\mathsf{SL}(f,\epsilon)$ terminates (i.e., it is a well-defined total function), and it outputs ϵ' such that $\epsilon' \sqsubseteq \epsilon$ and $F\big(\mathsf{exall}(\epsilon')(f)\big) \sqsubseteq \mathsf{exall}(\epsilon')(f)$.

Example 10 Consider the following result from the evenness analysis:

$$\boxed{\begin{array}{l} x \mapsto \top_e \\ y \mapsto \top_e \end{array}}\ \texttt{y:=2y;}\ \boxed{\begin{array}{l} x \mapsto \top_e \\ y \mapsto \mathsf{even} \end{array}}\ \texttt{x:=2y;}\ \boxed{\begin{array}{l} x \mapsto \mathsf{even} \\ y \mapsto \mathsf{even} \end{array}}\ \texttt{y:=x}\ \boxed{\begin{array}{l} x \mapsto \mathsf{even} \\ y \mapsto \mathsf{even} \end{array}}$$

Suppose that we have used the analysis in order to verify that variable `y` stores an even integer at the end. The following extractor annotation expresses this verification goal:

$$\boxed{\{\}}\ \texttt{y:=2y;}\ \boxed{\{\}}\ \texttt{x:=2y;}\ \boxed{\{\}}\ \texttt{y:=x}\ \boxed{\{y\}}$$

When the abstract-value slicer for the evenness analysis is given the above analysis result and extractor annotation, it returns the extractor annotation below:

$$\boxed{\{\}}\ \texttt{y:=2y;}\ \boxed{\{\}}\ \texttt{x:=2y;}\ \boxed{\{x\}}\ \texttt{y:=x}\ \boxed{\{y\}}$$

Thus, the original analysis result is sliced to:

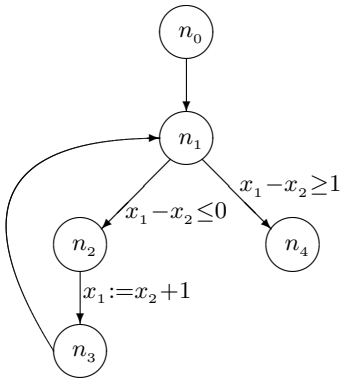
$$\boxed{\begin{array}{l} x \mapsto \top_e \\ y \mapsto \top_e \end{array}}\ \texttt{y:=2y;}\ \boxed{\begin{array}{l} x \mapsto \top_e \\ y \mapsto \top_e \end{array}}\ \texttt{x:=2y;}\ \boxed{\begin{array}{l} x \mapsto \mathsf{even} \\ y \mapsto \top_e \end{array}}\ \texttt{y:=x}\ \boxed{\begin{array}{l} x \mapsto \top_e \\ y \mapsto \mathsf{even} \end{array}}$$

Note that the sliced result correctly expresses that the only necessary information for the verification is the evenness of `x` after `x:=2y` and the verification goal at the end. □

Example 11 Figure 5 shows the result of the abstract-value slicing for zone analysis. Figure 5(b) shows the input to the slicer; the DBMs in the second row describe the result of zone analysis, and the extractors in the first row specify that only the $(2,1),(1,2)$-th entries of the DBM at n_4 are used for verification. Figure 5(c) shows the sliced result; the first row describes the result of the abstract-value slicer for this input, and the second row describes the application of the obtained extractors to the abstract interpretation result. For this example, this table indicates that among 19 non-∞ DBM entries in the abstract interpretation result, only 5 entries are needed to prove the property of interest. Finally, Figure 5(d) expresses the abstract interpretation result and its slice in the form of constraints. □

4. METHODS FOR DESIGNING BACK-TRACERS FOR ATOMIC TERMS

In this section, we provide two methods for designing back-tracers for atomic terms. As explained in Section 3.2, it is relatively easy to define a *correct* back-tracer for an




```
n0
n1      while (x1 - x2 <= 0) do
n2        x1 := x2 + 1
n3      od
n4
```

(a) Program

	n_0	n_1	n_2	n_3	n_4
Initial extractor annotation	{}	{}	{}	{}	{(1,2), (2,1)}
Abstract interpretation result	$\begin{array}{c\|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & 4 & 3 \\ x_1 & -1 & \infty & \infty \\ x_2 & -1 & 0 & \infty \end{array}$	$\begin{array}{c\|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & 3 \\ x_1 & \infty & \infty & \infty \\ x_2 & -1 & 1 & \infty \end{array}$	$\begin{array}{c\|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & 3 \\ x_1 & \infty & \infty & \infty \\ x_2 & -1 & 0 & \infty \end{array}$	$\begin{array}{c\|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & 3 \\ x_1 & \infty & \infty & -1 \\ x_2 & -1 & 1 & \infty \end{array}$	$\begin{array}{c\|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & 3 \\ x_1 & \infty & \infty & -1 \\ x_2 & -1 & 1 & \infty \end{array}$

(b) Input Arguments for Abstract-value Slicer

	n_0	n_1	n_2	n_3	n_4
Result extractors from abstract-value slicer	{(2,1)}	{(2,1)}	{}	{(2,1)}	{(1,2), (2,1)}
Sliced abstract interpretation result	$\begin{array}{c\|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & \infty \\ x_1 & \infty & \infty & \infty \\ x_2 & \infty & 0 & \infty \end{array}$	$\begin{array}{c\|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & \infty \\ x_1 & \infty & \infty & \infty \\ x_2 & \infty & 1 & \infty \end{array}$	$\begin{array}{c\|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & \infty \\ x_1 & \infty & \infty & \infty \\ x_2 & \infty & \infty & \infty \end{array}$	$\begin{array}{c\|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & \infty \\ x_1 & \infty & \infty & \infty \\ x_2 & \infty & 1 & \infty \end{array}$	$\begin{array}{c\|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & \infty & \infty & \infty \\ x_1 & \infty & \infty & -1 \\ x_2 & \infty & 1 & \infty \end{array}$

(c) Results from Abstract-value Slicer

	n_0	n_1	n_2	n_3	n_4
Before slicing	$1 \leq x_1 \leq 4 \wedge 1 \leq x_2 \leq 3 \wedge x_1 \leq x_2$	$1 \leq x_2 \leq 3 \wedge x_1 \leq x_2 + 1$	$1 \leq x_2 \leq 3 \wedge x_1 \leq x_2$	$1 \leq x_2 \leq 3 \wedge x_1 = x_2 + 1$	$1 \leq x_2 \leq 3 \wedge x_1 = x_2 + 1$
After slicing	$x_1 \leq x_2$	$x_1 \leq x_2 + 1$		$x_1 \leq x_2 + 1$	$x_1 = x_2 + 1$

(d) Results as Constraints

Fig. 5. Result of Abstract-value Slicing for the Program in Figure 2

atomic term t. However, designing a *good* back-tracer for t is difficult, and requires special knowledge about the abstract interpretation that is used. The first method in the section aims at producing *accurate* back-tracers for atomic terms: a slicer with the back-tracer that is produced usually filters out more information from the abstract interpretation result, than the one with naively-designed back-tracers. The second method, on the other hand, aims at a back-tracer with low cost on time and space. Throughout the section, we assume a fixed abstract interpretation that uses

an abstract domain $(\mathcal{A}, \sqsubseteq, \bot, \sqcup)$ and an abstract semantics $[\![-]\!]$ for atomic terms. We also assume that a fixed extractor domain $(\mathcal{E}, \mathsf{ex})$ is given for this abstract interpretation, and that $\mathcal{E}$ is finite.

