
ROPAS

Research On Program Analysis System
National Creative Research Initiative Center 1998-2003
Programming Research Laboratory, School of Computer Science & Engineering
Seoul National University

ROPAS MEMO
2006-34
October 19, 2006

Off-line Semantic Slicing From Abstract Interpretation Results*

Hongseok Yang
Seoul National University
hyang@ropas.snu.ac.kr

Sunae Seo
Korea Advanced Institute of Science and Technology
saseo@ropas.kaist.ac.kr

Kwangkeun Yi
Seoul National University
kwang@ropas.snu.ac.kr

Taisook Han
Korea Advanced Institute of Science and Technology
han@pllab.kaist.ac.kr

October 19, 2006

Abstract

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 abstract interpretation results. 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.

1 Introduction

Though Proof-Carrying Code(PCC) technologies [NS02, NR01, App01, HST+02] 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 [Nec97, NL97, NR01], or limited to a class of properties that are automatically inferable by the current type system technologies [HST+02, AF00, MWCG98].

*This work is partially 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 the ITRC (Information Technology Research Center) grant IITA-2005-C1090-0502-0031 from Korea Ministry of Information and Communication, by Korea National Security Research Institute, and by Fasoo.com

One proposal [SYY03] for automatic construction of proofs for a wide class of program properties is to combine abstract interpretation [CC77, Cou99] and Hoare logic [Hoa69]. 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 [CC77, Cou99]. 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 $\boxed{\texttt{p}}\ x{:=}E\ \boxed{\texttt{q}}$, the soundness proofs of the abstract interpreter are used to produce a proof that $\texttt{p}$ implies the weakest precondition of $x{:=}E$ for $\texttt{q}$.

This approach in 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. And, 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 properties. For instance, in our experiment with an existing relational analysis [Min01], $62\% - 81\%$ 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. Note that no techniques for representing subproofs compactly by some clever encoding can solve this problem, because they assume that all subproofs are necessary; it is not the purpose of such techniques to identify the useless subproofs. Hence without addressing this problem, the proof-construction method often produces unnecessarily big Hoare proofs.

In PCC, proof's size is a critical issue. Big proofs accompanying mobile code deteriorate 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.

Example 1 As an example where abstract interpretation results are stronger than necessary,

[1] We explain this further in Section 6.

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:

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

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:

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

□

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.

$$\texttt{x:=1; y:=2x}$$

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:

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

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 helps 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 $x \mapsto$`odd` after `y:=2x` is not necessary, but it fails to discover that $x \mapsto$`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 $x \mapsto$`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" [Min01]. 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.

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. □

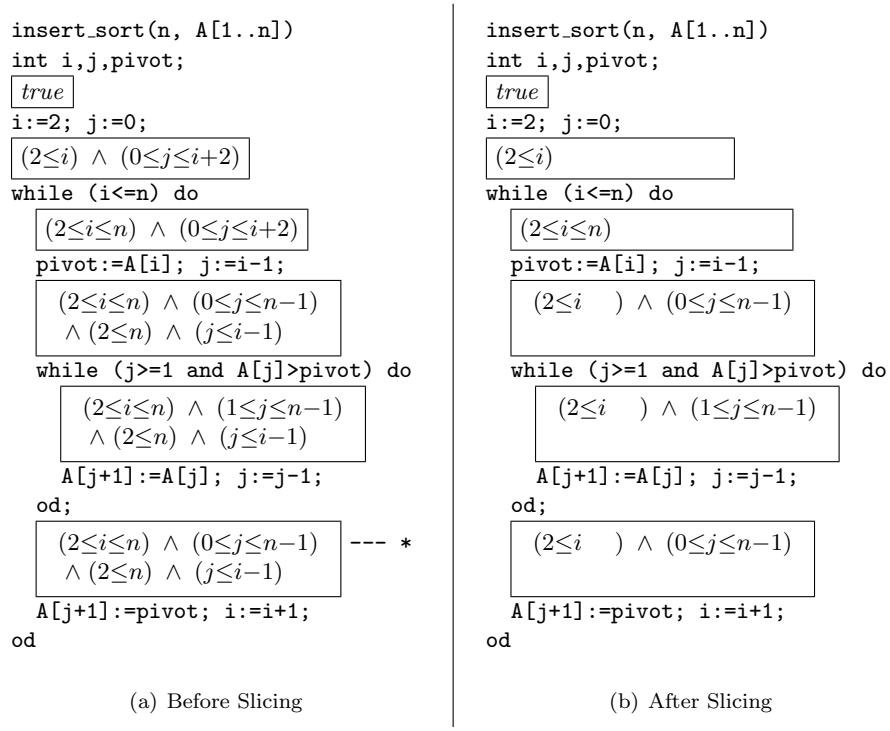
1.2 Our Solution

In this paper, we present an algorithm, called *abstract-value slicer*, that filters out unnecessary invariants from the results of an 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.

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

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



(a) Before Slicing (b) After Slicing

Figure 1: Annotated Insertion Sort Before and After Slicing

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 of the slicing from the analysis is meaningful for the reuse of the analysis. First, the analysis results can be reused. Once-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. And, 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 [Min01], 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 [Tip95, Riv05a] and cone of influences [CGP99] 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 [Riv05a] 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 [GM04]. 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 [GS97, CGJ+00, BR01, BMMR01, HJMS03, HJMS02]. The analyzers [BR01, HJMS03] 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.

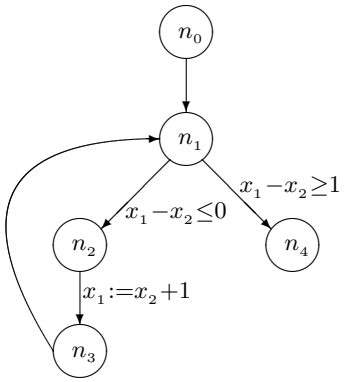
From the view point that our off-line backward abstract-value slicing after an over-approximate forward post-condition analyzer can be simulated by a single under-approximate backward pre-condition analyzer, related works are backward abstract interpreters that under-approximate the weakest preconditions. Please note that, however, backward abstract interpreters in [Riv05b, Cou05, CC99, Mas01, HL92, DGS95, Bou93], *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 [Bou93] and alarm inspection [Riv05b]. On the other hand, abstract model checkers [DGG97] can be seen as backward abstract interpreters that under-approximate the weakest preconditions [Cou81].

Projection analysis [WH87, Hug88, DW90] in functional programs and mode analyses [KL02, HKL04] 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 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, these backward techniques' analysis domains are not so general as to be used in our case. Our abstract-value slicer works with complex domains that are infinite or relational, such as interval domain and zone domain [CC77, Min01], can have nontrivial domain operations (e.g., closure operation in zone domain), or can require an acceleration method (i.e., 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 requirements for these two parameters, and prove that the requirements ensure the correctness of the instantiated slicer. In the next two sections, we present methods for constructing parameters to the slicer, which satisfy the soundness requirements. In Section 4, we describe two techniques for constructing correct back-tracers. In Section 5, we consider well-known methods for constructing an abstract interpreter systematically, such as the cartesian product and the reduction, and define the corresponding methods for the slicers. The defined methods describe how we can construct parameters to the generic slicer, so as to have slicers for the systematically constructed abstract interpreters. In Section 6, we explain the experimental results about one specific abstract-value slicer in the context of proof construction. Finally, we conclude the paper in Section 7.

[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.




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

(a) Program

$n_0 = n_i,\ n_4 = n_f,\ a_0 =$

	x_0	x_1	x_2
x_0	∞	4	3
x_1	-1	∞	∞
x_2	-1	0	∞

(b) Entry, Exit Node and Initial Abstract State

	n_0	n_1	n_2	n_3	n_4
Analysis result as DBMs	x_0 x_1 x_2 x_0 ∞ 4 3 x_1 -1 ∞ ∞ x_2 -1 0 ∞	x_0 x_1 x_2 x_0 ∞ ∞ 3 x_1 ∞ ∞ ∞ x_2 -1 1 ∞	x_0 x_1 x_2 x_0 ∞ ∞ 3 x_1 ∞ ∞ ∞ x_2 -1 0 ∞	x_0 x_1 x_2 x_0 ∞ ∞ 3 x_1 ∞ ∞ -1 x_2 -1 1 ∞	x_0 x_1 x_2 x_0 ∞ ∞ 3 x_1 ∞ ∞ -1 x_2 -1 1 ∞
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

Figure 2: An Example Program and Its Abstract Interpretation Result from Zone Analysis

2 Abstract Interpretation

We consider programs represented by control flow graphs [CC77] . 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 interpretation first uses $\mathcal{A}$ and $[\![-]\!]$ to define "abstract

[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'.

step function" F:

$$\begin{array}{rcl} F & : & \prod_{n\in V}\mathcal{A} \to_m \prod_{n\in V}\mathcal{A} \\ F(f)(n) & \stackrel{\text{def}}{=} & \begin{cases} a_0 & \text{if } n = n_i \\ \bigsqcup\{[\![L(mn)]\!](f(m)) \mid mn \in E\} & \text{otherwise} \end{cases} \end{array}$$

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-fix-point-computation strategy. This post fixpoint becomes the result of the abstract interpretation.

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. And 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

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

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 semi-lattice; 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:

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

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:

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

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:

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

□

[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 $\lambda n. f(n) \sqcup g(n)$.

Example 5 (Zone Analysis) Zone analysis [Min01] 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 [Min01]. 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:

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

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

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

where $\sigma[x_0 \mapsto 0]$ means the extension of state σ with an additional component for x_0: $\sigma[x_0 \mapsto 0](x_i) \stackrel{\text{def}}{=}$ **if** $(i = 0)$ **then** 0 **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 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

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

Figure 3: Abstract Semantics of Atomic Terms in the Zone Analysis

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 [Min01]. In Section 6, we will discuss this better semantics and other improvements used in the original zone analysis [Min01].

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 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 the abstract interpretation: it computes a new post fixpoint f'

[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.

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 the 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 an $\mathcal{A}$-indexed family $\{(\mathcal{E}_a, \mathsf{ex}_a)\}_{a \in \mathcal{A}}$ where $\mathcal{E}_a$ is a finite lattice with structure $(\sqsubseteq_a, \bot_a, \top_a, \sqcup_a, \sqcap_a)$ and ex_a is a function from $\mathcal{E}_a$ to $\mathcal{A}$ such that

$$(\forall e \in \mathcal{E}_a.\ a \sqsubseteq \mathsf{ex}_a(e)) \ \wedge\ (\forall e, e' \in \mathcal{E}_a.\ e \sqsubseteq_a e' \implies \mathsf{ex}_a(e) \sqsubseteq \mathsf{ex}_a(e')).$$

Intuitively, each element in $\mathcal{E}_a$ denotes an "information extractor" that selects some information from the abstract value a, which is to be saved/preserved, and function ex_a extracts information (to be saved/preserved) from a based on such information extractors. Note that we impose two conditions on the extractor application ex_a. The first condition means that the extracting operation $\mathsf{ex}_a(-)$ lifts the underlying abstract value a. When an extractor e in $\mathcal{E}_a$ is applied to the abstract value a, it does not insert any new information, but only selects some information from a; thus, $\mathsf{ex}_a(e)$ should have less information than a (i.e., $a \sqsubseteq \mathsf{ex}_a(e)$), as expressed by the first condition. The second condition is the monotonicity of ex_a, and it ensures that the order $\sqsubseteq_a$ means the "strength" of the information extractors in the reverse direction: if $e \sqsubseteq_a e'$, then e' extracts less information than e. We call $\mathcal{E}_a$ *extractor lattice at* a and ex_a *extractor application at* a. In the paper, we will often omit the subscript a in the order $\sqsubseteq_a$, whenever the "type" of the order is clear from the context.

In the paper, we often use an extractor domain whose extractor lattice $\mathcal{E}_a$ is fixed: for all $a, a' \in \mathcal{A}$, lattice $\mathcal{E}_a$ and $\mathcal{E}_{a'}$ are the same. We call such an extractor domain *simple*, and denote it by the pair $(\mathcal{E}, \{\mathsf{ex}_a\}_{a \in \mathcal{A}})$ of extractor lattice and family of extractor applications.

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

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

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 $\top_e$: all the unselected information is replaced by $\top_e$, while the other selected information remains as it is. □

Example 7 We construct a simple extractor domain $(\mathcal{E}, \{\mathsf{ex}_a\}_{a \in \mathcal{M}})$ 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, $\mathsf{ex}_a(e)$ 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_a is given below:

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

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 $\{(\mathcal{E}_a, \mathsf{ex}_a)\}_{a \in \mathcal{A}}$ be an extractor domain for the abstract interpretation $(\mathcal{A}, [\![-]\!])$. The *back-tracer* $(\!|t|\!)$ *for an atomic term* t in this extractor domain is a parameterized backward extractor transformer. For each atomic term t, let a and b be a pre and post conditions such that the abstract interpretation can prove the triple $\{a\}t\{b\}$[7]. Intuitively, for this pair (a,b) of pre and post conditions, a back-tracer $(\!|t|\!)_{ab}$ for t transforms a post extractor e (for b) to a pre extractor e' (for a), such that the e'-part of a is sufficient to get the e-part of b in the abstract interpretation. More precisely, for all abstract values $a, b \in \mathcal{A}$ with $[\![t]\!]a \sqsubseteq b$, back-tracer $(\!|t|\!)_{ab}$ is a function from $\mathcal{E}_b$ to $\mathcal{E}_a$ such that

$$\forall e \in \mathcal{E}_b.\ \Big([\![t]\!] \circ \mathsf{ex}_a \circ (\!|t|\!)_{ab}\Big)(e) \sqsubseteq \mathsf{ex}_b(e).$$

Note that the back-tracer $(\!|t|\!)_{ab}$ of an atomic term t is not required to be monotone. Thus, it is relatively easy to design one correct back-tracer. For instance, suppose that for every $e \in \mathcal{E}_b$, there exists $e' \in \mathcal{E}_a$ such that[8]

$$\big([\![t]\!] \circ \mathsf{ex}_a\big)(e') \sqsubseteq \mathsf{ex}_b(e).$$

Then, we can define $(\!|t|\!)_{ab}(e)$ to be one such e'. However, designing good back-tracers, not just correct one, is a nontrivial problem, and requires insights about the abstract interpreter and the extractor domain. In Section 4, we will discuss this issue in detail, and provide general techniques for designing good back-tracers.