4.1 Best Back-tracer Construction

The first method constructs the *best* back-tracer for each atomic term, considered in Proposition 3.4. It is a slight modification of a rather well-known "reversing technique" [Hughes and Launchbury 1992; Duesterwald et al. 1995]. Let t be an atomic term, and let (a, b) be a pair of pre and post conditions in $\mathsf{prepost}(t)$. For these t, a, b, the method defines the back-tracer $(\!|t|\!)_{ab}$ as follows:

$$(\!|t|\!)_{ab}(e) \stackrel{\text{def}}{=} \bigsqcup\{e_0 \in \mathcal{E} \mid [\![t]\!](\mathsf{ex}(e_0)(a)) \sqsubseteq \mathsf{ex}(e)(b)\}.$$

Intuitively, $(\!|t|\!)_{ab}(e)$ is the "conjunction" of all correct pre extractors: $(\!|t|\!)_{ab}(e)$ selects some information from a, precisely when all the correct pre extractors select the same information. Note that for every correct pre extractor e_0, the computed extractor $e' = (\!|t|\!)_{ab}(e)$ filters out at least as much information from a as e_0 (i.e., $\mathsf{ex}(e_0)(a) \sqsubseteq \mathsf{ex}(e')(a)$), and so, it induces a better slice of a than e_0.

In addition to the bestness of the method, we need to check whether the method constructs a correct back-tracer. Unfortunately, this method does not always construct a correct back-tracer; in general, the constructed $(\!|t|\!)_{ab}$ does not satisfy the following soundness condition from the definition of back-tracers:

$$\forall e, e' \in \mathcal{E}.\ (\!|t|\!)_{ab}(e) = e' \implies [\![t]\!]\Big(\mathsf{ex}(e')(a)\Big) \sqsubseteq \mathsf{ex}(e)(b).$$

To make the above soundness condition hold, we should restrict the use of the method for join-preserving functions as the solutions in other studies [Hughes and Launchbury 1992; Duesterwald et al. 1995]: we use the constructed $(\!|t|\!)_{ab}$, only when $[\![t]\!]$ and $\mathsf{ex}(-)(a)$ preserve finite joins.

Lemma 4.1 If $[\![t]\!]$ and $\lambda e.\, \mathsf{ex}(e)(a)$ preserve finite joins for all a, then $(\!|t|\!)$ satisfies the soundness condition for back-tracers for t. Moreover, in this case, $(\!|t|\!)_{ab}$ is the best back-tracer for t (which exists by Proposition 3.4).

Proof: In this proof, we show that the method constructs a correct back-tracer by the join-preservation, and then show why it is a best back-tracer. To see why the join-preservation provides a solution, suppose that $[\![t]\!]$ and $\mathsf{ex}(-)(a)$ preserves finite joins. Then, their composition $\lambda e.\, [\![t]\!]\big(\mathsf{ex}(e)(a)\big)$ also preserves finite joins. So, for all extractors $e, e' \in \mathcal{E}$ such that $(\!|t|\!)_{ab} e = e'$, we have that

$$\begin{aligned}
&[\![t]\!]\Big(\mathsf{ex}(e')(a)\Big) \\
&= [\![t]\!]\Big(\mathsf{ex}\big((\!|t|\!)_{ab}(e)\big)(a)\Big) \\
&= [\![t]\!]\Big(\mathsf{ex}\Big(\bigsqcup\{e_0 \mid [\![t]\!]\big(\mathsf{ex}(e_0)(a)\big) \sqsubseteq \mathsf{ex}(e)(b)\}\Big)(a)\Big) \qquad \text{(by the definition of } (\!|t|\!)_{ab}\text{)} \\
&= \Big(\bigsqcup\{[\![t]\!]\big(\mathsf{ex}(e_0)(a)\big) \mid [\![t]\!]\big(\mathsf{ex}(e_0)(a)\big) \sqsubseteq \mathsf{ex}(e)(b)\}\Big) \qquad \text{(by join preservation)} \\
&\sqsubseteq \mathsf{ex}(e)(b).
\end{aligned}$$

This order relationship implies the correctness of $(\!|t|\!)_{ab}$. Now, we prove that $(\!|t|\!)_{ab}$ is a best back-tracer. Consider a back-tracer k for t. Then, for all $(a, b) \in \mathsf{prepost}(t)$ and $e \in \mathcal{E}$,

$$[\![t]\!]\Big(\mathsf{ex}\Big(k_{ab}(e)\Big)(a)\Big) \quad \sqsubseteq \quad \mathsf{ex}(e)(b).$$

Thus, by the definition of $(\!|t|\!)$, we have that $k_{ab}(e) \sqsubseteq (\!|t|\!)_{ab}(e)$, as required. □

The very definition of $(\!|t|\!)_{ab}$ gives a default (usually inefficient) implementation, if all the extractor applications are computable. When a post extractor e for b is given, the implementation calculates all the correct pre extractors for a, b, t, e; this is possible, because the extractor domain $\mathcal{E}$ is finite (by assumption) and $[\![t]\!]$ is computable. Then, the implementation returns the greatest one from the calculated extractors. This default implementation is, however, very slow, and in many cases, it can be improved dramatically. We illustrate this improvement using zone analysis.

Example 12 Consider zone analysis $(\mathcal{M}, \sqsubseteq, \bot, \sqcup)$ and the extractor domain $(\mathcal{E}, \mathsf{ex})$ in Example 7. In this case, the method in this section can be applied to obtain the best back-tracer for an atomic term, if the term is either a boolean expression or an assignment of the form $x_i := x_i + c$; zone analysis interprets all such atomic terms as join-preserving functions, and extractor application $\mathsf{ex}(-)(a)$ preserves finite joins for all a. In this example, we will explain how to efficiently implement this best back-tracer.

Let t be an atomic term that is either a boolean expression or an assignment of the form $x_i := x_i + c$. Our implementation of the best back-tracer for t is based on two important observations.