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 $(\mathcal{E}, \{\mathsf{ex}_a\}_{a \in \mathcal{P}})$ for the analysis in Example 6. The back-tracer $(\!|t|\!)_{ab}$ for each atomic term t in this domain is defined as follows:

$$\begin{array}{rcl}
(\!|x := 2E|\!)_{ab}(e) & \stackrel{\text{def}}{=} & e - \{x\} \\
(\!|x := y|\!)_{ab}(e) & \stackrel{\text{def}}{=} & \textbf{if } x \in e \textbf{ then } (e - \{x\}) \cup \{y\} \textbf{ else } e \\
(\!|x := E|\!)_{ab}(e) & \stackrel{\text{def}}{=} & e - \{x\} \quad \text{(for all the other assignments)} \\
(\!|\texttt{skip}|\!)_{ab}(e) & \stackrel{\text{def}}{=} & e \\
(\!|E \leq E'|\!)_{ab}(e) & \stackrel{\text{def}}{=} & e
\end{array}$$

[7] Thus, $[\![t]\!]a \sqsubseteq b$

[8] This supposition holds if the extractor lattice $\mathcal{E}_a$ contains a *total extractor* e_0 (that is, $\mathsf{ex}_a(e_0) = a$). It is because e_0 itself satisfies the condition in the supposition.

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-tracers which we have just defined are slightly misleading. Although none of these back-tracers use the analysis results (i.e., the subscripts a, b in $(\!|t|\!)_{ab}$), the (other) usual back-tracers are different and use the analysis results critically, in order to compute weaker (i.e., larger) extractors. To see this, suppose that the programming language has a boolean expression $\mathtt{even}?(x)$, which tests whether variable x is even. The abstract semantics and the back-tracer for $\mathtt{even}?(x)$ are given below:

$$\begin{array}{rcl} [\![\mathtt{even}?(x)]\!]a & \stackrel{\text{def}}{=} & \textbf{if } (\mathsf{even} \sqsubseteq_e a(x)) \textbf{ then } a[x \mapsto \mathsf{even}] \textbf{ else } a \\ (\!|\mathtt{even}?(x)|\!)_{ab}(e) & \stackrel{\text{def}}{=} & \textbf{if } (\mathsf{even} \sqsubseteq_e b(x)) \textbf{ then } (e - \{x\}) \textbf{ else } e \end{array}$$

The function $[\![\mathtt{even}?(x)]\!]$ refines the input a by replacing $a(x)$ by the minimum of $a(x)$ and even. Thus, for pre and post conditions (a, b) for $\mathtt{even}?(x)$ (i.e., $[\![\mathtt{even}?(x)]\!]a \sqsubseteq_e b$), 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 $(\!|\mathtt{even}?(x)|\!)_{ab}$ 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 $(\mathcal{E}, \{\mathsf{ex}_a\}_{a \in \mathcal{M}})$ 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) for 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_0, we have that $\mathsf{ex}_b(e_0) = \mathsf{ex}_b(e_0 - \{ij\})$. 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 $k_0 = k_n$, $n \geq 1$, 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. Intuitively, by keeping a negative cycle from the input DBM, $(\!|x_i := E|\!)_{ab}(e)$ maintains the information that the input means a unsatisfiable constraint, i.e., the empty state set. Otherwise, i.e., if a does not have a negative cycle, $(\!|x_i := E|\!)_{ab}(e)$ follows the general two-step computation of the back-tracer, but in an optimized way. It eliminates all the indices kl from e such that $b_{kl} = \infty$ or the assignment $x_i := E$ generates the kl-th entry of b without using the input a (i.e., $i = k \vee i = l$). Then, for each remaining index kl in e, $(\!|x_i := E|\!)_{ab}(e)$ selects a path from k to l in a that provides strong enough information about $(a^*)_{kl}$: $(\!|x_i := E|\!)_{ab}(e)$ chooses a sequence $k_0 \ldots k_n$ of integers

in $[0, N]$ such that $k_0 = k$, $k_n = l$, and

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

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 $\mathsf{mPath}(a, k, l)$.[9] 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-tracers work more clearly, consider the following abstract computation of $x_3 := 0$:

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

Let a and b be, respectively, the input and output DBMs of this computation. 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 $(\!|x_3 := 0|\!)$ 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. □

Back-tracers for all atomic terms induces a back-tracer for an entire program $P = (V, E, n_i, n_f, L)$. Suppose that f and g are the abstract-value annotations for P (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:

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

where $\prod_{n \in V} \mathcal{D}_n$ is the cartesian product of lattices $\mathcal{D}_n$, ordered pointwise. 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. We remark that $(\!|P|\!)_{fg}$ uses $(\!|L(nm)|\!)_{f(n)g(m)}$ only when the subscripts $f(n)g(m)$ are correct: $[\![L(nm)]\!]f(n) \sqsubseteq g(m)$. This is because for each $nm \in E$,

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

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

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

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

where edges is defined by $\mathsf{edges}(k_0 k_1 ... k_n) = \{k_0k_1,\ k_1k_2,\ \ldots,\ k_{n-1}k_n\}$.

Figure 4: Back-tracers for Atomic Terms in the Zone Analysis

For each $h \in \prod_{n \in V} \mathcal{A}$, let ex_h be the application of an extractor annotation:

$$
\begin{array}{rcl}
\mathsf{ex}_h & : & (\prod_{n \in V} \mathcal{E}_{h(n)}) \to (\prod_{n \in V} \mathcal{A}) \\
\mathsf{ex}_h(\epsilon) & \stackrel{\text{def}}{=} & \lambda n{\in}V.\ \mathsf{ex}_{h(n)}(\epsilon(n))
\end{array}
$$

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

Lemma 3.1 *For all ϵ in $\prod_{n \in N} \mathcal{E}_{g(n)}$, we have that* $\Big(F \circ \mathsf{ex}_f \circ (\!|P|\!)_{fg}\Big)(\epsilon) \ \sqsubseteq\ \mathsf{ex}_g(\epsilon)$.

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

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

Thus, by the soundness of the back-tracer for $L(mn)$, we have that

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

We now prove the required inequality as follows:

$$
\begin{array}{l}
\Big(F \circ \mathsf{ex}_f \circ (\!|P|\!)_{fg}\Big)(\epsilon)(n) \\
= \ F\Big((\mathsf{ex}_f \circ (\!|P|\!)_{fg})(\epsilon)\Big)(n)
\end{array}
$$

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

□

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 $\{(\mathcal{E}_a, \mathsf{ex}_a)\}_{a\in\mathcal{A}}$ and back-tracers $(\!|-|\!)$ for all atomic terms in this domain. Suppose that we are given a program $P = (V, E, n_i, n_f, L)$, and let F be the abstract one-step execution of P in the abstract interpretation. The abstract-value slicer SL for the program P has two input parameters: a post fixpoint f of F and an extractor annotation ϵ for f (i.e., $\epsilon \in \prod_{n\in V} \mathcal{E}_{f(n)}$). The first input parameter f denotes the result of the abstract interpretation, and the second ϵ specifies the part of f that is used for verification; although $\mathsf{ex}_f(\epsilon)$ 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[10] $B_f = \lambda\epsilon_0.\, (\!|P|\!)_{ff}\epsilon_0 \sqcap \epsilon_0)$ on $\prod_{n\in V} \mathcal{E}_{f(n)}$, and then computes its fixpoint ϵ' such that $\epsilon' \sqsubseteq \epsilon$, by repeatedly applying B_f from ϵ:

$$\epsilon' \stackrel{\text{def}}{=} B_f^k(\epsilon) \ \text{ for some } k, \text{ such that } \ B_f^k(\epsilon) = B_f^{k+1}(\epsilon).$$

Note that SL always succeeds in computing such ϵ', because the domain $\prod_{n\in V} \mathcal{E}_{f(n)}$ of B_f is finite. The result of the slicer $\mathsf{SL}(f, \epsilon)$ is this computed fixpoint ϵ'.

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:

$$\mathsf{SL}(f, \epsilon) \sqsubseteq \epsilon \quad \text{and} \quad F\big(\mathsf{ex}_f(\mathsf{SL}(f, \epsilon))\big) \sqsubseteq \mathsf{ex}_f(\mathsf{SL}(f, \epsilon)).$$

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 $\mathsf{ex}_f(\epsilon)$, it can also be verified by $\mathsf{ex}_f(\mathsf{SL}(f, \epsilon))$. And the second property means that $\mathsf{ex}_f(\mathsf{SL}(f, \epsilon))$ 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 for P (Lemma 3.1).

Lemma 3.2 *For all post fixpoints f of F and all $\epsilon' \in \prod_{n\in N}(\mathcal{E}_{f(n)})$,*

$$B_f(\epsilon') = \epsilon' \implies F(\mathsf{ex}_f\epsilon') \sqsubseteq \mathsf{ex}_f\epsilon'.$$

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

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

$$\begin{array}{lcll} \mathsf{ex}_f\,\epsilon' & \sqsupseteq & (F \circ \mathsf{ex}_f \circ (\!|P|\!)_{ff})(\epsilon') & \text{(by Lemma 3.1)} \\ & \sqsupseteq & (F \circ \mathsf{ex}_f \circ B_f)(\epsilon') & \text{(since } B_f \sqsubseteq (\!|P|\!)_{ff}\text{, and both } F \text{ and } \mathsf{ex}_f \text{ are monotone)} \\ & = & F(\mathsf{ex}_f(\epsilon')) & \text{(since } B_f(\epsilon') = \epsilon'\text{).} \end{array}$$

□

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

Proposition 3.3 (Correctness) *For all post fixpoints f of F and all $\epsilon \in \prod_{n \in N}(\mathcal{E}_{f(n)})$, the slicer $\mathsf{SL}(f, \epsilon)$ terminates, and outputs ϵ' such that $\epsilon' \sqsubseteq \epsilon$ and $F(\mathsf{ex}_f(\epsilon')) \sqsubseteq \mathsf{ex}_f(\epsilon')$.*

Example 10 Consider the following result from the evenness analysis:

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

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:

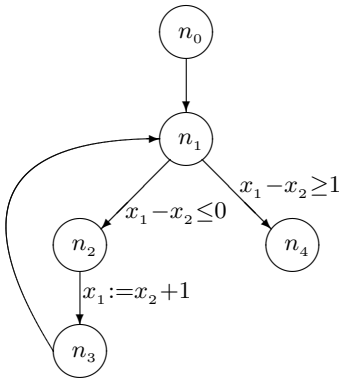
$$\boxed{\begin{array}{lll} x & \mapsto & \top_e \\ y & \mapsto & \top_e \end{array}}\ \texttt{y:=2y;}\ \boxed{\begin{array}{lll} x & \mapsto & \top_e \\ y & \mapsto & \top_e \end{array}}\ \texttt{x:=2y;}\ \boxed{\begin{array}{lll} x & \mapsto & \mathsf{even} \\ y & \mapsto & \top_e \end{array}}\ \texttt{y:=x}\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