(1) First, no matter whether t is a boolean expression or an assignment, the abstract semantics $[\![t]\!]$ of t is a pointwise transformation of DBM matrices; to compute the ij-th entry of the output DBM, $[\![t]\!]$ uses at most the ij-th entry of the input DBM. More precisely, there exists a family $\{f_{ij}\}_{ij \in (N+1)\times(N+1)}$ of monotone functions on $\mathsf{Ints} \cup \{-\infty, \infty\}$ (ordered by $\leq$) such that

$$\forall ij. \quad ([\![t]\!]a)_{ij} = f_{ij}(a_{ij}).$$

(2) Second, when a function family $\{f_{ij}\}_{ij}$ determines $[\![t]\!]$, it can be used to simplify the "correctness condition" for pre and post extractors: for every $(a, b) \in \mathsf{prepost}(t)$, pre extractor $e_0 \in \mathcal{E}$ and post extractor $e \in \mathcal{E}$, we have that

$$\Big([\![t]\!](\mathsf{ex}(e_0)(a)) \sqsubseteq \mathsf{ex}(e)(b)\Big) \iff \Big(\forall ij.\ ij \in e \Rightarrow (ij \in e_0 \ \vee \ f_{ij}(\infty) \leq b_{ij})\Big).$$

Intuitively, this simplified condition says that every index ij in the post extractor e should belong to the pre extractor e_0, except when t can "generate" the information b_{ij} (i.e., $x_j - x_i \leq b_{ij}$) without using the input DBM. To see this, note that since f_{ij} is monotone, $f_{ij}(\infty) \leq b_{ij}$ implies that for every input DBM a', the ij-th entry of $[\![t]\!](a')$ should be less than or equal to b_{ij}. Thus, no information from the input DBM is necessary for t to "produce" the ij-th entry of b.

We now use these two observations to optimize the best back-tracer for t:

$$
\begin{aligned}
&\bigsqcup\{e_0 \mid [\![t]\!](\mathsf{ex}(e_0)(a)) \sqsubseteq \mathsf{ex}(e)(b)\} \\
&= \bigcap\{e_0 \mid [\![t]\!](\mathsf{ex}(e_0)(a)) \sqsubseteq \mathsf{ex}(e)(b)\} && \text{(by the lattice structure of } \mathcal{E}) \\
&= \bigcap\{e_0 \mid \forall ij.\ ij \in e \Rightarrow (ij \in e_0 \ \vee\ f_{ij}(\infty) \leq b_{ij})\} && \text{(by the second observation)} \\
&= \bigcap\{e_0 \mid \forall ij.\ (ij \in e \wedge f_{ij}(\infty) \not\leq b_{ij}) \Rightarrow ij \in e_0\} \\
&= \bigcap\big\{e_0 \mid \{ij \mid ij \in e \wedge f_{ij}(\infty) \not\leq b_{ij}\} \ \subseteq\ e_0\big\} \\
&= \{ij \mid ij \in e \wedge f_{ij}(\infty) \not\leq b_{ij}\} \\
&= e - \{ij \mid f_{ij}(\infty) \leq b_{ij}\}
\end{aligned}
$$

Note that the obtained formula indicates the efficient implementation of the best back-tracer $(\!|f|\!)_{ab}$ as a single set subtraction. Moreover, the subtracted set in the formula is a fixed set that does not depend on the post extractor e. This property can allow a further optimization of the set subtraction. In fact, the back-tracers for boolean expressions $E \leq E'$, assignments $x_i := x_i + c$ and command `skip` in Figure 4 are such further optimizations. □

Before finishing the discussion on the construction of best back-tracers, we consider one special case that the extractor domain is an atomic lattice. Recall (from the standard lattice theory [Davey and Priestley 1990]) that an element x in a lattice L is an *atom* if and only if it is the second smallest element in L:

$$x \neq \bot \wedge \Big(\forall x' \in L.\, (x' \sqsubseteq x \wedge x' \neq x) \Rightarrow x' = \bot)\Big),$$

and that a lattice L is *atomic* if and only if every element x in the lattice L can be reconstructed by combining (by join) all the atoms x' such that $x' \sqsubseteq x$:

$$\forall x \in L.\ x = \bigsqcup\{x' \mid x' \sqsubseteq x \text{ and } x' \text{ is an atom}\}.$$

The following proposition suggests that we can optimize the best back-tracer $(\!|t|\!)_{ab}$ when $\mathcal{E}$ is atomic.

Proposition 4.2 When $\mathcal{E}$ is atomic, the best back-tracer $(\!|t|\!)_{ab}$ is identical to the function below:

$$\lambda e.\ \bigsqcup\{e_0 \in \mathcal{E} \mid [\![t]\!]\big(\mathsf{ex}(e_0)(a)\big) \sqsubseteq \mathsf{ex}(e)(b) \text{ and } e_0 \text{ is an atom}\}.$$

Proof: Let k_{ab} be the function defined in the proposition. For all $e_0 \in \{e_0 \in \mathcal{E} \mid [\![t]\!]\big(\mathsf{ex}(e_0)(a)\big) \sqsubseteq \mathsf{ex}(e)(b)$ and e_0 is an atom$\}$, it follows $e_0 \sqsubseteq (\!|t|\!)_{ab}(e)$ by Lemma 3.3. Since $k_{ab}(e)$ is the join of all such e_0, we have that $k_{ab}(e) \sqsubseteq (\!|t|\!)_{ab}(e)$. For the other direction, consider an arbitrary $e_1 \in \mathcal{E}$ such that

$$[\![t]\!]\big(\mathsf{ex}(e_1)(a)\big) \ \sqsubseteq\ \mathsf{ex}(e)(b).$$

Note that such e_1 can be $(\!|t|\!)_{ab}$ by the definition (Def. 3.1) of back-tracer. For all atoms e_0 such that $e_0 \sqsubseteq e_1$,

$$
\begin{array}{lll}
[\![t]\!]\big(\mathsf{ex}(e_0)(a)\big) & \sqsubseteq & [\![t]\!]\big(\mathsf{ex}(e_1)(a)\big) \qquad \text{(by the monotonicity of } [\![f]\!] \text{ and } \mathsf{ex}(-)(a)) \\
& \sqsubseteq & \mathsf{ex}(e)(b) \qquad\qquad\qquad\qquad\qquad\qquad \text{(by the choice of } e_1).
\end{array}
$$

Thus, $e_0 \sqsubseteq k_{ab}(e)$. This implies that $e_1 \sqsubseteq k_{ab}(e)$ since $\mathcal{E}$ is atomic. So, $(\!|t|\!)_{ab}(e) \sqsubseteq k_{ab}(e)$. □

4.2 Extension Method

The second method, called *extension method*, is a dual approach to the atomic lattice case considered in the previous section. Extension method assumes two properties of the extractor domain. Recall (from the standard lattice theory [Davey and Priestley 1990]) that an element x in a lattice L is a *dual atom* if and only if it is the second biggest element in L:

$$x \neq \top \wedge \Big(\forall x' \in L.\, (x \sqsubseteq x' \wedge x' \neq x) \Rightarrow x' = \top\Big),$$

and that a lattice L is *dual atomic* if and only if every element x in the lattice L can be reconstructed by combining (by meet) all the dual atoms x' such that $x \sqsubseteq x'$:

$$\forall x \in L.\, x = \bigsqcap\{x' \mid x \sqsubseteq x' \text{ and } x' \text{ is a dual atom}\}.$$

The first assumption of the extension method is that each extractor lattice $\mathcal{E}$ is dual atomic, and the second assumption is that each extractor application preserves all finite meets:

$$\mathsf{ex}(\top) = \top \;\text{ and }\; \mathsf{ex}(e \sqcap e') = \mathsf{ex}(e) \sqcap \mathsf{ex}(e').$$

Here $\top$ and $\sqcap$ are the lattice operations for upper closure operators, and they are defined pointwise.[16] Intuitively, these two assumptions mean that every extractor e in $\mathcal{E}$ represents a collection $\{e_1, \ldots, e_n\}$ of dual atomic extractors; e extracts information from a by first applying each e_i to a and then conjoining all the resulting information:

$$\mathsf{ex}(e)(a) = \bigsqcap_{i=1,\ldots,n} \mathsf{ex}(e_i)(a).$$

When the extractor domain satisfies the above assumptions, the extension method provides a recipe for constructing correct back-tracers for all atomic terms. Let t be an atomic term and (a, b) a pair of pre and post conditions in $\mathsf{prepost}(t)$. The first step of the extension method is to define a *partial back-tracer* g: g is a partial function of type $\mathcal{E} \rightharpoonup \mathcal{E}$ such that (1) the domain of g is precisely the set of dual atoms in $\mathcal{E}$ and (2) for all post extractors in $\mathsf{dom}(g)$, g calculates correct pre extractors:

$$\forall e \in \mathsf{dom}(g).\, [\![t]\!]\Big(\mathsf{ex}\big(g(e)\big)(a)\Big) \sqsubseteq \mathsf{ex}(e)(b).$$

The next step is to extend g to the following complete back-tracer:

$$(\!|t|\!)_{ab}(e) \stackrel{\text{def}}{=} \bigsqcap\{g(e_1) \mid e \sqsubseteq e_1 \;\text{ and }\; e_1 \text{ is a dual atom}\}.$$

The total back-tracer $(\!|t|\!)_{ab}(e)$ decomposes the post extractor e into dual atoms, and then applies g to all the obtained dual atoms; finally, it merges all the resulting pre extractors (by meet). Note that, since $\mathcal{E}$ is finite, the meet here is over the finite sets, and so, it is well-defined. The following lemma shows that the constructed $(\!|t|\!)_{ab}$ is correct.

Lemma 4.3 For all extractors $e, e' \in \mathcal{E}$, we have that

$$e' = (\!|t|\!)_{ab}(e) \quad\Longrightarrow\quad [\![t]\!]\Big(\mathsf{ex}(e')(a)\Big) \;\sqsubseteq\; \mathsf{ex}(e)(b).$$

[16] Precisely, $\top(a) = \top$ and $(\mathsf{ex}(e) \sqcap \mathsf{ex}(e'))(a) = \mathsf{ex}(e)(a) \sqcap \mathsf{ex}(e')(a)$.

Proof: We prove the lemma as follows:

$$
\begin{aligned}
&[\![t]\!]\Big(\mathsf{ex}\big((\!|t|\!)_{ab}\, e\big)(a)\Big)\\
&= [\![t]\!]\Big(\mathsf{ex}\Big(\bigsqcap\{g(e_1) \mid e \sqsubseteq e_1 \text{ and } e_1 \text{ is a dual atom}\}\Big)(a)\Big)\\
&\qquad\qquad \text{(by the definition of } (\!|t|\!)_{ab})\\
&= [\![t]\!]\Big(\bigsqcap\{\mathsf{ex}\big(g(e_1)\big)(a) \mid e \sqsubseteq e_1 \text{ and } e_1 \text{ is a dual atom}\}\Big)\\
&\qquad\qquad \text{(by the meet preservation of } \mathsf{ex})\\
&\sqsubseteq \bigsqcap\{[\![t]\!]\Big(\mathsf{ex}\big(g(e_1)\big)(a)\Big) \mid e \sqsubseteq e_1 \text{ and } e_1 \text{ is a dual atom}\} \quad \text{(since } [\![t]\!] \text{ is monotone)}\\
&\sqsubseteq \bigsqcap\{\mathsf{ex}(e_1)(b) \mid e \sqsubseteq e_1 \text{ and } e_1 \text{ is a dual atom}\}\\
&\qquad\qquad \text{(since } [\![t]\!]\big(\mathsf{ex}(g(e_1))(a)\big) \sqsubseteq \mathsf{ex}(e_1)(b) \text{ for all dual atoms } e_1)\\
&= \mathsf{ex}\Big(\bigsqcap\{e_1 \mid e \sqsubseteq e_1 \text{ and } e_1 \text{ is a dual atom}\}\Big)(b)\\
&\qquad\qquad \text{(by the meet preservation of } \mathsf{ex})\\
&= \mathsf{ex}(e)(b) \qquad\qquad \text{(since } \mathcal{E} \text{ is dual atomic).}
\end{aligned}
$$

□

The extension method has two advantages over the best back-tracer construction. First, the extension method usually provides a relatively efficient default implementation of the defined back-tracers. By "efficient", we do not mean a linear-time algorithm, but simply mean a polynomial-time algorithm, instead of exponential-time algorithm. Suppose that we have defined a back-tracer $(\!|t|\!)_{ab}$, by applying the extension method to a partial back-tracer g. The default implementation of $(\!|t|\!)_{ab}$ uses an internal table T that records the graph of function g, and is defined as follows:

```
(* e is an input (a post extractor),
   e_0 is an output (a pre extractor) *)
X := {e_1 | e ⊑ e_1 and e_1 is a dual atom};
e_0 := ⊤;
for each e_1 ∈ X
do
    lookup e'_1 s.t. (e_1, e'_1) ∈ T;
    e_0 := e_0 ⊓ e'_1
od
```

Here we specify the implementation imperatively, in order to distinguish it from the definition of $(\!|t|\!)_{ab}$. Note that this implementation simply follows the definition of $(\!|t|\!)_{ab}$ without any special optimizations. However, the implementation is efficient. In the worst case, the set X contains all the dual atoms, and so, all the basic operations in the implementation, such as the look-up of table T and the meet operation of extractors, are executed at most as many times as the number of dual atoms in $\mathcal{E}$. Even when the extractor lattice $\mathcal{E}$ is big, they contain only smaller number of dual atoms; in many cases, even though the cardinality of an extractor lattice is exponential in the size of the program, the number of dual atoms in the

lattice is polynomial in the program size. Thus, in such cases, the implementation runs in polynomial time. Note that in the best back-tracer construction, a naive implementation, which directly follows the definition, executes basic operations as many times as the size of the extractor lattice.