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 atomic term t. However, designing a *good* back-tracer for t is difficult, and requires special knowledge about




```
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

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

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 $\{(\mathcal{E}_a, \mathsf{ex}_a)\}_{a \in \mathcal{A}}$ is given for this abstract interpretation.

4.1 Best Back-tracer Construction

The first method constructs the *best* back-tracer for each atomic term, and it is a slight modification of a rather well-known "reversing technique" [HL92, DGS95]. Let t be an atomic term, and let a, b be abstract values in $\mathcal{A}$ such that $[\![t]\!]a \sqsubseteq b$. For these t, a, b, the method defines the

back-tracer $(\!|t|\!)_{ab}$ as follows:

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

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 the join in the definition of $(\!|t|\!)_{ab}(e)$ always exists, because it is over a finite subset of $\mathcal{E}_a$ and $\mathcal{E}_a$ has all finite joins. Another thing to note is 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., $\mathsf{ex}_a(e_0) \sqsubseteq \mathsf{ex}_a(e')$), and so, it induces a better slice of a than e_0.

Unfortunately, this method does not always construct a correct back-tracer; in general, the constructed $(\!|t|\!)_{ab}$ does not satisfy the following correctness requirement from the definition of the back-tracer:

$$\forall e \in \mathcal{E}_b.\ \Big([\![t]\!] \circ \mathsf{ex}_a \circ (\!|t|\!)_{ab}\Big)(e) \ \sqsubseteq\ \mathsf{ex}_b(e).$$

For this correctness problem, we adopt the solution in [HL92, DGS95]. It is to restrict the use of the method for join-preserving functions: we use the constructed $(\!|t|\!)_{ab}$, only when $[\![t]\!]$ and ex_a preserve the finite joins. To see why this restriction is a solution, suppose that $[\![t]\!]$ and ex_a preserve the finite joins. Then, their composition $([\![t]\!] \circ \mathsf{ex}_a)$ also preserves the finite joins, and so, for all extractors $e \in \mathcal{E}_b$, we have that

$$\begin{aligned}
&\Big([\![t]\!] \circ \mathsf{ex}_a \circ (\!|t|\!)_{ab}\Big)(e) \\
&= ([\![t]\!] \circ \mathsf{ex}_a)\Big(\bigsqcup\{e_0 \mid ([\![t]\!] \circ \mathsf{ex}_a)(e_0) \sqsubseteq \mathsf{ex}_b(e)\}\Big) && \text{(by the definition of } (\!|t|\!)_{ab}) \\
&= \Big(\bigsqcup\{([\![t]\!] \circ \mathsf{ex}_a)(e_0) \mid ([\![t]\!] \circ \mathsf{ex}_a)(e_0) \sqsubseteq \mathsf{ex}_b(e)\}\Big) && \text{(by join preservation)} \\
&\sqsubseteq \mathsf{ex}_b(e).
\end{aligned}$$

Now, note that this order relationship implies the correctness of $(\!|t|\!)_{ab}$.

Lemma 4.1 *If both $[\![t]\!]$ and ex_a preserve the finite joins, then $(\!|t|\!)_{ab}$ satisfies the correctness requirement for back-tracers.*

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 is finite 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 $\{(\mathcal{E}, \mathsf{ex}_a)\}_{a \in \mathcal{M}}$ 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 every extractor application ex_a preserves the finite joins. 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

$$\forall ij.\ \ (\llbracket t \rrbracket a)_{ij} = f_{ij}(a_{ij}).$$

2. Second, when a function family $\{f_{ij}\}_{ij}$ determines $\llbracket t \rrbracket$, it can be used to simplify the "correctness condition" for pre and post extractors: for every pair (a, b) of pre and post conditions (i.e., $\llbracket t \rrbracket a \sqsubseteq b$), pre extractor $e_0 \in \mathcal{E}$ and post extractor $e \in \mathcal{E}$, we have that

$$\Big(\llbracket t \rrbracket(\mathsf{ex}_a(e_0)) \sqsubseteq \mathsf{ex}_b(e)\Big) \iff \Big(\forall ij.\ \ ij \in e \Rightarrow (ij \in e_0 \ \vee\ f_{ij}(\infty) \leq b_{ij})\Big).$$

   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 every output b' of $\llbracket t \rrbracket$ has an ij-th entry strong enough to imply b_{ij} (i.e., $b'_{ij} \leq b_{ij}$). 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:

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

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-tracer for boolean expressions $E \leq E'$, assignments $x_i := x_i + c$ and command `skip` in Figure 4 is such a further optimization. □

4.2 Extension Method

The second method, called *extension method*, assumes two properties of the extractor domain. Recall (from the standard lattice theory [DP90]) 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 $\mathcal{E}_a$ is dual atomic, and the second assumption is that each extractor application preserves all finite meets:

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

Intuitively, these two assumptions mean that every extractor e in $\mathcal{E}_a$ 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:

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

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 abstract values in $\mathcal{A}$ such that $[\![t]\!]a \sqsubseteq b$. The first step of the extension method is to define *partial back-tracer* g for t at a and b: g is a partial function of type $\mathcal{E}_b \rightharpoonup \mathcal{E}_a$ such that (1) the domain of g is precisely the set of dual atoms in $\mathcal{E}_b$ and (2) for all post extractors in $\mathsf{dom}(g)$, g calculates correct pre extractors:

$$\forall e \in \mathsf{dom}(g).\ ([\![t]\!] \circ \mathsf{ex}_a \circ g)(e) \sqsubseteq \mathsf{ex}_b e.$$

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 $\mathcal{E}_a$ 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 4.2 *For all extractors $e \in \mathcal{E}_b$, we have that*

$$\Big([\![t]\!] \circ \mathsf{ex}_a \circ (\!|t|\!)_{ab}\Big)(e) \ \sqsubseteq \ \mathsf{ex}_b(e).$$

Proof: We prove the lemma as follows:

$$\begin{aligned}
&[\![t]\!]\Big(\mathsf{ex}_a((\!|t|\!)_{ab} e)\Big) \\
&= [\![t]\!]\Big(\mathsf{ex}_a\Big(\bigsqcap\{g(e_1) \mid e \sqsubseteq e_1 \ \text{ and } \ e_1 \text{ is a dual atom}\}\Big)\Big) && \text{(by the definition of } (\!|t|\!)_{ab}) \\
&= [\![t]\!]\Big(\bigsqcap\{(\mathsf{ex}_a \circ g)(e_1) \mid e \sqsubseteq e_1 \ \text{ and } \ e_1 \text{ is a dual atom}\}\Big) \\
& && \text{(since } \mathsf{ex}_a \text{ preserve all finite meets)} \\
&\sqsubseteq \bigsqcap\{[\![t]\!]((\mathsf{ex}_a \circ g)(e_1)) \mid e \sqsubseteq e_1 \ \text{ and } \ e_1 \text{ is a dual atom}\} && \text{(since } [\![t]\!] \text{ is monotone)} \\
&\sqsubseteq \bigsqcap\{\mathsf{ex}_b(e_1) \mid e \sqsubseteq e_1 \ \text{ and } \ e_1 \text{ is a dual atom}\} \\
& && \text{(since } ([\![t]\!] \circ \mathsf{ex}_a \circ g)(e_1) \sqsubseteq \mathsf{ex}_b(e_1) \text{ for all dual atoms } e_1) \\
&= \mathsf{ex}_b\Big(\bigsqcap\{e_1 \mid e \sqsubseteq e_1 \ \text{ and } \ e_1 \text{ is a dual atom}\}\Big) && \text{(since } \mathsf{ex}_b \text{ preserves all finite meets)} \\
&\sqsubseteq \mathsf{ex}_b(e) && \text{(since } \mathcal{E}_b \text{ is dual atomic).}
\end{aligned}$$

□

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 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),
   e0 is an output (a pre extractor) *)
X := {e1 | e ⊑ e1 and e1 is a dual atom};
e0 := ⊤;
for each e1 ∈ X
do
    lookup e1' s.t. (e1, e1') ∈ T;
    e0 := e0 ⊓ e1'
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 $\mathcal{E}_b$. Even when the extractor lattices $\mathcal{E}_a$ and $\mathcal{E}_b$ are 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 lattices.

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 the back-tracer 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 $(\mathcal{E}, \{\mathsf{ex}_a\})$ 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 applications ex_a preserve finite meets. Here we apply the method to construct the 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 correct partial back-tracer g of $x_i := E$ 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,

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

When a does not contain a negative cycle,

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

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

5 Systematic Construction of An Abstract-value Slicer

An abstract interpretation is often constructed systematically. It is either an instance of a framework for analyses of a specific form, such as the one for nonrelational analyses [Cou99], or the combination of other existing abstract interpretations [CC79, CC92], such as the reduced product of abstract interpretations.

In this section, we consider abstract-value slicers for such systematically-constructed abstract interpretations. First, we consider Cousot's framework for designing nonrelational abstract interpretations [Cou99], and define a corresponding framework for abstract-value slicers. This framework for slicers defines a generic abstract-value slicer, which can be instantiated for all nonrelational abstract interpretations in Cousot's framework. Next, we consider three well-known methods for constructing an abstract interpretation — cartesian product, reduction, and (a nonmonotonic version of) cardinal power — and define corresponding methods for abstract-value slicers. Throughout this section, we assume the correctness of considered abstract interpretations; all the considered constructions for abstract interpreters are assumed to be used only with proper parameters, so that the resulting abstract interpretation computes correct approximate program invariants.

5.1 Slicer for Nonrelational Abstract Interpretations

Intuitively, an abstract interpretation is nonrelational, if it does not keep track of the relationship between variables. Let Vars be a finite set of program variables. Mathematically, a

nonrelational abstract interpretation is characterized by the form of its abstract domain $\mathcal{A}$: $\mathcal{A}$ is the function space from Vars to some join semi-lattice $(\mathcal{V}, \sqsubseteq, \bot, \sqcup)$, i.e., $\mathcal{A} = [\mathsf{Vars} \to \mathcal{V}]$. Here each element in $\mathcal{V}$ denotes a set of integers, and a function $a \in \mathcal{A}$ denotes a set of memory states where the value of each variable x belongs to the integer set $a(x)$.

In [Cou99], Cousot gave the generic abstract semantics of atomic terms for the nonrelational abstract interpretations. In this paper, we consider a simplified version of Cousot's generic semantics: (1) our abstract semantics of boolean expressions is more approximate than Cousot's, and (2) our semantics considers a smaller set of atomic terms than Cousot's. Consider a nonrelational abstract interpretation whose abstract domain is $[\mathsf{Vars} \to \mathcal{V}]$ for some join semi-lattice $(\mathcal{V}, \sqsubseteq, \bot, \sqcup)$. The generic abstract semantics assumes that $\mathcal{V}$ has the following abstract operators:

$$\hat{+} : \mathcal{V} \times \mathcal{V} \to_m \mathcal{V} \qquad \hat{\mathsf{db}} : \mathcal{V} \to_m \mathcal{V}$$
$$\hat{\mathsf{leq}} : \mathsf{Ints} \to (\mathcal{V} \to_m \mathcal{V}) \qquad \hat{\mathsf{geq}} : \mathsf{Ints} \to (\mathcal{V} \to_m \mathcal{V}) \qquad \mathsf{abs} : \mathsf{Ints} \to \mathcal{V}$$

The first two operators, $\hat{+}$ and $\hat{\mathsf{db}}$, are the abstraction of the concrete addition and doubling (i.e., $\lambda i.2 \times i$),[11] and the next two operators, $\hat{\mathsf{leq}}(n)$ and $\hat{\mathsf{geq}}(n)$, are pruning operators for the predicates $- \leq n$ and $- \geq n$, respectively; $\hat{\mathsf{leq}}(n)(v)$ abstracts all integers n_0 in (the concretization of) v that satisfy $n_0 \leq n$, and $\hat{\mathsf{geq}}(n)(v)$ abstracts all integers n_0 in v that satisfy $n_0 \geq n$. The last operator abs is the abstraction of integers; it maps each integer n to some $v \in \mathcal{V}$ such that n belongs to the concretization of v. Usually, $\mathsf{abs}(n)$ is the best abstraction of $\{n\}$, but this bestness is not needed for the results in this paper, and so, we do not require it.

The generic semantics defines two semantic functions: one for integer expressions, and the other for atomic terms. For each integer expression E, the generic abstract semantics defines a monotone function $[\![E]\!]$ from $\mathcal{A}$ to $\mathcal{V}$ as follows:

$$[\![n]\!]a \stackrel{\text{def}}{=} \mathsf{abs}(n) \qquad [\![x]\!]a \stackrel{\text{def}}{=} a(x) \qquad [\![E + E']\!]a \stackrel{\text{def}}{=} [\![E]\!]a \hat{+} [\![E']\!]a \qquad [\![2E]\!]a \stackrel{\text{def}}{=} \hat{\mathsf{db}}([\![E]\!]a)$$

For each atomic term t, it defines the following monotone function $[\![t]\!]$ from $\mathcal{A}$ to $\mathcal{A}$:

$$\begin{aligned}
[\![x := E]\!]a &\stackrel{\text{def}}{=} a[x \mapsto [\![E]\!]a] \\
[\![\texttt{skip}]\!]a &\stackrel{\text{def}}{=} a \\
[\![x \leq n]\!]a &\stackrel{\text{def}}{=} a[x \mapsto \hat{\mathsf{leq}}(n)(a(x))] \\
[\![n \leq x]\!]a &\stackrel{\text{def}}{=} a[x \mapsto \hat{\mathsf{geq}}(n)(a(x))] \\
[\![E \leq E']\!]a &\stackrel{\text{def}}{=} a \quad \text{(for all the other inequalities)}
\end{aligned}$$

Note that the abstract semantics of integer expressions and assignments is defined "homomorphically": each integer operation, such as $+$, is interpreted by its abstract meaning, such as $\hat{+}$. The abstract semantics of an inequality $E \leq E'$ uses operators $\hat{\mathsf{leq}}$ and $\hat{\mathsf{geq}}$, and prunes some states that do not satisfy the inequality. In [Cou99], Cousot proved the correctness of this abstract semantics, assuming that the provided abstract operators on $\mathcal{V}$ satisfy certain soundness conditions.

We now define a generic abstract-value slicer for the nonrelational abstract interpretations. This generic slicer has two analysis-specific parameters. The first parameter is a $\mathcal{V}$-indexed family $\{(\mathcal{S}_v, \mathsf{vex}_v)\}_{v \in \mathcal{V}}$, which intuitively specifies an "extractor domain" for $\mathcal{V}$. In the family, we require that each $\mathcal{S}_v$ be a finite lattice with structure $(\sqsubseteq_v, \bot_v, \top_v, \sqcap_v, \sqcup_v)$, and that vex_v be a monotone function from $\mathcal{S}_v$ to $\mathcal{V}$ such that $v \sqsubseteq \mathsf{vex}_v(s)$. Note that the parameter is required to satisfy exactly the same requirements as the ones for the extractor domains, but its "target" $\mathcal{V}$ is not the abstract domain $\mathcal{A}$ of the abstract interpretation, but the building block of $\mathcal{A}$. The second parameter is the five parameterized "back-tracers" for the abstract operators $\hat{+}$,

[11] To simplify the presentation, we consider only doubling, instead of arbitrary multiplication.

$$
\begin{array}{rcll}
\mathsf{bplus}(u,v,w) & : & \mathcal{S}_w \to \mathcal{S}_u \times \mathcal{S}_v & (\text{when } \hat{+}(u,v) \sqsubseteq w) \\
\mathsf{bdb}(v,w) & : & \mathcal{S}_w \to \mathcal{S}_v & (\text{when } \hat{\mathsf{db}}(v) \sqsubseteq w) \\
\mathsf{bleq}(n,v,w) & : & \mathcal{S}_w \to \mathcal{S}_v & (\text{when } \hat{\mathsf{leq}}(n)(v) \sqsubseteq w) \\
\mathsf{bgeq}(n,v,w) & : & \mathcal{S}_w \to \mathcal{S}_v & (\text{when } \hat{\mathsf{geq}}(n)(v) \sqsubseteq w) \\
\mathsf{bord}(v,w) & : & \mathcal{S}_w \to \mathcal{S}_v & (\text{when } v \sqsubseteq w)
\end{array}
$$

(a) Type of Back-tracers

$$
\begin{array}{rcl}
(\hat{+}(u,v) \sqsubseteq w) & \Longrightarrow & \big((\hat{+} \circ (\mathsf{vex}_u \times \mathsf{vex}_v) \circ \mathsf{bplus}(u,v,w))(s) \sqsubseteq \mathsf{vex}_w s\big) \\
(\hat{\mathsf{db}}(v) \sqsubseteq w) & \Longrightarrow & \big((\hat{\mathsf{db}} \circ \mathsf{vex}_v \circ \mathsf{bdb}(v,w))(s) \sqsubseteq \mathsf{vex}_w s\big) \\
(\hat{\mathsf{leq}}(n)(v) \sqsubseteq w) & \Longrightarrow & \big((\hat{\mathsf{leq}}(n) \circ \mathsf{vex}_v \circ \mathsf{bleq}(n,v,w))(s) \sqsubseteq \mathsf{vex}_w s\big) \\
(\hat{\mathsf{geq}}(n)(v) \sqsubseteq w) & \Longrightarrow & \big((\hat{\mathsf{geq}}(n) \circ \mathsf{vex}_v \circ \mathsf{bgeq}(n,v,w))(s) \sqsubseteq \mathsf{vex}_w s\big) \\
(v \sqsubseteq w) & \Longrightarrow & \big((\mathsf{vex}_v \circ \mathsf{bord}(v,w))(s) \sqsubseteq \mathsf{vex}_w s\big)
\end{array}
$$

where $f \times g = \lambda(x,y).\,(f(x), g(y))$.

(b) Soundness Requirements for Back-tracers

Figure 6: Abstract Semantics of Atomic Terms in the Nonrelational Analysis

$\hat{\mathsf{db}}$, $\hat{\mathsf{leq}}$, $\hat{\mathsf{geq}}$, and the order $\sqsubseteq$ on $\mathcal{V}$. The type of these five back-tracers — bplus, bdb, bleq, bgeq, bord — and the requirements for them are shown in Figure 6. Note that each of the five requirements has the same form as the correctness condition of the back-tracers for atomic terms; as in the case of atomic terms, these requirements mean that the defined back-tracers return strong enough "extractors": the returned "extractors" select enough information from the input that can make abstract operators on $\mathcal{V}$ produce the "important part" of the output.

Given the analysis-specific parameters, the generic abstract-value slicer is instantiated into a specific slicer. The extractor domain $\{(\mathcal{E}_a, \mathsf{ex}_a)\}_{a \in \mathcal{A}}$ of the instantiated slicer is the pointwise extension of the first parameter $\{(\mathcal{S}_v, \mathsf{vex}_v)\}_{v \in \mathcal{V}}$:

$$\mathcal{E}_a \stackrel{\text{def}}{=} \prod_{x \in \mathsf{Vars}} \mathcal{S}_{a(x)} \qquad\qquad \mathsf{ex}_a(e) \stackrel{\text{def}}{=} \lambda x.\mathsf{vex}_{a(x)}(e(x)).$$

Intuitively, an extractor e in $\mathcal{E}_a$ is a table indexed by variables; for each variable x, the x-entry of this table is an element in $\mathcal{S}_{a(x)}$, and specifies how much information should be extracted from $a(x)$.

The other component of the instantiated slicer, namely, the back-tracers for atomic terms, are shown in Figure 7. Note that the figure defines the back-tracers for not just atomic terms but also integer expressions. Both back-tracers $(\!|E|\!)_{av}$ and $(\!|t|\!)_{ab}$ compute a sufficiently strong extractor for the input, given a specification about the important part of the output. The most complex case is the back-tracer for the assignment $x := E$. Suppose that $(\!|x := E|\!)$ was given a pair (a,b) of pre and post conditions (i.e., $[\![x := E]\!]a \sqsubseteq b$) and a post extractor $e \in \mathcal{E}_b$, and returned e_0, a pre extractor. The returned extractor e_0 is an indexed "table." When the index $y \in \mathsf{Vars}$ is not the assigned variable x (i.e., $y \not\equiv x$), the table e_0 contains the following entry for y:

$$\Big(\mathsf{bord}\big(a(y), b(y)\big)(e(y))\Big) \sqcap \Big((\!|E|\!)_{a(b(x))}(e(x))(y)\Big).$$

To understand this entry, recall that in the abstract semantics of $x := E$, $a(y)$ can be used for two purposes: to obtain $b(y)$ and to compute the value $[\![E]\!]a$. The left hand side of the meet operation ensures that the $e_0(y)$-part of $a(y)$ is strong enough to produce the $e(y)$-part of $b(y)$. And the right hand side of the meet operation guarantees that the $e_0(y)$-part of $a(y)$ can make the computation $[\![E]\!](a)$ generate the $e(x)$-part of $b(x)$. When the index y is the assigned

$$
\begin{array}{rcl}
(\!|E|\!)_{av} & : & \mathcal{S}_v \to \mathcal{E}_a \\
(\!|n|\!)_{av}\, s & \stackrel{\text{def}}{=} & \top \\
(\!|x|\!)_{av}\, s & \stackrel{\text{def}}{=} & \lambda y.\ \textbf{if}\ (x \equiv y)\ \textbf{then}\ \mathsf{bord}(a(x), v)(s)\ \textbf{else}\ \top \\
(\!|E + E'|\!)_{av}\, s & \stackrel{\text{def}}{=} & \textbf{let}\ w = [\![E]\!]a\ \textbf{and}\ w' = [\![E']\!]a \\
 & & \textbf{in}\ \Big((\lambda(x,y).\ x \sqcap y) \circ ((\!|E|\!)_{aw} \times (\!|E'|\!)_{aw'}) \circ \mathsf{bplus}(w, w', v)\Big)(s) \\
(\!|2E|\!)_{av}\, s & \stackrel{\text{def}}{=} & \textbf{let}\ u = [\![E]\!]a\ \textbf{in}\ \Big(((\!|E|\!)_{au} \circ \mathsf{bdb}(u, v))(s)\Big) \\
 & & \\
(\!|t|\!)_{ab}\, e & : & \mathcal{E}_b \to \mathcal{E}_a \\
(\!|x := E|\!)_{ab}\, e & \stackrel{\text{def}}{=} & \Big(\lambda y.\ \textbf{if}\ (x \equiv y)\ \textbf{then}\ \top\ \textbf{else}\ \mathsf{bord}(a(y), b(y))(e(y))\Big) \sqcap \Big((\!|E|\!)_{a(b(x))}(e(x))\Big) \\
(\!|\texttt{skip}|\!)_{ab}\, e & \stackrel{\text{def}}{=} & \lambda y.\ \mathsf{bord}(a(y), b(y))(e(y)) \\
(\!|x \le n|\!)_{ab}\, e & \stackrel{\text{def}}{=} & \lambda y.\ \textbf{if}\ (x \equiv y)\ \textbf{then}\ \Big(\mathsf{bleq}(n, a(x), b(x))(e(x))\Big)\ \textbf{else}\ \Big(\mathsf{bord}(a(y), b(y))(e(y))\Big) \\
(\!|n \le x|\!)_{ab}\, e & \stackrel{\text{def}}{=} & \lambda y.\ \textbf{if}\ (x \equiv y)\ \textbf{then}\ \Big(\mathsf{bgeq}(n, a(x), b(x))(e(x))\Big)\ \textbf{else}\ \Big(\mathsf{bord}(a(y), b(y))(e(y))\Big) \\
(\!|E \le E'|\!)_{ab}\, e & \stackrel{\text{def}}{=} & \lambda y.\ \mathsf{bord}(a(y), b(y))(e(y))
\end{array}
$$

Figure 7: Back-tracers for Integer Expressions and Atomic Terms

$\mathcal{V} \stackrel{\text{def}}{=} \mathsf{Interval} = \{[n,m] \mid n, m \in \mathsf{Ints} \cup \{\infty, -\infty\}\}$
$\bot \stackrel{\text{def}}{=} [\infty, -\infty]$

$[n,m] \sqsubseteq [n',m'] \stackrel{\text{def}}{\Leftrightarrow} n' \le n\ \wedge\ m \le m'$
$[n,m] \sqcup [n',m'] \stackrel{\text{def}}{=} [\mathsf{min}(n,n'), \mathsf{max}(m,m')]$

$\hat{\mathsf{leq}}(n)([n',m']) \stackrel{\text{def}}{=} [n', \mathsf{min}(n,m')]$
$\hat{\mathsf{geq}}(n)([n',m']) \stackrel{\text{def}}{=} [\mathsf{max}(n,n'), m']$
$\mathsf{abs}(n) \stackrel{\text{def}}{=} [n,n]$

$[n,m] \hat{+} [n',m'] \stackrel{\text{def}}{=} [n+n', m+m']$
$\hat{\mathsf{db}}([n,m]) \stackrel{\text{def}}{=} [2n, 2m]$

Figure 8: Parameters for Interval Analysis

variable x, the corresponding entry of the table e_0 only keeps the right hand side of the above meet operation; the old value of the variable x is not directly related to the new value of x, but it is only indirectly related through the right hand side expression E.

Example 14 Interval analysis [CC77] is the most well-known nonrelational abstract interpretation. In Figure 8, we recall the parameters that are needed to instantiate Cousot's generic abstract semantics to interval analysis.

We construct an abstract-value slicer for interval analysis, by instantiating the generic slicer. The first parameter is given below:

$$
\begin{array}{l}
\mathcal{S}_{[n,m]} \stackrel{\text{def}}{=} \wp(\{\mathsf{l}, \mathsf{u}\}),\ \text{ordered by superset (i.e., } s \sqsubseteq s' \text{ iff } s \supseteq s') \\
\mathsf{vex}_{[n,m]}(s) \stackrel{\text{def}}{=} \big[(\textbf{if}\ \mathsf{l} \in s\ \textbf{then}\ n\ \textbf{else}\ -\infty),\ \ (\textbf{if}\ \mathsf{u} \in s\ \textbf{then}\ m\ \textbf{else}\ \infty)\big].
\end{array}
$$

Each s in $\mathcal{S}_{[n,m]}$ specifies the bounds of $[n,m]$ that should be selected; if s contains l, the lower bound n of the interval $[n,m]$ must be extracted, and if s contains u, the upper bound m has to be extracted.

Let $\mathsf{dInf}([n,m])$ be the following function of type $\mathcal{S}_{[n,m]} \to \mathcal{S}_{[n,m]}$:

$$\mathsf{dInf}([n,m])(s) \stackrel{\text{def}}{=} s - \{\mathsf{l} \mid n = -\infty\} - \{\mathsf{u} \mid m = \infty\}$$

The back-tracers are defined as follows:

$$\begin{aligned}
\mathsf{bleq}(n_0,[n,m],[n',m'])(s) &\stackrel{\text{def}}{=} \mathsf{dInf}([n',m'])(s) - \{\mathsf{u} \mid n_0 \le m'\} \\
\mathsf{bgeq}(n_0,[n,m],[n',m'])(s) &\stackrel{\text{def}}{=} \mathsf{dInf}([n',m'])(s) - \{\mathsf{l} \mid n_0 \ge n'\} \\
\mathsf{bord}([n,m],[n',m'])(s) &\stackrel{\text{def}}{=} \mathsf{dInf}([n',m'])(s) \\
\mathsf{bplus}([n,m],[n',m'],[n'',m''])(s) &\stackrel{\text{def}}{=} (\mathsf{dInf}([n'',m''])(s),\ \mathsf{dInf}([n'',m''])(s)) \\
\mathsf{bdb}([n,m],[n',m'])(s) &\stackrel{\text{def}}{=} \mathsf{dInf}([n',m'])(s)
\end{aligned}$$

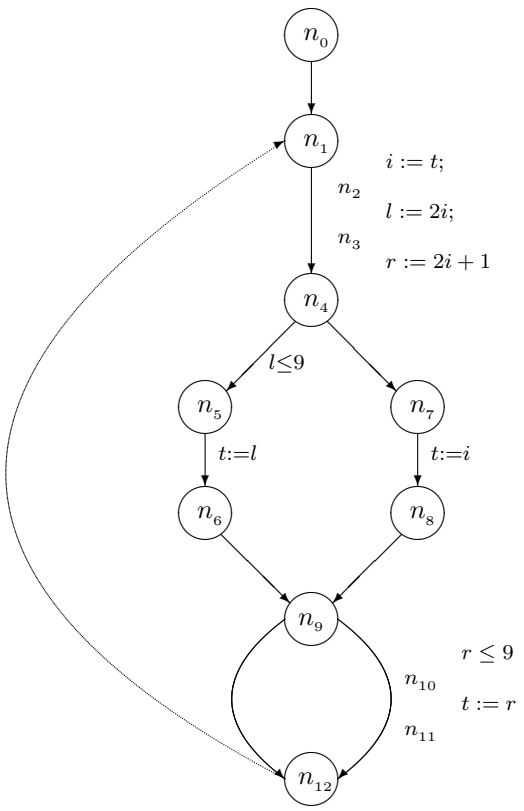
Figure 9: Back-tracers for Abstract Operators and Order for Intervals