Second, the extension method is more widely applicable than the best back-tracer construction. The assumptions of the extension method are only about the extractor domain, not about the abstract interpretation. Thus, once the extractor domain is well-chosen, the method can be used to get back-tracers for all atomic terms. This contrasts with the best back-tracer construction, which can be applicable to an atomic term t only when $[\![t]\!]$ preserves finite joins.

Example 13 For zone analysis and the extractor domain $(\mathcal{E}, \mathsf{ex})$ in Example 7, the extension method can be applied to all atomic terms; $\mathcal{E}$ is a dual atomic lattice, whose dual atoms are singleton index sets $\{ij\}$, and the extractor application ex preserves finite meets. Here we apply the method to construct back-tracers for assignments $x_i := E$ that were not handled in Example 12, namely those that are not of the form $x_i := x_i + c$. In fact, for such assignments $x_i := E$, we cannot apply the best back-tracer construction, because $[\![x_i := E]\!]$ does not preserve some finite joins.

Let a, b be DBMs such that $[\![x_i := E]\!]a \sqsubseteq b$. To apply the extension method, we need to define a correct partial back-tracer g for $x_i := E$ at a and b. We define such g by doing the case analysis on the input DBM a. When a contains a cycle with negative weight, we pick one such cycle $\mathsf{pickNegCycle}(a) = k_0 k_1 \ldots k_n$ in a, and define g to be a constant function $\lambda e.\{k_0k_1, k_1k_2, \ldots, k_{n-1}k_n\}$. Otherwise, i.e., when a does not contain a negative cycle, we define g as follows, using paths with minimum weight:

$$g(\{kl\}) \stackrel{\text{def}}{=} \begin{array}{l} \textbf{if } (b_{kl} = \infty \ \vee \ k = i \ \vee \ l = i) \\ \textbf{then } \{\} \\ \textbf{else } \Big(\textbf{if } a_{kl} \leq b_{kl} \textbf{ then } \{kl\} \textbf{ else } \mathsf{edges}(\mathsf{mPath}(a, k, l))\Big) \end{array} .$$

Here we used the subroutine edges in Figure 4, which takes a path $k_0k_1...k_n$ and returns the set $\{k_0k_1, k_1k_2, \ldots, k_{n-1}k_n\}$ of all the edges in the path. The defined function g first checks whether $[\![t]\!]$ needs to use the input to generate b_{kl}. If so, g returns the entries of the input a that are necessary for generating b_{kl}. Otherwise, g returns the empty set.

Now the extension method gives the back-tracer $(\!|x_i := E|\!)_{ab}$ defined as follows:

$$(\!|x_i := E|\!)_{ab}(e) \stackrel{\text{def}}{=} \bigsqcap \{g(e_1) \mid e \sqsubseteq e_1 \ \text{ and } \ e_1 \text{ is a dual atom}\}$$

This definition can be optimized to the back-tracer for $x_i := E$ in Figure 4. When a contains a negative cycle,

$$\begin{array}{ll} & (\!|x_i := E|\!)_{ab}(e) \\ = & \bigsqcap \{g(e_1) \mid e \sqsubseteq e_1 \ \text{ and } \ e_1 \text{ is a dual atom}\} \\ = & \bigsqcap \{\mathsf{edges}(\mathsf{pickNegCycle}(a)) \mid e \sqsubseteq e_1 \ \text{ and } \ e_1 \text{ is a dual atom}\} \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by the definition of } g) \\ = & \mathsf{edges}(\mathsf{pickNegCycle}(a)). \end{array}$$

When a does not contain a negative cycle,

$$
\begin{aligned}
&(\!|x_i := E|\!)_{ab}(e)\\
&= \bigsqcap\{g(e_1) \mid e \sqsubseteq e_1 \text{ and } e_1 \text{ is a dual atom}\}\\
&= \bigcup\{g(e_1) \mid e \sqsubseteq e_1 \text{ and } e_1 \text{ is a dual atom}\} \qquad \text{(by the lattice structure of } \mathcal{E})\\
&= \bigcup_{kl\in e} g(\{kl\})\\
&\qquad \text{(since } (e_1 = \{kl\} \text{ for some } kl \in e) \iff (e \sqsubseteq e_1 \text{ and } e_1 \text{ is a dual atom}))\\
&= \bigcup_{kl\in e} \textbf{if } (b_{kl}{=}\infty \vee i{=}k \vee i{=}l)\\
&\qquad\quad \textbf{then } \{\}\\
&\qquad\quad \textbf{else } \Big(\textbf{if } a_{kl} \le b_{kl} \textbf{ then } \{kl\} \textbf{ else } \mathsf{edges}(\mathsf{mPath}(a,k,l))\Big)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{(by the definition of } g)\\
&= \textbf{let } e' = e - \{kl \mid b_{kl} = \infty \vee k = i \vee l = i\}\\
&\quad \textbf{in } \bigcup_{kl\in e'} \textbf{if } a_{kl} \le b_{kl} \textbf{ then } \{kl\} \textbf{ else } \mathsf{edges}(\mathsf{mPath}(a,k,l))\\
&= \textbf{let } e' = e - \{kl \mid b_{kl} = \infty\} - \{ik, ki \mid 0 \le k \le N\} \qquad .\\
&\quad \textbf{in } \bigcup_{kl\in e'} \textbf{if } a_{kl} \le b_{kl} \textbf{ then } \{kl\} \textbf{ else } \mathsf{edges}(\mathsf{mPath}(a,k,l))
\end{aligned}
$$

Note that in both cases, the optimized definition coincides with the back-tracer for $(\!|x_k := E|\!)$ in Figure 4. □

5. EXPERIMENTS

We designed an abstract-value slicer for the full zone analysis [Miné 2001], and tested the efficiency of the resulting slicer in the context of proof generation.