```
max_heapify(A)
  i,l,r,t:= 1;
  do
    i:=t;
    l:=2i; r:=2i+1;
    if ((l<=9) and (A[l]>A[i]))
      then t:=l
      else t:=i
    fi;
    if ((r<=9) and (A[r]>A[t]))
      then t:=r
    fi;
    if (t!=i)
      then swap(A[i],A[t])
    fi
  while (t!=i)
```

Figure 10: Original Program `max_heapify`, An Iterative Version

The second parameter is the back-tracers for the abstract operators and the order for intervals, and it is shown in Figure 9. All those back-tracers preprocess the input s with $\mathsf{dInf}([n,m])$. This preprocessing gets rid of u from s when the upper bound m is ∞ so that the bound m has no information (i.e., $x \le m \Leftrightarrow \mathit{true}$). Similarly, the preprocessing eliminates l from s when the lower bound n is $-\infty$ and has no information. The main step of each back-tracer, which comes after the preprocessing, depends on the original abstract operation that the function currently back-traces. We focus on the back-tracer bleq for pruning $\hat{\mathsf{leq}}$. After preprocessing, $\mathsf{bleq}(n_0,[n,m],[n',m'])$ eliminates u from the preprocessed s, when $n_0 \le m'$. Intuitively, the step detects the case that the upper bound m of the input is not needed to obtain m', and in that case, it eliminates u from the preprocessed s. To see this intuition, note that if $n_0 \le m'$, then for every input interval $[n'',m'']$, the upper bound of the output interval $\hat{\mathsf{leq}}(n_0)([n'',m''])$ is at most n_0, and so, it is at most m'. Thus, $n_0 \le m'$ implies that the upper bound m of the input interval $[n,m]$ is not necessary for obtaining the upper bound m' of the output interval.

We manually applied interval analysis and the slicer for the code in Figure 10, which takes a binary tree in an array of size 9 and transforms it into a heap data structure [CLRS01]. We