5.1 Abstract-value Slicer for the Full Zone Analysis

The full zone analysis is different from our simplified version in Example 5 in two aspects. First, the full analysis additionally applies the DBM closure operator $-^*$ (in Example 5) before all DBM joins in the analysis. Second, it has better abstract semantics of atomic terms. For all the assignments $x_i := E$, if E does not have the form c, $x_i + c$ or $x_j + c$, our simplified analysis replaces $x_i := E$ by a random assignment $x_i :=?$, which chooses an integer nondeterministically and assigns the chosen number to x_i; then, the simplified analysis defines $[\![x_i := E]\!]$ to be the strongest postcondition transformer of $x_i :=?$. The full version, on the other hand, does not do such a replacement, but defines more accurate abstract semantics of $x_i := E$ using interval analysis. Given an input DBM a, the full analysis first applies the closure $-^*$ to a, just like the simplified analysis. But then, instead of updating all x_i-related entries by ∞, the full analysis estimates the range of the right hand side expression E of $x_i := E$, using interval analysis. It projects a^* into the following abstract value $\mathsf{prj}(a^*)$ in interval analysis

$$\mathsf{prj}(a^*) = \lambda x_j.\, [-(a^*)_{j0}, (a^*)_{0j}],$$

and runs $[\![E]\!](\mathsf{prj}(a^*))$ in interval analysis to obtain the (approximate) range $[n, m]$ of E. Finally, using this obtained range of E, the full analysis updates the $i0$ and $0i$ entries of the input a^*, and returns the following DBM:

$$\Big((a^*)[ki{\mapsto}\infty,\ ik{\mapsto}\infty]_{1\le k\neq i\le N}\Big)[0i{\mapsto}m,\ i0{\mapsto}{-n}].$$

program	number of DBM entries (1)total[a]	extracted[b]	(2)removed[c]	(2)/(1)	slicing time
Insertionsort	92	22	70	76%	0.07
Partition[d]	120	46	74	62%	0.03
Bubblesort	217	42	175	81%	0.11
KMP[e]	463	133	330	72%	0.28
Heapsort	817	181	636	78%	0.29

[a]number of non-∞ DBM entries in the results of zone analysis.
[b]number of the DBM entries in (1) that are extracted (i.e., not changed) by the slicer.
[c]number of the DBM entries in (1) that are changed to ∞ by the slicer.
[d]Partition function in Quicksort.
[e]Knuth-Morris-Pratt pattern matching algorithm.

Table I. Number of Sliced DBM Entries

We designed an abstract-value slicer for the full zone analysis, by modifying the slicer for the simplified version in Example 7 and 9. To deal with the additional uses of the closure operator $-^*$, we defined the back-tracer β for $-^*$ as follows. For all a, b such that $a^* \sqsubseteq b$,

$$\begin{array}{rcl} \beta_{ab} & : & \mathcal{E} \to \mathcal{E} \\ \beta_{ab}(e) & \stackrel{\text{def}}{=} & \textbf{if } (\mathsf{hasNegCycle}(a) = \mathit{true}) \\ & & \textbf{then } \mathsf{edges}(\mathsf{pickNegCycle}(a)) \\ & & \textbf{else let } e' = e - \{kl \mid b_{kl} = \infty\} \\ & & \quad \textbf{in } \bigcup_{kl \in e'} \Big(\textbf{if } a_{kl} \leq b_{kl} \textbf{ then } \{kl\} \textbf{ else } \mathsf{edges}(\mathsf{mPath}(a,k,l)) \Big) \end{array}$$

The defined β was, then, inserted into the old slicer in Example 9, in order to back-trace newly added closure applications in the full analysis. To handle the modified abstract semantics $[\![x := E]\!]$, we designed a slicer for interval analysis, which contains a back-tracer $(\!|E|\!)$ for expression E. Then, using $(\!|E|\!)$, we changed the old back-tracer $(\!|x := E|\!)$, in order to account for the new improved abstract semantics of $x := E$.

5.2 Experimental Results

We implemented the full zone analysis, the abstract-value slicer, and the proof construction algorithm using our previous work [Seo et al. 2003]. In our experiment, we first executed the analysis 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 the number of invariants (i.e., DBM entries) in the results that have been eliminated (i.e., replaced by ∞). Finally, we applied the proof construction algorithm to both the original abstract interpretation results and their sliced versions, and measured how much the slicer reduced the size of the constructed proofs.

Table I shows the number of invariants that have been sliced out by the abstract-value slicer. The second column, labeled by "total", contains the number of all the nontrivial DBM entries (i.e., entries that are not ∞) in the result of the abstract interpreter, and the fourth column, labeled by "removed", shows how many of

program	before slicing		after slicing		(1)-(3)/(1)	reduction in
	(1)FOL[a]	(2)formulas[b]	(3)FOL[c]	(4)formulas[d]		proof size[e]
Insertionsort	248	2530	166	1122	33%	53%
Partition	398	3866	201	1847	49%	52%
Bubblesort	894	12230	389	2677	56%	76%
KMP	1364	26898	653	7683	52%	70%
Heapsort	2542	52370	1028	7936	60%	84%

[a]number of nodes for first-order logic rules that appear in the proof tree for an original (unsliced) analysis result.
[b]number of first-order formulas that appear in the proof tree for the original analysis result.
[c]number of nodes for first-order logic rules that appear in the proof tree for a sliced analysis result.
[d]number of formulas that appear in the proof tree for the sliced analysis result.
[e]Here the size of a proof counts all of applied Hoare logic rules, applied first-order logic rules, and first-order logic formulas in the proof.

Table II. Reduction in the Proof Size

those nontrivial entries the slicer found unnecessary for verifying the absence of array bounds errors. The experimental result shows that about 62% to 81% of computed invariants are not needed for the verification.

The reduction in the size of constructed proofs is shown in Table II. The constructed proofs are trees whose nodes express the application of Hoare logic rule or first-order logic rule. The nodes for first-order logic rules have different sizes, depending on the first-order logic formulas that are contained in the nodes. Thus, for each constructed proof, we counted three entities: the nodes for Hoare logic rules, the nodes for first-order logic rules, and the first-order formulas. The abstract-value slicer did not reduce the number of Hoare logic rules, because Hoare rules are applied as many times as the number of program constructs in the program, and the abstract-value slicer does not change the program. However, the slicer reduced the number of first-order logic rules and the number of first-order formulas. In Table II, we show those numbers before and after slicing. The experimental result shows that in the proof trees for sliced analysis results, about 33% to 60% less rules are used for showing implications between first-order logic formulas. In the seventh column of the table, we show the reduction ratio in the size of the whole proofs. For each of the constructed proof trees, we add the number of the nodes and that of first-order formulas, and then, we compute the reduction ratio in this number. The experimental result shows that the proof trees for sliced analysis results are about 52% to 84% smaller than those for original analysis results.

6. CONCLUSION

In this paper, we have presented a framework for abstract-value slicers that weaken the abstract interpretation results. We have presented two design guides to define back-tracers for atomic terms that propagate the slicing information of each atomic term backwards. In fact, designing a back-tracer is a key task in implementing an abstract-value slicer.