```
                 i,l,r,t := 1;
n0
                 do
n1, n2, n3
                   i:=t;
                   l:=2i; r:=2i+1;
n4
                   if ((l<=9) and (*))
n5, n6               then t:=l
n7, n8               else t:=i
n9                 fi;
                   if ((r<=9) and (*))
n10, n11             then i:=r
                   fi;
n12
                   skip
                 while (*)
```

Figure 11: Example Program **simple_max_heapify**

represented the code as a program (i.e., control flow graph) in Figure 11, and applied interval analysis first to verify that all the array accesses of the original program are within bounds. The second row of the table in Figure 12(a) shows a part of the analysis result. Then, we ran the abstract-value slicer with a "post" extractor annotation which selects the intervals that directly imply the absence of array bounds errors. The first row of the table in Figure 12(a) shows a part of this post extractor annotation. The tables in Figure 12(b) and 12(c) show the result of the slicing. As a whole, 54 out of 104 bounds in the analysis result have been identified as unnecessary, so that they changed into either ∞ or $-\infty$. Look at the invariant at the node n_4. At this node, the original analysis result has the invariant

$$1 \leq i \leq 9 \ \wedge \ 2 \leq l \leq 18 \ \wedge \ 3 \leq r \leq 19 \ \wedge \ 1 \leq t \leq 9.$$

After slicing, only half of the constraints in the above invariant remain, so that the new sliced invariant becomes:

$$1 \leq i \leq 9 \ \wedge \ 2 \leq l \ \wedge \ 3 \leq r.$$

The upper and lower bounds of t are missing in the sliced result, because the true and false branches of the following conditional statement assign l and i to t, and t is not used before these assignments. The upper bounds of l and r are also missing, even though they are marked important at n_5 and n_{10}. The reason is that the conditions $l \leq 9$ and $r \leq 9$ of the following two conditional statements provide strong enough upper bounds for l and r. These bounds are even tighter than $l \leq 18$ and $r \leq 19$ in the invariants, respectively. □

		n_1	n_4	n_5	n_6	n_9	n_{12}
Initial extractor annotation	i	{}	{}	{l, u}	{}	{}	{l, u}
	l	{}	{}	{l, u}	{}	{}	{}
	r	{}	{}	{}	{}	{}	{}
	t	{}	{}	{}	{}	{}	{l, u}
Abstract interpretation result	i	$[1, 9]$	$[1, 9]$	$[1, 9]$	$[1, 9]$	$[1, 9]$	$[1, 9]$
	l	$[1, 18]$	$[2, 18]$	$[2, 9]$	$[2, 9]$	$[2, 18]$	$[2, 18]$
	r	$[1, 19]$	$[3, 19]$	$[3, 19]$	$[3, 19]$	$[3, 19]$	$[3, 19]$
	t	$[1, 9]$	$[1, 9]$	$[1, 9]$	$[2, 9]$	$[1, 9]$	$[1, 9]$

(a) Input Arguments for Abstract-value Slicer

		n_1	n_4	n_5	n_6	n_9	n_{12}
Result extractor from abstract-value slicer	i	{}	{l, u}	{l, u}	{l, u}	{l, u}	{l, u}
	l	{}	{l}	{l, u}	{}	{}	{}
	r	{}	{l}	{l}	{l}	{l}	{}
	t	{l, u}	{}	{}	{l, u}	{l, u}	{l, u}
Sliced abstract interpretation result	i	$[-\infty, \infty]$	$[1, 9]$	$[1, 9]$	$[1, 9]$	$[1, 9]$	$[1, 9]$
	l	$[-\infty, \infty]$	$[2, \infty]$	$[2, 9]$	$[-\infty, \infty]$	$[-\infty, \infty]$	$[-\infty, \infty]$
	r	$[-\infty, \infty]$	$[3, \infty]$	$[3, \infty]$	$[3, \infty]$	$[3, \infty]$	$[-\infty, \infty]$
	t	$[1, 9]$	$[-\infty, \infty]$	$[-\infty, \infty]$	$[2, 9]$	$[1, 9]$	$[1, 9]$

(b) Results from Abstract-value Slicer

	n_1	n_4	n_5	n_6	n_9	n_{12}
Before slicing	$1 \leq i \leq 9$ $\wedge 1 \leq l \leq 18$ $\wedge 1 \leq r \leq 19$ $\wedge 1 \leq t \leq 9$	$1 \leq i \leq 9$ $\wedge 2 \leq l \leq 18$ $\wedge 3 \leq r \leq 19$ $\wedge 1 \leq t \leq 9$	$1 \leq i \leq 9$ $\wedge 2 \leq l \leq 9$ $\wedge 3 \leq r \leq 19$ $\wedge 1 \leq t \leq 9$	$1 \leq i \leq 9$ $\wedge 2 \leq l \leq 9$ $\wedge 3 \leq r \leq 19$ $\wedge 2 \leq t \leq 9$	$1 \leq i \leq 9$ $\wedge 2 \leq l \leq 18$ $\wedge 3 \leq r \leq 19$ $\wedge 1 \leq t \leq 9$	$1 \leq i \leq 9$ $\wedge 2 \leq l \leq 18$ $\wedge 3 \leq r \leq 19$ $\wedge 1 \leq t \leq 9$
After slicing	$1 \leq t \leq 9$	$1 \leq i \leq 9$ $\wedge 2 \leq l$ $\wedge 3 \leq r$	$1 \leq i \leq 9$ $\wedge 2 \leq l \leq 9$ $\wedge 3 \leq r$	$1 \leq i \leq 9$ $\wedge 3 \leq r$ $\wedge 2 \leq t \leq 9$	$1 \leq i \leq 9$ $\wedge 3 \leq r$ $\wedge 1 \leq t \leq 9$	$1 \leq i \leq 9$ $\wedge 1 \leq t \leq 9$

(c) Results as Constraints

Figure 12: Result of Abstract-value Slicing for **simple_max_heapify**(Analyzed by the Interval Analysis)

$$\mathcal{V} \stackrel{\text{def}}{=} \mathsf{Parity} = \wp(\{\mathsf{even}, \mathsf{odd}\}) \qquad v \sqsubseteq v \stackrel{\text{def}}{\Leftrightarrow} v \subseteq v' \qquad \bot \stackrel{\text{def}}{=} \{\} \qquad v \sqcup v' \stackrel{\text{def}}{=} v \cup v'$$

$$\hat{\mathsf{leq}}(n)(v) \stackrel{\text{def}}{=} v \qquad \mathsf{abs}(n) \stackrel{\text{def}}{=} \textbf{if } (n \text{ is even}) \textbf{ then } \{\mathsf{even}\} \textbf{ else } \{\mathsf{odd}\}$$

$$\hat{\mathsf{geq}}(n)(v) \stackrel{\text{def}}{=} v \qquad \hat{\mathsf{db}}(v) \stackrel{\text{def}}{=} \textbf{if } (v = \{\}) \textbf{ then } \{\} \textbf{ else } \{\mathsf{even}\}$$

$$p_0 \in (v \hat{+} v') \stackrel{\text{def}}{\Leftrightarrow} \Big(\exists p, p'.\ p \in v \wedge p' \in v' \wedge \big((p_0 = \mathsf{even} \wedge p = p') \vee (p_0 = \mathsf{odd} \wedge p \neq p')\big)\Big)$$

Figure 13: Parameters for Parity Analysis

Example 15 The second example is an analysis, called parity analysis, that discovers the parity of program variables. Parity analysis is an instance of Cousot's generic abstract semantics with the parameters in Figure 13.

In order to obtain an abstract-value slicer for parity analysis, we instantiate the generic abstract-value slicer with the parameters in Figure 14. The parameters are chosen so that the resulting slicer works on the level of variables: at each node in a program, the slicer identifies a set of variables whose abstract values are necessary for the verification. More specifically, $\mathcal{S}_v$ in the first parameter consists of p and np, which respectively mean picking and

The first parameter is given below:

$$\mathcal{S}_p \stackrel{\text{def}}{=} \{\mathsf{p}, \mathsf{np}\} \qquad s \sqsubseteq s' \stackrel{\text{def}}{\Leftrightarrow} s = \mathsf{p} \vee s' = \mathsf{np} \qquad \mathsf{vex}_v(s) \stackrel{\text{def}}{=} \textbf{if } (s = \mathsf{p}) \textbf{ then } v \textbf{ else } \{\mathsf{even}, \mathsf{odd}\}$$

Let $\mathsf{dTop}(v)$ be the following function of type $\mathcal{S}_v \to \mathcal{S}_v$:

$$\lambda s.\, \textbf{if } (v = \{\mathsf{even}, \mathsf{odd}\}) \textbf{ then } \mathsf{np} \textbf{ else } s.$$

The second parameter is given below:

$$\begin{aligned}
\mathsf{bleq}(n, v, w)(s) &\stackrel{\text{def}}{=} \mathsf{dTop}(w)(s) \\
\mathsf{bgeq}(n, v, w)(s) &\stackrel{\text{def}}{=} \mathsf{dTop}(w)(s) \\
\mathsf{bord}(v, w)(s) &\stackrel{\text{def}}{=} \mathsf{dTop}(w)(s) \\
\mathsf{bplus}(u, v, w)(s) &\stackrel{\text{def}}{=} \textbf{if } (u = \{\}) \\
&\qquad \textbf{then } (\mathsf{dTop}(w)(s),\ \mathsf{np}) \\
&\qquad \textbf{else } \Big(\textbf{if } (v{=}\{\}) \textbf{ then } (\mathsf{np},\ \mathsf{dTop}(w)(s)) \textbf{ else } (\mathsf{dTop}(w)(s),\ \mathsf{dTop}(w)(s))\Big) \\
\mathsf{bdb}(v, w)(s) &\stackrel{\text{def}}{=} \textbf{if } (w = \{\}) \textbf{ then } (\mathsf{dTop}(w)(s)) \textbf{ else } (\mathsf{np})
\end{aligned}$$

Figure 14: Parameters for an Abstract-value Slicer for Parity Analysis

dropping a variable. For instance, if an extractor $[x{\mapsto}\mathsf{p}, y{\mapsto}\mathsf{np}]$ is applied to an abstract state $[x{\mapsto}\{\mathsf{even}\}, y{\mapsto}\{\mathsf{odd}\}]$, which means $even(x) \wedge odd(y)$, then it results in $[x{\mapsto}\{\mathsf{even}\}, y{\mapsto}\{\mathsf{even}, \mathsf{odd}\}]$, which means $even(x)$.

Note that all the back-tracers in Figure 14 use the subroutine $\mathsf{dTop}(v)$ that checks whether it is worth selecting the value v (i.e., $v \neq \{\mathsf{even}, \mathsf{odd}\}$); if not, $\mathsf{dTop}(v)$ transforms its input s to np. Another thing to note is that in the definition of bplus, we impose the priority on the first argument; when both u and v are $\{\}$ and $\mathsf{dTop}(w)(s)$ is p, bplus decides to keep the first input and returns $(\mathsf{p}, \mathsf{np})$, even though keeping the second argument and lifting the first argument to $\{\mathsf{even}, \mathsf{odd}\}$ (by taking $(\mathsf{np}, \mathsf{p})$) also makes the analysis prove that w approximates the output values (i.e., $\{\mathsf{even}, \mathsf{odd}\} \hat{+} \{\} \sqsubseteq w$). □

We now prove that the back-tracers for atomic terms in the generic slicer satisfy the soundness requirements.

Lemma 5.1 *If $[\![E]\!]a \sqsubseteq v$, then we have that*

$$\forall s \in \mathcal{S}_v.\ ([\![E]\!] \circ \mathsf{ex}_a \circ (\!|E|\!)_{av})(s) \sqsubseteq \mathsf{vex}_v(s).$$

Proof: We prove the lemma by induction on the structure of E.

- **case** $E \equiv n$:

$$\begin{aligned}
&\Big([\![n]\!] \circ \mathsf{ex}_a \circ (\!|n|\!)_{av}\Big)(s) \\
&= \alpha(n) && \text{(by the definition of } [\![n]\!]) \\
&\sqsubseteq \mathsf{vex}_v(s) && \text{(since } \alpha(n) = [\![n]\!]a \sqsubseteq v \text{ and } v \sqsubseteq \mathsf{vex}_v(s)).
\end{aligned}$$

- **case** $E \equiv x$:

$$\begin{aligned}
&\Big([\![x]\!] \circ \mathsf{ex}_a \circ (\!|x|\!)_{av}\Big)(s) \\
&= (\mathsf{ex}_a \circ (\!|x|\!)_{av})(s)(x) && \text{(by the definition of } [\![x]\!])
\end{aligned}$$

$= \mathsf{ex}_a(\llparenthesis x \rrparenthesis_{av}(s))(x)$

$= \mathsf{vex}_{a(x)}(\llparenthesis x \rrparenthesis_{av}(s)(x))$ (by the definition of ex_a)

$= \mathsf{vex}_{a(x)}(\mathsf{bord}(a(x), v)(s))$
(since $\llparenthesis x \rrparenthesis_{av} = \lambda s.\lambda y.\mathbf{if}\ (x \equiv y)\ \mathbf{then}\ \mathsf{bord}(a(x), v)(s)\ \mathbf{else}\ \top$)

$\sqsubseteq \mathsf{vex}_v(s)$ (by the condition for bord).

- **case** $E \equiv E + E'$: Let w and w' be, respectively, $[\![E]\!]a$ and $[\![E']\!]a$. From the definition of $\llparenthesis E + E' \rrparenthesis_{av}$, it follows that

$$\begin{array}{lll} \llparenthesis E + E' \rrparenthesis_{av} & \sqsubseteq & \llparenthesis E \rrparenthesis_{aw} \circ \mathsf{fst} \circ \mathsf{bplus}(w, w', v) \quad \text{and} \\ \llparenthesis E + E' \rrparenthesis_{av} & \sqsubseteq & \llparenthesis E' \rrparenthesis_{aw'} \circ \mathsf{snd} \circ \mathsf{bplus}(w, w', v). \end{array}$$

  Using this inequality, we prove the required condition as follows:

$\Big([\![E + E']\!] \circ \mathsf{ex}_a \circ \llparenthesis E + E' \rrparenthesis_{av}\Big)(s)$

$= \Big([\![E]\!]\Big((\mathsf{ex}_a \circ \llparenthesis E + E' \rrparenthesis_{av})(s)\Big)\Big) \hat{+} \Big([\![E']\!]\Big((\mathsf{ex}_a \circ \llparenthesis E + E' \rrparenthesis_{av})(s)\Big)\Big)$
(by the definition of $[\![E + E']\!]$)

$= \Big(([\![E]\!] \circ \mathsf{ex}_a \circ \llparenthesis E + E' \rrparenthesis_{av})(s)\Big) \hat{+} \Big(([\![E']\!] \circ \mathsf{ex}_a \circ \llparenthesis E + E' \rrparenthesis_{av})(s)\Big)$

$\sqsubseteq \Big(([\![E]\!] \circ \mathsf{ex}_a \circ \llparenthesis E \rrparenthesis_{aw} \circ \mathsf{fst} \circ \mathsf{bplus}(w, w', v))(s)\Big) \hat{+} \Big(([\![E']\!] \circ \mathsf{ex}_a \circ \llparenthesis E + E' \rrparenthesis_{av})(s)\Big)$
(by the obtained inequality for $\llparenthesis E + E' \rrparenthesis_{av}$ and the monotonicity of $\hat{+}$, $[\![E]\!]$, ex_a)

$\sqsubseteq \Big((\mathsf{vex}_w \circ \mathsf{fst} \circ \mathsf{bplus}(w, w', v))(s)\Big) \hat{+} \Big(([\![E']\!] \circ \mathsf{ex}_a \circ \llparenthesis E + E' \rrparenthesis_{av})(s)\Big)$