The motivating application of the slicer is to reduce the proof size in the proof construction method [Seo et al. 2003] that 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 enables us to have smaller proofs. In our experiment in constructing the proofs for the absence of array bound violations in five small yet representative array-access programs, our slicing algorithm reduce the proofs' sizes. In our experiment with the zone analysis, the slicer identified $62\% - 81\%$ of the abstract interpretation results as unnecessary, and resulted in $52\% - 84\%$ reduction in the proof size.

Our abstract-value slicer has been targeted for one specific application, the construction of Hoare proofs from abstract interpretation results. For instance, our slicer guarantees that the sliced analysis results are post fixpoints of the abstract transfer function. Because of this guarantee, the following proof-construction phase does not have to call a (possibly expensive) theorem prover, but it can instead rely on the soundness of the abstract interpretation only [Seo et al. 2003].

While this paper was being reviewed, Besson et al. [Besson et al. 2007] has independently considered the problem of weakening abstraction interpretation results and developed an approach similar to our abstract value slicer. However, the focus of their work is slightly different from ours. Their work emphasizes the issue of the existence of the weakest abstract interpretation results that prove the property of interest. On the other hand, this paper focuses on algorithms for weakening abstract interpretation results and a general framework for validating the soundness of such algorithms.

One interesting future direction is to disconnect the tie between the proof construction and our framework for abstract-value slicers, and revisit the framework. For instance, instead of asking the sliced analysis results to be post fixpoints of *abstract* transfer functions, we might require them to be post fixpoints of *concrete* transfer functions. This might lead to a new formulation of abstract-value slicers, which is suitable for studying semantics-driven slicing.

Abstract-value slicers can be seen as algorithms for simplifying an abstract domain without losing the abstract-interpretation based proof of a property of interest. Concretely, consider a join semi lattice $\mathcal{D}$, a monotone function $F\colon \mathcal{D} \to_m \mathcal{D}$, and abstract values $d_0, d \in \mathcal{D}$, such that

$$d_0 \sqsubseteq d \quad \text{and} \quad F(d_0) \sqsubseteq d_0.$$

Here $\mathcal{D}$ represents an abstract domain for an entire program (not for a single program point) and F an abstract transfer function. Abstract values d_0 and d denote an abstract-interpretation result and a property to verify, respectively,[17] and the condition on d_0 and d means that the abstract interpreter is able to prove d. In this setting, an abstract-value slicer can be considered to compute an upper closure operator ρ on $\mathcal{D}$, such that

$$\rho(d_0) \sqsubseteq d \quad \text{and} \quad (\rho \circ F)(\rho(d_0)) \sqsubseteq \rho(d_0) \ \ (\text{equivalently, } F(\rho(d_0)) \sqsubseteq \rho(d_0)).$$

That is, it simplifies the abstract domain $\mathcal{D}$ to $\rho(\mathcal{D})$, such that the induced best abstract transfer function $\rho \circ F$ in the simplified domain can still verify the property d, using $\rho(d_0)$. Moreover, the slicer attempts to make ρ as abstract as possible.

[17] Domain $\mathcal{D}$ amounts to $\prod_{n \in V} \mathcal{A}$ in Section 2, and d_0 and d correspond to f and $\mathsf{exall}(\epsilon_0, f)$ in Definition 3.6.

The question about simplifying or compressing abstract domains has already been studied in the theory of abstract domain transformations [Filé et al. 1996; Giacobazzi and Ranzato 1997; Giacobazzi et al. 2000; Cortesi et al. 1998]. It would be interesting to see how the existing results can be used to give a new insight for designing better abstract value slicers. We currently expect that the work on compressing abstract domains can answer when the most abstract (i.e., biggest) upper closure operator ρ satisfying the condition in the previous paragraph exists.

Finally, another interesting future direction is to develop more recipes for building abstract-value slicers. In particular, by using the known techniques for constructing abstract interpretations systematically [Cousot and Cousot 1979; Giacobazzi et al. 2000; Giacobazzi and Ranzato 1999; Giacobazzi and Scozzari 1998], we can provide corresponding systematic methods for building abstract-value slicers. Some preliminary results in this direction appear in [Yang et al. 2006].

ACKNOWLEDGMENTS

We would like to thank David Schmidt, Alan Mycroft, Xavier Rival, Daejun Park and anonymous referees for their helpful comments. Seo and Han were supported by Korea Ministry of Information and Communication under the Information Technology Research Center support program, supervised by the Institute of Information Technology Assessment (IITA-2005-C1090-0502-0031). Yang was supported by EPSRC and the Basic Research Program of the Korea Science & Engineering Foundation (grant No. R08-2003-000-10370-0). Yi was supported by Brain Korea 21 Project of Korea Ministry of Education and Human Resources, by IT Leading R&D Support Project of Korea Ministry of Information and Communication, by Korea Research Foundation grant KRF-2003-041-D00528, and by National Security Research Institute of Korea.

REFERENCES

Appel, A. W. 2001. Foundational proof-carrying code. In *Proceedings of the IEEE Symposium on Logic in Computer Science* (Boston, Massachusetts, USA). IEEE Computer Society Press, Los Alamitos, 247–258.

Appel, A. W. and Felty, A. P. 2000. 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* (Boston, Massachusetts, USA). ACM Press, New York, 243–253.

Ball, T., Majumdar, R., Millstein, T., and Rajamani, S. K. 2001. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (SnowBird, Utah, USA). ACM Press, New York, 203–213.

Ball, T. and Rajamani, S. K. 2001. Automatically validating temporal safety properties of interfaces. In *Proceedings of the SPIN Workshop on Model Checking of Software* (Toronto, Canada). Lecture Notes in Computer Science, vol. 2057. Springer-Verlag, 103–122.

Besson, F., Jensen, T., and Turphin, T. 2007. Small witnesses for abstract interpretation-based proofs. In *European Symposium on Programming*. Lecture Notes in Computer Science, vol. 4421. Springer-Verlag, 268–283.

Bourdoncle, F. 1993. Abstract debugging of higher-order imperative languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, United States). ACM Press, New York, 46–55.

Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. 2000. Counterexample-guided abstraction refinement. In *Computer Aided Verification* (Chicago, Illinois, USA). Lecture Notes in Computer Science, vol. 1855. Springer-Verlag, 154–169.

Clarke, E. M., Grumberg, O., and Peled, D. A. 1999. *Model checking*. The MIT Press.

Cortesi, A., Filé, G., and Winsborough, W. H. 1998. The quotient of an abstract interpretation. *Theoretical Computer Science 202,* 1-2, 163–192.

Cousot, P. 1981. Semantic foundations of program analysis. In *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, Chapter 10, 303–342.

Cousot, P. 1999. The calculational design of a generic abstract interpreter. In *Calculational System Design*, M. Broy and R. Steinbrüggen, Eds. NATO ASI Series F. IOS Press, Amsterdam.

Cousot, P. 2005. Abstract interpretation. MIT course 16.399, `http://web.mit.edu/16.399/www/`.

Cousot, P. and Cousot, R. 1977. 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* (Los Angeles, California, USA). ACM Press, New York, 238–252.

Cousot, P. and Cousot, R. 1979. Systematic design of program analysis frameworks. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA). ACM Press, New York, 269–282.

Cousot, P. and Cousot, R. 1999. Refining model checking by abstract interpretation. *Automated Software Engineering 6,* 1, 69–95.

Dams, D., Gerth, R., and Grumberg, O. 1997. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems 19,* 2, 253–291.

Davey, D. A. and Priestley, H. A. 1990. *Introduction to lattices and order*. Cambridge University Press.

Davis, K. and Wadler, P. L. 1990. Backwards strictness analysis: Proved and improved. In *Functional Programming: Proceedings of the 1989 Glasgow Workshop, 21-23 August 1989* (London, UK). Springer-Verlag, 12–30.

Duesterwald, E., Gupta, R., and Soffa, M. L. 1995. Demand-driven computation of interprocedural data flow. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA). ACM Press, New York, 37–48.

Filé, G., Giacobazzi, R., and Ranzato, F. 1996. A unifying view of abstract domain design. *ACM Computing Surveys 28,* 2, 333–336.

Giacobazzi, R. and Mastroeni, I. 2004. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy). ACM Press, New York, 186–197.

Giacobazzi, R. and Ranzato, F. 1997. Refining and compressing abstract domains. In *International Colloquium on Automata, Languages and Programming*. Lecture Notes in Computer Science, vol. 1256. Springer-Verlag, 771–781.

Giacobazzi, R. and Ranzato, F. 1999. The reduced relative power operation on abstract domains. *Theoretical Computer Science 216,* 1-2, 159–211.

Giacobazzi, R., Ranzato, F., and Scozzari, F. 2000. Making abstract interpretations complete. *Journal of the ACM 47,* 2, 361–416.

Giacobazzi, R. and Scozzari, F. 1998. A logical model for relational abstract domains. *ACM Transactions on Programming Languages and Systems 20,* 5, 1067–1109.

Graf, S. and Saïdi, H. 1997. Construction of abstract state graphs with pvs. In *Computer Aided Verification* (Haifa, Israel). Lecture Notes in Computer Science, vol. 1254. Springer-Verlag, 72–83.

Hamid, N., Shaoi, Z., Trifonov, V., Monnier, S., and Ni, Z. 2002. A syntactic approach to foundational proof-carrying code. In *Proceedings of the IEEE Symposium on Logic in Computer Science* (Copenhagen, Denmark). IEEE Computer Society Press, Los Alamitos, 89–100.

Henzinger, T., Jhala, R., Majumdar, R., and Sutre, G. 2002. Lazy abstraction. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA). ACM Press, New York, 58–70.

Henzinger, T., Jhala, R., Majumdar, R., and Sutre, G. 2003. Software verification with blast. In *Proceedings of the SPIN Workshop on Model Checking of Software* (Portland, Oregon, USA). Lecture Notes in Computer Science, vol. 2648. Springer-Verlag, 235–239.

Hoare, C. A. R. 1969. An axiomatic basis for computer programming. *Communications of the ACM 12,* 10, 576–580.

Howe, J. M., King, A., and Lu, L. 2004. Analysing logic programs by reasoning backwards. In *Program Development in Computational Logic*. Lecture Notes in Computer Science, vol. 3049. Springer-Verlag, 152–188.

Hughes, J. 1988. Backwards analysis of functional programs. In *Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*. 187–208.

Hughes, J. and Launchbury, J. 1992. Reversing abstract interpretations. In *European Symposium on Programming* (Rennes, France). Lecture Notes in Computer Science, vol. 582. Springer-Verlag, 269–286.

King, A. and Lu, L. 2002. A backward analysis for constraint logic programs. *Theory and Practice of Logic Programming 2,* 4-5, 517–547.

Massé, D. 2001. Combining backward and forward analyses of temporal properties. In *Proceedings of the Second Symposium PADO'2001, Programs as Data Objects* (Arhus, Denmark). Lecture Notes in Computer Science, vol. 2053. Springer-Verlag, 155–172.

Miné, A. 2001. A new numerical abstract domain based on difference-bound matrices. In *Proceedings of the Second Symposium PADO'2001, Programs as Data Objects* (Aarhus, Denmark). Lecture Notes in Computer Science, vol. 2053. Springer-Verlag, 155–172.

Morrisett, G., Walker, D., Crary, K., and Glew, N. 1998. From System F to typed assembly language. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA). ACM Press, New York, 85–97.

Necula, G. C. 1997. Proof-carrying code. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France). ACM Press, New York, 106–119.

Necula, G. C. and Lee, P. 1997. Safe, untrusted agents using proof-carrying code. In *Special Issue on Mobile Agent Security*, G. Vigna, Ed. Lecture Notes in Computer Science, vol. 1419. Springer-Verlag, 61–91.

Necula, G. C. and Rahul, S. P. 2001. Oracle-based checking of untrusted software. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (London, UK). ACM Press, New York, 142–154.

Necula, G. C. and Schneck, R. 2002. Proof-carrying code with untrusted proof rules. In *Software Security – Theories and Systems* (Tokyo, Japan). Lecture Notes in Computer Science, vol. 2609. Springer-Verlag, 283–298.

Rival, X. 2005a. Abstract dependences for alarm diagnosis. In *Asian Symposium on Programming Languages and Systems* (Tsukuba, Japan). Lecture Notes in Computer Science, vol. 3780. Springer-Verlag, 347–363.

Rival, X. 2005b. Understanding the origin of alarms in ASTRÉE. In *Static Analysis Symposium* (London, UK). Lecture Notes in Computer Science, vol. 3672. Springer-Verlag, 303–319.

Seo, S., Yang, H., and Yi, K. 2003. Automatic construction of Hoare proofs from abstract interpretation results. In *Asian Symposium on Programming Languages and Systems* (Beijing, China). Lecture Notes in Computer Science, vol. 2895. Springer-Verlag, 230–245.

Tip, F. 1995. A survey of program slicing techniques. *Journal of Programming Languages 3,* 3, 121–189.

Wadler, P. and Hughes, R. J. M. 1987. Projections for Strictness Analysis. In *Functional Programming Languages and Computer Architecture* (Portland, Oregon, USA), G. Kahn, Ed. Lecture Notes in Computer Science, vol. 274. Springer, Berlin, 385–407.

Yang, H., Seo, S., Yi, K., and Han, T. 2006. Off-line semantic slicing from abstract interpretation results. Technical Memorandum ROPAS-2006-34, Programming Research Laboratory, School of Computer Science & Engineering, Seoul National University. October. Available at `http://ropas.snu.ac.kr/lib/dock/YaSeYiHa2006.pdf`.