(by the induction hypothesis and the monotonicity of $\hat{+}$)

$\sqsubseteq \Big((\mathsf{vex}_w \circ \mathsf{fst} \circ \mathsf{bplus}(w, w', v))(s)\Big) \hat{+} \Big((\mathsf{vex}_{w'} \circ \mathsf{snd} \circ \mathsf{bplus}(w, w', v))(s)\Big)$
(by the same reasoning as the left hand side expression)

$= (\hat{+} \circ (\mathsf{vex}_w \times \mathsf{vex}_{w'}) \circ \mathsf{bplus}(w, w', v))(s)$

$\sqsubseteq \mathsf{vex}_v(s)$ (by the soundness of bplus).

- **case** $E \equiv 2E'$: Let u be $[\![E']\!]a$.

$\Big([\![2E']\!] \circ \mathsf{ex}_a \circ \llparenthesis 2E' \rrparenthesis_{av}\Big)(s)$

$= \Big(\hat{\mathsf{db}} \circ [\![E']\!] \circ \mathsf{ex}_a \circ \llparenthesis E' \rrparenthesis_{au} \circ \mathsf{bdb}(u, v)\Big)(s)$ (by the definition of $[\![2E']\!]$ and $\llparenthesis 2E' \rrparenthesis$)

$\sqsubseteq (\hat{\mathsf{db}} \circ \mathsf{vex}_u \circ \mathsf{bdb}(u, v))(s)$ (by the induction hypothesis and the monotonicity of $\hat{\mathsf{db}}$)

$\sqsubseteq \mathsf{vex}_v(s)$ (by the soundness of bdb).

□

Lemma 5.2 (Correctness) *If $[\![t]\!]a \sqsubseteq b$, then we have that*

$$\forall e \in \mathcal{S}_b.\ (([\![t]\!] \circ \mathsf{ex}_a \circ \llparenthesis t \rrparenthesis_{ab})(e) \ \sqsubseteq\ \mathsf{ex}_b(e).$$

Proof: An atomic term t is one of $x{:=}E$, $x{\leq}n$, $n{\leq}x$, and $E{\leq}E'$. We deal with these four cases separately.

- **case** $\mathsf{t} \equiv x{:=}E$: Suppose that $[\![x := E]\!]a \sqsubseteq b$. We derive two inequalities, one for the variable x and the other for all the other variables y:

  $([\![E]\!] \circ \mathsf{ex}_a \circ \llparenthesis x := E \rrparenthesis_{ab})(e) \sqsubseteq \mathsf{vex}_{b(x)}(e(x))$ and $(\mathsf{ex}_a \circ \llparenthesis x := E \rrparenthesis_{ab})(e)(y) \sqsubseteq \mathsf{vex}_{b(y)}(e(y))$.

  The following derivation shows the inequality for y:

$$
\begin{aligned}
&(\mathsf{ex}_a \circ (\!|x := E|\!)_{ab})(e)(y)\\
&= \mathsf{vex}_{a(y)}\Big((\!|x := E|\!)_{ab}(e)(y)\Big) && \text{(by the definition of } \mathsf{ex}_a)\\
&= \mathsf{vex}_{a(y)}\Big(\Big(\mathsf{bord}(a(y), b(y))(e(y))\Big) \sqcap \Big((\!|E|\!)_{a(b(x))}(e(x))(y)\Big)\Big)\\
&\qquad\qquad \text{(by the definition of } (\!|x := E|\!)_{ab}(e)(y))\\
&\sqsubseteq \mathsf{vex}_{a(y)}\Big(\mathsf{bord}(a(y), b(y))(e(y))\Big) && \text{(by the monotonicity of } \mathsf{vex}_{a(y)})\\
&\sqsubseteq \mathsf{vex}_{b(y)}(e(y)) && \text{(by the soundness of } \mathsf{bord}).
\end{aligned}
$$

And the following derivation shows the inequality for x:

$$
\begin{aligned}
&([\![E]\!] \circ \mathsf{ex}_a \circ (\!|x := E|\!)_{ab})(e)\\
&= ([\![E]\!] \circ \mathsf{ex}_a)\Big((\!|x := E|\!)_{ab}(e)\Big)\\
&\sqsubseteq ([\![E]\!] \circ \mathsf{ex}_a)\Big((\!|E|\!)_{a(b(x))}(e(x))\Big)\\
&\qquad \text{(by the definition of } (\!|x := E|\!)_{ab} \text{ and the monotonicity of } [\![E]\!] \circ \mathsf{ex}_a\ )\\
&= \Big([\![E]\!] \circ \mathsf{ex}_a \circ (\!|E|\!)_{a(b(x))}\Big)(e(x))\\
&\sqsubseteq \mathsf{vex}_{b(x)}(e(x)) && \text{(by Lemma 5.1).}
\end{aligned}
$$

Using the obtained two inequalities, we prove the lemma for this case:

$$
\begin{aligned}
&[\![x{:=}E]\!]\Big((\mathsf{ex}_a \circ (\!|x{:=}E|\!)_{ab})\,(e)\Big)\\
&= \Big((\mathsf{ex}_a \circ (\!|x{:=}E|\!)_{ab})\,(e)\Big)\Big[x \mapsto [\![E]\!]\Big((\mathsf{ex}_a \circ (\!|x{:=}E|\!)_{ab})\,(e)\Big)\Big]\\
&\qquad \text{(by the definition of } [\![x := E]\!])\\
&= \lambda y.\ \textbf{if}\ (x \equiv y)\ \textbf{then}\ \Big([\![E]\!]\big((\mathsf{ex}_a \circ (\!|x{:=}E|\!)_{ab})\,(e)\big)\Big)\ \textbf{else}\ \Big((\mathsf{ex}_a \circ (\!|x{:=}E|\!)_{ab})(e)(y)\Big)\\
&\quad \text{(since the abstract environment } \eta[x \mapsto v] \text{ is equal to } \lambda y.\textbf{if}\ x \equiv y\ \textbf{then}\ v\ \textbf{else}\ \eta(y))\\
&\sqsubseteq \lambda y.\ \textbf{if}\ (x \equiv y)\ \textbf{then}\ \Big([\![E]\!]\big((\mathsf{ex}_a \circ (\!|x{:=}E|\!)_{ab})\,(e)\big)\Big)\ \textbf{else}\ \Big(\mathsf{vex}_{b(y)}(e(y))\Big)\\
&\qquad \text{(by the obtained inequality for } y)\\
&\sqsubseteq \lambda y.\ \textbf{if}\ (x \equiv y)\ \textbf{then}\ \Big(\mathsf{vex}_{b(x)}(e(x))\Big)\ \textbf{else}\ \Big(\mathsf{vex}_{b(y)}(e(y))\Big)\\
&\qquad \text{(by the obtained inequality for } x)\\
&= \lambda y.\ \mathsf{vex}_{b(y)}(e(y))\\
&= \mathsf{ex}_b(e).
\end{aligned}
$$

- **case** $\mathtt{t} \equiv \mathtt{skip}$: Suppose that $[\![\mathtt{skip}]\!]a \sqsubseteq b$.

$$
\begin{aligned}
&([\![\mathtt{skip}]\!] \circ \mathsf{ex}_a \circ (\!|\mathtt{skip}|\!)_{ab})(e)\\
&= \mathsf{ex}_a((\!|\mathtt{skip}|\!)_{ab}(e)) && \text{(by the definition of } [\![\mathtt{skip}]\!])\\
&= \lambda x.\, \mathsf{vex}_{a(x)}\Big(\big((\!|\mathtt{skip}|\!)_{ab}(e)\big)(x)\Big) && \text{(by the definition of } \mathsf{ex}_a)\\
&= \lambda x.\, \mathsf{vex}_{a(x)}\Big(\mathsf{bord}(a(x), b(x))(e(x))\Big) && \text{(by the definition of } (\!|\mathtt{skip}|\!)_{ab})\\
&\sqsubseteq \lambda x.\, \mathsf{vex}_{b(x)}(e(x)) && \text{(by the soundness of } \mathsf{bord})\\
&= \mathsf{ex}_b(e).
\end{aligned}
$$

- **case** $\mathsf{t} \equiv x{\leq}n$ or $n{\leq}x$: When t is $x{\leq}n$, let k and k' be, respectively, $\hat{\mathsf{leq}}(n)$ and $\mathsf{bleq}(n)$, and when t is $n{\leq}x$, let k and k' be, respectively, $\hat{\mathsf{geq}}(n)$ and $\mathsf{bgeq}(n)$. Using the soundness of bleq and bgeq, we prove the following inequality:

$$
\begin{array}{ll}
& k\Big((\mathsf{ex}_a \circ (\!|t|\!)_{ab})(e)(x)\Big) \\
= & k\Big(\mathsf{vex}_{a(x)}\Big((\!|t|\!)_{ab}(e)(x)\Big)\Big) \qquad \text{(by the definition of } \mathsf{ex}_a) \\
= & k\Big(\mathsf{vex}_{a(x)}\Big(k'(a(x),b(x))(e(x))\Big)\Big) \qquad \text{(by the definition of } (\!|t|\!)_{ab}) \\
= & \Big(k \circ \mathsf{vex}_{a(x)} \circ k'(a(x),b(x))\Big)(e(x)) \\
\sqsubseteq & \mathsf{vex}_{b(x)}(e(x)) \qquad \text{(by the soundness of } \mathsf{bleq} \text{ and } \mathsf{bgeq}).
\end{array}
$$

  Using the proved inequality, we prove the lemma as follows:

$$
\begin{array}{ll}
& (\llbracket t \rrbracket \circ \mathsf{ex}_a \circ (\!|t|\!)_{ab})(e) \\
= & \Big((\mathsf{ex}_a \circ (\!|t|\!)_{ab})(e)\Big)\Big[x{\mapsto}k\Big((\mathsf{ex}_a \circ (\!|t|\!)_{ab})(e)(x)\Big)\Big] \qquad \text{(by the definition of } \llbracket t \rrbracket) \\
\sqsubseteq & \Big((\mathsf{ex}_a \circ (\!|t|\!)_{ab})(e)\Big)[x{\mapsto}\mathsf{vex}_{b(x)}(e(x))] \qquad \text{(by the obtained inequality)} \\
= & \lambda y.\ \textbf{if}\ (x \equiv y)\ \textbf{then}\ \big(\mathsf{vex}_{b(x)}(e(x))\big)\ \textbf{else}\ \big((\mathsf{ex}_a \circ (\!|t|\!)_{ab})(e)(y)\big) \\
& \qquad \text{(since the abstract environment } \eta[x{\mapsto}v] \text{ is equal to } \lambda y.\textbf{if}\ x \equiv y\ \textbf{then}\ v\ \textbf{else}\ \eta(y)) \\
= & \lambda y.\ \textbf{if}\ (x \equiv y)\ \textbf{then}\ \big(\mathsf{vex}_{b(x)}(e(x))\big)\ \textbf{else}\ \big(\mathsf{vex}_{a(y)}((\!|t|\!)_{ab}(e)(y))\big) \qquad \text{(by the definition of } \mathsf{ex}_a) \\
= & \lambda y.\ \textbf{if}\ (x \equiv y)\ \textbf{then}\ \big(\mathsf{vex}_{b(x)}(e(x))\big)\ \textbf{else}\ \Big(\mathsf{vex}_{a(y)}\Big(\mathsf{bord}(a(y),b(y))(e(y))\Big)\Big) \\
& \qquad \text{(by the definition of } (\!|t|\!)_{ab}) \\
\sqsubseteq & \lambda y.\ \textbf{if}\ (x \equiv y)\ \textbf{then}\ \big(\mathsf{vex}_{b(x)}(e(x))\big)\ \textbf{else}\ \big(\mathsf{vex}_{b(y)}(e(y))\big) \qquad \text{(by the condition for } \mathsf{bord}) \\
= & \mathsf{ex}_b e.
\end{array}
$$

- **case** $\mathsf{t} \equiv E{\leq}E'$: Suppose that $\llbracket E{\leq}E' \rrbracket a \sqsubseteq b$.

$$
\begin{array}{ll}
& (\llbracket E{\leq}E' \rrbracket \circ \mathsf{ex}_a \circ (\!|E{\leq}E'|\!)_{ab})(e) \\
= & (\mathsf{ex}_a \circ (\!|E{\leq}E'|\!)_{ab})(e) \qquad \text{(by the definition of } \llbracket E{\leq}E' \rrbracket) \\
= & \lambda y.\ \mathsf{vex}_{a(y)}\big((\!|E{\leq}E'|\!)_{ab}(e)(y)\big) \qquad \text{(by the definition of } \mathsf{ex}_a) \\
= & \lambda y.\ \mathsf{vex}_{a(y)}\big(\mathsf{bord}(a(y),b(y))(e(y))\big) \qquad \text{(by the definition of } (\!|E{\leq}E'|\!)_{ab}) \\
\sqsubseteq & \lambda y.\ \mathsf{vex}_{b(y)}(e(y)) \qquad \text{(by the condition for } \mathsf{bord}) \\
= & \mathsf{ex}_b e.
\end{array}
$$

□

5.2 Slicer for Product Abstract Interpretations

Intuitively, the cartesian product of abstract interpretations $(\mathcal{A}, \llbracket - \rrbracket^{\mathcal{A}})$ and $(\mathcal{B}, \llbracket - \rrbracket^{\mathcal{B}})$ is the "parallel execution" of these abstract interpretations [CC79, CC92]. Formally, it is defined as follows:

$$\mathcal{C} \stackrel{\text{def}}{=} \mathcal{A} \times \mathcal{B} \qquad \llbracket t \rrbracket^{\mathcal{C}} \stackrel{\text{def}}{=} \llbracket t \rrbracket^{\mathcal{A}} \times \llbracket t \rrbracket^{\mathcal{B}}.$$

Note that the cartesian product does not provide any additional advantage over the "sequential composition" that runs first $(\mathcal{A}, \llbracket - \rrbracket^{\mathcal{A}})$ and then $(\mathcal{B}, \llbracket - \rrbracket^{\mathcal{B}})$ for each input program. Thus, the main value of this cartesian product construction does not lie in the improvement of

existing analyses. Rather, it is to prepare existing analyses so that other analysis-improvement techniques, such as reduction, can be easily applicable.

An abstract-value slicer for this product abstract interpretation $(\mathcal{C}, [\![-]\!]^{\mathcal{C}})$ can be systematically constructed from abstract-value slicers for $(\mathcal{A}, [\![-]\!]^{\mathcal{A}})$ and $(\mathcal{B}, [\![-]\!]^{\mathcal{B}})$. Let $\mathfrak{E} = (\{(\mathcal{E}_a, \mathsf{ex}^{\mathcal{E}}_a)\}_{a \in \mathcal{A}}, (\!|-|\!)^{\mathcal{E}})$ and $\mathfrak{F} = (\{(\mathcal{F}_b, \mathsf{ex}^{\mathcal{F}}_b)\}_{b \in \mathcal{B}}, (\!|-|\!)^{\mathcal{F}})$ be abstract-value slicers for $(\mathcal{A}, [\![-]\!]^{\mathcal{A}})$ and $(\mathcal{B}, [\![-]\!]^{\mathcal{B}})$, respectively. From these slicers, we construct an abstract-value slicer $\mathfrak{H}$ for $(\mathcal{C}, [\![-]\!]^{\mathcal{C}})$ as follows:

$$\mathcal{H}_{(a,b)} \stackrel{\text{def}}{=} \mathcal{E}_a \times \mathcal{F}_b \qquad \mathsf{ex}^{\mathcal{H}}_{(a,b)} \stackrel{\text{def}}{=} \mathsf{ex}^{\mathcal{E}}_a \times \mathsf{ex}^{\mathcal{F}}_b \qquad (\!|t|\!)^{\mathcal{H}}_{(a,b)(a',b')} \stackrel{\text{def}}{=} (\!|t|\!)^{\mathcal{E}}_{aa'} \times (\!|t|\!)^{\mathcal{F}}_{bb'}$$

In this definition, we use the cartesian product $f \times g$ of functions f and g, which is defined to be $\lambda(x,y).(f(x), g(y))$. Note that the constructed slicer $\mathfrak{H}$ is the parallel composition of the component slicers, just like the cartesian product $(\mathcal{C}, [\![-]\!]^{\mathcal{C}})$.

The slicer $\mathfrak{H}$ satisfies all the requirements in the definition of the abstract-value slicers. The requirements for the extractor domain $\{(\mathcal{H}_{(a,b)}, \mathsf{ex}^{\mathcal{H}}_{(a,b)})\}_{(a,b) \in \mathcal{C}}$ follow from the preservation property of cartesian product: the cartesian product of two finite lattices is again a finite lattice, and the cartesian product of functions preserves the monotonicity and the lower bound. Since $\mathcal{H}_{(a,b)}$ is a cartesian product of two finite lattices, it is also a finite lattice, and since $\mathsf{ex}^{\mathcal{H}}_{(a,b)}$ is the cartesian product of two monotone functions with lower bounds, respectively, a and b, it is also monotone and has a lower bound (a,b) (i.e., $\forall (e_0, e_1) \in \mathcal{H}_{(a,b)}.\ \mathsf{ex}^{\mathcal{H}}_{(a,b)}(e_0, e_1) \sqsupseteq (a,b)$). The other requirement for back-tracers follows from the fact that the cartesian product of functions commutes with function composition: $(f \times g) \circ (f' \times g') = (f \circ f') \times (g \circ g')$. We prove this requirement in the lemma below:

Lemma 5.3 (Correctness of Back-tracers) *Let $(a,b), (a',b')$ be abstract values of the product abstract interpretation $(\mathcal{C}, [\![-]\!]^{\mathcal{C}})$. If $[\![t]\!]^{\mathcal{C}}(a,b) \sqsubseteq (a',b')$, then we have that*

$$\forall (e_0, e_1) \in \mathcal{H}_{(a',b')}.\quad \left([\![t]\!]^{\mathcal{C}} \circ \mathsf{ex}^{\mathcal{H}}_{(a,b)} \circ (\!|t|\!)^{\mathcal{H}}_{(a,b)(a',b')}\right)(e_0, e_1) \quad \sqsubseteq \quad \mathsf{ex}^{\mathcal{H}}_{(a',b')}(e_0, e_1)$$

Proof: We prove the lemma below:

$$\begin{aligned}
&\left([\![t]\!]^{\mathcal{C}} \circ \mathsf{ex}^{\mathcal{H}}_{(a,b)} \circ (\!|t|\!)^{\mathcal{H}}_{(a,b)(a',b')}\right)(e_0, e_1) \\
&= \left(([\![t]\!]^{\mathcal{A}} \times [\![t]\!]^{\mathcal{B}}) \circ (\mathsf{ex}^{\mathcal{E}}_a \times \mathsf{ex}^{\mathcal{F}}_b) \circ ((\!|t|\!)^{\mathcal{E}}_{aa'} \times (\!|t|\!)^{\mathcal{F}}_{bb'})\right)(e_0, e_1) \\
&\qquad\qquad \text{(by the definition of } [\![-]\!]^{\mathcal{C}}, \mathsf{ex}^{\mathcal{H}}, \text{ and } (\!|t|\!)^{\mathcal{H}}) \\
&= \left(([\![t]\!]^{\mathcal{A}} \circ \mathsf{ex}^{\mathcal{E}}_a \circ (\!|t|\!)^{\mathcal{E}}_{aa'}) \times ([\![t]\!]^{\mathcal{B}} \circ \mathsf{ex}^{\mathcal{F}}_b \circ (\!|t|\!)^{\mathcal{F}}_{bb'})\right)(e_0, e_1) \\
&\qquad\qquad \text{(since } (f \times g) \circ (f' \times g') = (f \circ f') \times (g \circ g')) \\
&\sqsubseteq \left(\mathsf{ex}^{\mathcal{E}}_{a'} \times \mathsf{ex}^{\mathcal{F}}_{b'}\right)(e_0, e_1) \qquad \text{(by the correctness of } (\!|-|\!)^{\mathcal{E}} \text{ and } (\!|-|\!)^{\mathcal{F}}) \\
&= \mathsf{ex}^{\mathcal{H}}_{(a',b')}(e_0, e_1).
\end{aligned}$$

□

5.3 Slicer for Reduced Abstract Interpretations

The reduction is a well-known technique for improving the accuracy of an abstract interpretation [CC79, CC92]. Let $(\mathcal{A}, [\![-]\!]^{\mathcal{A}})$ be an abstract interpretation, and let ρ be a lower closure operator on $\mathcal{A}$, i.e., ρ is a monotone function that is idempotent ($\rho \circ \rho = \rho$) and reductive

		n_1	n_4	n_5	n_6	n_9	n_{12}
Initial extractor annotation	i	(np,{})	(np,{})	(np,{l, u})	(np,{})	(np,{})	(np,{l, u})
	l	(np,{})	(np,{})	(np,{l, u})	(np,{})	(np,{})	(np,{})
	r	(np,{})	(np,{})	(np,{})	(np,{})	(np,{})	(np,{})
	t	(np,{})	(np,{})	(np,{})	(np,{})	(np,{})	(np,{l, u})
Abstract interpretation result	i	(al, [1, 9])	(al, [1, 9])	(al, [1, 9])	(al, [1, 9])	(al, [1, 9])	(al, [1, 9])
	l	(al, [1, 18])	(ev, [2, 18])	(ev, [2, 8])	(ev, [2, 8])	(ev, [2, 18])	(ev, [2, 18])
	r	(od,[1, 19])	(od,[3, 19])	(od,[3, 19])	(od,[3, 19])	(od,[3, 19])	(od,[3, 19])
	t	(al, [1, 9])	(al, [1, 9])	(al, [1, 9])	(ev, [2, 8])	(al, [1, 9])	(al, [1, 9])

(a) Input Arguments for Abstract-value Slicer

		n_1	n_4	n_5	n_6	n_9	n_{12}
Result extractors from abstract-value slicer	i	(np,{})	(np,{l, u})	(np,{l, u})	(np,{l, u})	(np,{l, u})	(np,{l, u})
	l	(np,{})	(p, {l})	(np,{l, u})	(np,{})	(np,{})	(np,{})
	r	(np,{})	(np,{l})	(np,{l})	(np,{l})	(np,{l})	(np,{})
	t	(np,{l, u})	(np,{})	(np,{})	(np,{l, u})	(np,{l, u})	(np,{l, u})
Sliced abstract interpretation result	i	(al,[±∞])	(al, [1, 9])	(al,[1, 9])	(al,[1, 9])	(al,[1, 9])	(al,[1, 9])
	l	(al,[±∞])	(ev,[2, ∞])	(al,[2, 8])	(al,[±∞])	(al,[±∞])	(al,[±∞])
	r	(al,[±∞])	(al, [3, ∞])	(al,[3, ∞])	(al,[3, ∞])	(al,[3, ∞])	(al,[±∞])
	t	(al,[1, 9])	(al, [±∞])	(al,[±∞])	(al,[2, 8])	(al,[1, 9])	(al,[1, 9])

(b) Results from Abstract-value Slicer

	n_1	n_4	n_5	n_6	n_9	n_{12}
Before slicing	$1{\leq}i{\leq}9$ $\wedge 1{\leq}l{\leq}18$ $\wedge 1{\leq}r{\leq}19$ $\wedge 1{\leq}t{\leq}9$ $\wedge odd(r)$	$1{\leq}i{\leq}9$ $\wedge 2{\leq}l{\leq}18$ $\wedge 3{\leq}r{\leq}19$ $\wedge 1{\leq}t{\leq}9$ $\wedge even(l)$ $\wedge odd(r)$	$1{\leq}i{\leq}9$ $\wedge 2{\leq}l{\leq}8$ $\wedge 3{\leq}r{\leq}19$ $\wedge 1{\leq}t{\leq}9$ $\wedge even(l)$ $\wedge odd(r)$	$1{\leq}i{\leq}9$ $\wedge 2{\leq}l{\leq}8$ $\wedge 3{\leq}r{\leq}19$ $\wedge 2{\leq}t{\leq}8$ $\wedge even(l)$ $\wedge odd(r)$ $\wedge even(t)$	$1{\leq}i{\leq}9$ $\wedge 2{\leq}l{\leq}18$ $\wedge 3{\leq}r{\leq}19$ $\wedge 1{\leq}t{\leq}9$ $\wedge even(l)$ $\wedge odd(r)$	$1{\leq}i{\leq}9$ $\wedge 2{\leq}l{\leq}18$ $\wedge 3{\leq}r{\leq}19$ $\wedge 1{\leq}t{\leq}9$ $\wedge even(l)$ $\wedge odd(r)$
After slicing	$1{\leq}t{\leq}9$	$1{\leq}i{\leq}9$ $\wedge 2{\leq}l$ $\wedge 3{\leq}r$ $\wedge even(l)$	$1{\leq}i{\leq}9$ $\wedge 2{\leq}l{\leq}8$ $\wedge 3{\leq}r$	$1{\leq}i{\leq}9$ $\wedge 3{\leq}r$ $\wedge 2{\leq}t{\leq}8$	$1{\leq}i{\leq}9$ $\wedge 3{\leq}r$ $\wedge 1{\leq}t{\leq}9$	$1{\leq}i{\leq}9$ $\wedge 1{\leq}t{\leq}9$

(c) Results as Constraints

where $\mathsf{al}, \mathsf{ev}, \mathsf{od}$ and $[\pm\infty]$ are respectively $\{\mathsf{even}, \mathsf{odd}\}, \{\mathsf{even}\}, \{\mathsf{odd}\}$ and $[-\infty, \infty]$.

Figure 15: Result of Abstract-value Slicing for **simple_max_heapify** (Analyzed by the Reduced Product of the Parity and Interval Analyses)

($\rho \sqsubseteq \mathsf{id}$). The reduction of $(\mathcal{A}, [\![-]\!]^{\mathcal{A}})$ by ρ is an abstract interpretation defined as follows:[12]

$$\mathcal{B} \stackrel{\text{def}}{=} \{\rho(a) \mid a \in \mathcal{A}\}$$
$$\sqsubseteq^{\mathcal{B}} \stackrel{\text{def}}{=} \sqsubseteq^{\mathcal{A}} \qquad \bot^{\mathcal{B}} \stackrel{\text{def}}{=} \bot^{\mathcal{A}} \qquad \sqcup^{\mathcal{B}} \stackrel{\text{def}}{=} \sqcup^{\mathcal{A}} \qquad [\![t]\!]^{\mathcal{B}} \stackrel{\text{def}}{=} \lambda b \in \mathcal{B}.\ (\rho \circ [\![t]\!]^{\mathcal{A}})(b)$$

The key of this construction is the lower closure operator ρ, which is used here to improve the accuracy of the "abstract execution" $[\![t]\!]^{\mathcal{A}}$ of atomic terms t. Intuitively, ρ transforms an abstract value a to a' such that a' means the same state set as a, but it has a better "representation" than a (i.e., $a' \sqsubseteq a$). Thus, the new "abstract execution" $[\![t]\!]^{\mathcal{B}}$ is more precise than the old $[\![t]\!]^{\mathcal{A}}$, and so, the reduced abstract interpretation usually produces a more precise analysis result.

[12] Cousot and Cousot have shown that the reduction always produces a correct abstract interpretation: for every reduced abstract interpretation, its abstract domain $\mathcal{B}$ is a join semi-lattice, and its abstract semantics $[\![t]\!]^{\mathcal{B}}$ is monotone [CC79].

We construct an abstract-value slicer for the reduced abstract interpretation $(\mathcal{B}, [\![-]\!]^{\mathcal{B}})$, using an abstract-value slicer for $(\mathcal{A}, [\![-]\!]^{\mathcal{A}})$ and the *back-tracer* for ρ. Let $\mathfrak{E} = (\{(\mathcal{E}_a, \mathsf{ex}^{\mathcal{E}}_a)\}_{a\in\mathcal{A}}, (\!|-|\!)^{\mathcal{E}})$ be an abstract-value slicer for $(\mathcal{A}, [\![-]\!]^{\mathcal{A}})$. The *back-tracer* β for ρ in $\{(\mathcal{E}_a, \mathsf{ex}^{\mathcal{E}}_a)\}_{a\in\mathcal{A}}$ is a function parameterized by the "computations" of ρ: for all abstract values a, b with $\rho(a) \sqsubseteq b$, β_{ab} is a function from $\mathcal{E}_b$ to $\mathcal{E}_a$ such that

$$\forall e \in \mathcal{E}_b.\ (\rho \circ \mathsf{ex}^{\mathcal{E}}_a \circ \beta_{ab})(e) \ \sqsubseteq\ \mathsf{ex}^{\mathcal{E}}_b(e).$$

Intuitively, $\beta_{ab}(e)$ computes the part of a that ρ used to calculate the e-part of the output b. When such back-tracer β for ρ is given, we can construct an abstract-value slicer $\mathfrak{F}$ for $(\mathcal{B}, [\![-]\!]^{\mathcal{B}})$ as follows:

$$\mathcal{F}_a \stackrel{\text{def}}{=} \mathcal{E}_a \qquad \mathsf{ex}^{\mathcal{F}}_a \stackrel{\text{def}}{=} (\rho \circ \mathsf{ex}^{\mathcal{E}}_a) \qquad (\!|t|\!)^{\mathcal{F}}_{ab} \stackrel{\text{def}}{=} ((\!|t|\!)^{\mathcal{E}}_{a([\![t]\!]^{\mathcal{A}}a)} \circ \beta_{([\![t]\!]^{\mathcal{A}}a)b})$$

The constructed slicer $\mathfrak{F}$ uses the same extractor lattices $\mathcal{E}_a$ of the given slicer $\mathfrak{E}$, but for the other parts of $\mathfrak{E}$, it modifies them slightly. First, it adds the post processor ρ to the extractor applications of $\mathfrak{E}$. Even when a belongs to $\mathcal{B}$, the extracted value $\mathsf{ex}^{\mathcal{E}}_a(e)$ might not be in $\mathcal{B}$. The additional post-processing of the constructed $\mathsf{ex}^{\mathcal{F}}_a$ fixes such "type errors", because it ensures that the extracted abstract values belong to the correct domain $\mathcal{B}$. Second, $\mathfrak{F}$ inserts β to the back-tracer for atomic terms in $\mathfrak{E}$; the inserted β lets the slicer $\mathfrak{F}$ back-trace the application of closure ρ.

The constructed $\mathfrak{F}$ satisfies all the requirements in the definition of the abstract-value slicers. It is straightforward to show that the defined $\{(\mathcal{F}_a, \mathsf{ex}^{\mathcal{F}}_a)\}_{a\in\mathcal{A}}$ satisfies the requirements for the extractor domain. In the below lemma, we prove that the requirement for the back-tracer also holds.

Lemma 5.4 (Correctness) *For all atomic terms t and abstract values a, b, if $[\![t]\!]^{\mathcal{B}}a \sqsubseteq b$, then*

$$\forall e \in \mathcal{F}_b.\ \left([\![t]\!]^{\mathcal{B}} \circ \mathsf{ex}^{\mathcal{F}}_a \circ (\!|t|\!)^{\mathcal{F}}_{ab}\right)(e) \ \sqsubseteq\ \mathsf{ex}^{\mathcal{F}}_b(e).$$

Proof: We prove the lemma as follows:

$$\begin{aligned}
&\left([\![t]\!]^{\mathcal{B}} \circ \mathsf{ex}^{\mathcal{F}}_a \circ (\!|t|\!)^{\mathcal{F}}_{ab}\right)(e) \\
&= \left(\left(\rho \circ [\![t]\!]^{\mathcal{A}}\right) \circ \left(\rho \circ \mathsf{ex}^{\mathcal{E}}_a\right) \circ \left((\!|t|\!)^{\mathcal{E}}_{a([\![t]\!]^{\mathcal{A}}a)} \circ \beta_{([\![t]\!]^{\mathcal{A}}a)b}\right)\right)(e) && \text{(by the definition of } [\![t]\!]^{\mathcal{B}}, \mathsf{ex}^{\mathcal{F}}_a, \text{ and } (\!|t|\!)^{\mathcal{F}}_{ab}) \\
&\sqsubseteq \left(\rho \circ [\![t]\!]^{\mathcal{A}} \circ \mathsf{ex}^{\mathcal{E}}_a \circ (\!|t|\!)^{\mathcal{E}}_{a([\![t]\!]^{\mathcal{A}}a)} \circ \beta_{([\![t]\!]^{\mathcal{A}}a)b}\right)(e) && \text{(by } \rho \sqsubseteq \mathsf{id} \text{ and the monotonicity of } \rho \circ [\![t]\!]^{\mathcal{A}}) \\
&\sqsubseteq \left(\rho \circ \mathsf{ex}^{\mathcal{E}}_{([\![t]\!]^{\mathcal{A}}a)} \circ \beta_{([\![t]\!]^{\mathcal{A}}a)b}\right)(e) && \text{(by the correctness of } (\!|t|\!)^{\mathcal{E}} \text{ and the monotonicity of } \rho) \\
&= \left(\rho \circ \rho \circ \mathsf{ex}^{\mathcal{E}}_{([\![t]\!]^{\mathcal{A}}a)} \circ \beta_{([\![t]\!]^{\mathcal{A}}a)b}\right)(e) && \text{(since } \rho \text{ is idempotent)} \\
&\sqsubseteq (\rho \circ \mathsf{ex}^{\mathcal{E}}_b)(e) && \text{(by the soundness of } \beta) \\
&= \mathsf{ex}^{\mathcal{F}}_b(e) && \text{(by the definition of } \mathsf{ex}^{\mathcal{F}}_b).
\end{aligned}$$

□

Example 16 In this example, we apply the reduction to the cartesian product of parity analysis in Example 15 and interval analysis in Example 14. Let $(\mathcal{P}, [\![-]\!]^{\mathcal{P}})$ and $(\mathcal{I}, [\![-]\!]^{\mathcal{I}})$ be the abstract interpretations for parity and interval analyses, respectively. Our reduction is based on the following function prop:

$$\begin{aligned}
\mathsf{prop} &: \mathsf{Parity} \times \mathsf{Interval} \to \mathsf{Interval} \\
\mathsf{prop}(\{\mathsf{even}, \mathsf{odd}\}, [n, m]) &\stackrel{\text{def}}{=} [n, m] \\
\mathsf{prop}(\{\mathsf{even}\}, [n, m]) &\stackrel{\text{def}}{=} [\mathsf{pickEven}(n, n{+}1), \mathsf{pickEven}(m{-}1, m)] \\
\mathsf{prop}(\{\mathsf{odd}\}, [n, m]) &\stackrel{\text{def}}{=} [\mathsf{pickOdd}(n, n{+}1), \mathsf{pickOdd}(m{-}1, m)] \\
\mathsf{prop}(\{\}, [n, m]) &\stackrel{\text{def}}{=} [\infty, -\infty]
\end{aligned}$$

where pickEven selects an even number among two consecutive integers, and pickOdd chooses an odd number among two consecutive integers. Intuitively, the input $(p, [n, m])$ to prop means a set of integers that "satisfy" both p and $[n, m]$, and the function prop refines the input interval $[n, m]$, by eliminating some impossible integers, i.e., those that do not satisfy the first parity argument. For instance, when the first argument is $\{\mathsf{even}\}$, function prop adjusts the boundary values of the second argument so that they both become even integers.

We apply the reduction with the following lower closure operator ρ:

$$\begin{array}{rcl} \rho & : & \mathcal{P} \times \mathcal{I} \to \mathcal{P} \times \mathcal{I} \\ \rho(\pi,\ \iota) & \stackrel{\text{def}}{=} & (\pi,\ \lambda x.\mathsf{prop}(\pi(x), \iota(x))). \end{array}$$

Note that the resulting abstract interpretation performs parity analysis and interval analysis at the same time, and that during the analysis, it frequently improves the intermediate result from interval analysis, by propagating the (intermediate) result from parity analysis.

In order to construct an abstract-value slicer using the method in this section, we need to define the back-tracer β for the lower closure operator ρ. We first define the "back-tracer" bprop of prop: for all $(p, [n, m])$ and $[n', m']$ such that $\mathsf{prop}(p, [n, m]) \sqsubseteq [n', m']$,

$$\begin{array}{rcl} \mathsf{bprop}_{(p,[n,m])[n',m']} & : & \mathcal{S}_{[n',m']} \to \mathcal{S}_p \\ \mathsf{bprop}_{(p,[n,m])[n',m']}\, s & \stackrel{\text{def}}{=} & \textbf{if } ((n{<}n' \wedge \mathsf{l} \in s) \vee (m'{<}m \wedge \mathsf{u} \in s)) \textbf{ then } \mathsf{p} \textbf{ else } \mathsf{np} \end{array}$$

Note that function bprop checks whether the first input p to prop is used to improve the second argument; if so, function bprop says that we should keep track of the first argument (by taking p). Next, we define the back-tracer β for ρ: for all $(\pi, \iota), (\pi', \iota') \in \mathcal{P} \times \mathcal{I}$ such that $\rho(\pi, \iota) \sqsubseteq (\pi', \iota')$,

$$\begin{array}{rcl} \beta_{(\pi,\iota)(\pi',\iota')} & : & \mathcal{E}_{\pi'} \times \mathcal{E}_{\iota'} \to \mathcal{E}_{\pi} \times \mathcal{E}_{\iota} \\ \beta_{(\pi,\iota)(\pi',\iota')}(e_0, e_1) & \stackrel{\text{def}}{=} & \textbf{let } \left(e'_0 = \lambda x.\ (\mathsf{dTop}_{\pi'(x)} \circ e_0)(x)\right) \textbf{ and } \left(e'_1 = \lambda x.\ (\mathsf{dInf}_{\iota'(x)} \circ e_1)(x)\right) \\ & & \textbf{in } \left(e'_0 \sqcap \left(\lambda x.\ (\mathsf{bprop}_{(\pi(x),\iota(x))(\iota'(x))} \circ e'_1)(x)\right),\ \ e'_1\right) \end{array}$$

Function β works in three steps. First, it improves the "representation" of the input (e_0, e_1): it replaces (e_0, e_1) by (e'_0, e'_1) such that the new extractor (e'_0, e'_1) has the same effect as the old extractor (e_0, e_1), but $(e'_0, e'_1) \sqsupseteq (e_0, e_1)$. Second, β computes all the variables x where ρ has propagated the "interesting" information from $\pi(x)$ to $\iota(x)$: ρ has used $\pi(x)$ to make $\iota(x)$ more precise, and this improvement of accuracy is crucial for obtaining $\mathsf{vex}_{\iota'(x)}(e'_1(x))$. Finally, β incorporates into e'_0 all the computed variables in the second step; the resulting extractor picks all the variables in e'_0 as well as the computed variables in the second step.

We manually applied the abstract interpretation and the slicer in this example to the program in Figure 11, and obtained the data in Figure 15. The abstract interpretation result at six program points is shown in the second row of the first table (Figure 15(a)). Note that the interval parts of the analysis result are more precise than the invariants computed by interval analysis alone (Figure 12(a)); the reduced abstract interpretation computed better upper bounds for variables l and t at n_5 and n_6. This improvement is mainly due to the lower closure operator ρ, which propagated the information from parity analysis to interval analysis. For instance, the abstract execution of $l \leq 9$ from n_4 to n_5 first computes the interval $[2, 9]$ for l, but it immediately refines the computed interval to $[2, 8]$ using the fact that l is an even number at n_4.

We ran the slicer with the analysis result and the extractor annotation that directly indicates the verification goal, namely, the absence of array bounds error. The table in Figure 15(a) shows the input to the slicer at six program points, and the table in Figure 15(b) shows both the output from the slicer at those program points and the sliced analysis result by this output. Note that the abstract-value slicer correctly back-traces the information propagation by ρ at

n_5; it marks the parity of l at n_4 as necessary, because to compute the upper bound 8 of l at n_5, the abstract interpretation needs to know the fact that l is even at n_4. Another thing to note is that the parity of l is not tracked at n_5 by the slicer. This is because the analyzer does not use the parity of l any more in the following abstract computation. □

5.4 Slicer for the Cardinal Power of Abstract Interpretations

Consider abstract interpretations $(\mathcal{A}, [\![-]\!]^{\mathcal{A}})$ and $(\mathcal{B}, [\![-]\!]^{\mathcal{B}})$ with their meaning functions (i.e., concretizations) $\gamma^{\mathcal{A}}\colon \mathcal{A} \to \wp(\mathsf{States})$ and $\gamma^{\mathcal{B}}\colon \mathcal{B} \to \wp(\mathsf{States})$. Suppose that the domain $\mathcal{A}$ of the first abstract interpretation is finite, and that the abstract semantics $[\![t]\!]^{\mathcal{A}}$ of each atomic term t satisfies the following condition:

> for all abstract values $a_1 \in \mathcal{A}$, set $A_0 = \{a_0 \in \mathcal{A} \mid [\![t]\!]^{\mathcal{A}} a_0 \sqcap a_1 \neq \bot\}$ "collectively" approximates the input states of t where (the concrete execution of) t can produce an output in $\gamma^{\mathcal{A}}(a_1)$. That is, $\bigcup_{a_0 \in A_0} \gamma^{\mathcal{A}}(a_0)$ contains all such input states of t.

The (nonmonotonic version of) cardinal power from $(\mathcal{A}, [\![-]\!]^{\mathcal{A}})$ to $(\mathcal{B}, [\![-]\!]^{\mathcal{B}})$ is applicable under this supposition, and it is an abstract interpretation defined as follows:[13]

$$
\begin{array}{rcl}
\mathcal{C} & \stackrel{\text{def}}{=} & \mathcal{A} \to \mathcal{B} \\
[\![t]\!]^{\mathcal{C}}(c) & \stackrel{\text{def}}{=} & \lambda a_1 \in \mathcal{A}.\ \mathbf{let}\ \Big(A_0 = \{a_0 \in \mathcal{A} \mid ([\![t]\!]^{\mathcal{A}} a_0) \sqcap a_1 \ \neq\ \bot\}\Big) \\
 & & \qquad\qquad \mathbf{in}\ \Big(\bigsqcup \{[\![t]\!]^{\mathcal{B}}(c(a_0)) \mid a_0 \in A_0\}\Big) \\
\gamma^{\mathcal{C}}(c) & \stackrel{\text{def}}{=} & \big\{\sigma \in \mathsf{States} \mid \forall a \in \mathcal{A}.\ (\sigma \in \gamma^{\mathcal{A}}(a)) \Rightarrow \big(\sigma \in \gamma^{\mathcal{B}}(c(a))\big)\big\}
\end{array}
$$

Note that each abstract element c in the cardinal power is a function from $\mathcal{A}$ to $\mathcal{B}$. Logically, this function c represents the conjunction of implications,

$$\bigwedge_{a \in \mathcal{A}} \Big(\big(- \in \gamma^{\mathcal{A}}(a)\big) \Rightarrow \big(- \in \gamma^{\mathcal{B}}(c(a))\big)\Big),$$

where each implication is from a property (i.e. element) in $\mathcal{A}$ to a property in $\mathcal{B}$. That is, c means (i.e., concretizes) the set of states σ such that for all $a \in \mathcal{A}$, if σ satisfies a (i.e., $\sigma \in \gamma^{\mathcal{A}}(a)$) then it also satisfies $c(a)$ (i.e., $\sigma \in \gamma^{\mathcal{B}}(c(a))$). The abstract semantics $[\![t]\!]^{\mathcal{C}}$ of an atomic term t updates each conjunct of c in three steps. Suppose that $[\![t]\!]^{\mathcal{C}}(c)$ is given $a_1 \in \mathcal{A}$. First, $[\![t]\!]^{\mathcal{C}}(c)(a_1)$ computes the set A_0 of abstract values a_0 such that $[\![t]\!]^{\mathcal{A}} a_0 \sqcap a_1 \neq \bot$. Then, it applies c to all elements in A_0, and obtains a subset $B_0 \subseteq \mathcal{B}$. Finally, $[\![t]\!]^{\mathcal{C}}(c)$ "executes" $[\![t]\!]^{\mathcal{B}}$ for all $b_0 \in B_0$, and combines all the results by the join operation. Intuitively, the first two steps of $[\![t]\!]^{\mathcal{C}}(c)(a_1)$ compute an approximation of the input states σ such that (1) σ is in $\gamma^{\mathcal{C}}(c)$ and (2) from σ, term t can produce an output satisfying a_1. To see this, note that by the assumption on $(\mathcal{A}, [\![-]\!]^{\mathcal{A}})$, the set A_0 in the first step collectively approximates the inputs where t can produce an output in $\gamma^{\mathcal{A}}(a_1)$. Since $B_0 = \{c(a_0) \mid a_0 \in A_0\}$ (collectively) approximates the intersection of $\gamma^{\mathcal{C}}(c)$ and the concretization of A_0 (i.e., $\bigcup_{a_0 \in A_0} \gamma^{\mathcal{A}}(a_0)$), the concretization of B_0 should contain all the states σ_0 in $\gamma^{\mathcal{C}}(c)$ where t can produce an output in $\gamma^{\mathcal{A}}(a_1)$. From this intuition about the first two steps, we can obtain an intuitive reading of the final result $[\![t]\!]^{\mathcal{C}}(c)(a_1)$: it is an approximation of output states σ' of t, such that (1) σ' satisfies $\gamma^{\mathcal{A}}(a_1)$, and (2) it is an output from a state in $\gamma^{\mathcal{C}}(c)$.

[13] Cousot and Cousot's original definition requires that all the abstract elements in the cardinal power be monotone functions from $\mathcal{A}$ to $\mathcal{B}$ [CC79]. Here we dropped this requirement, because (1) the requirement does not play the role in the soundness of the cardinal power construction, and (2) the requirement makes it difficult to construct an abstract-value slicer.

We construct an abstract-value slicer $\mathfrak{H}$ for the cardinal power $(\mathcal{C}, [\![-]\!]^{\mathcal{C}})$, when we are given a slicer $\mathfrak{F} = (\{(\mathcal{F}_b, \mathsf{ex}^{\mathcal{F}}_b)\}_{b\in\mathcal{B}}, (\!|-|\!)^{\mathcal{F}})$ for the target abstract interpretation $(\mathcal{B}, [\![-]\!]^{\mathcal{B}})$.

$$\begin{array}{rcl} \mathcal{H}_c & \stackrel{\text{def}}{=} & \prod_{a\in\mathcal{A}} \mathcal{F}_{c(a)} \qquad \mathsf{ex}^{\mathcal{H}}_c(g) \stackrel{\text{def}}{=} \lambda a.\, \mathsf{ex}^{\mathcal{F}}_{c(a)}(g(a)) \\ (\!|t|\!)^{\mathcal{H}}_{cc'}(g) & \stackrel{\text{def}}{=} & \lambda a_0.\, \textbf{let } \Big(A_1 = \{a_1 \in \mathcal{A} \mid ([\![t]\!]^{\mathcal{A}} a_0) \sqcap a_1 \neq \bot\}\Big) \\ & & \qquad \textbf{in } \Big(\prod \{(\!|t|\!)^{\mathcal{F}}_{c(a_0)c'(a_1)}(g(a_1)) \mid a_1 \in A_1\}\Big) \end{array}$$

Each extractor g in $\mathcal{H}_c$ is a (dependent) function that maps each abstract value a in $\mathcal{A}$ to an extractor for $c(a)$. Intuitively, for each $a \in \mathcal{A}$, $g(a)$ specifies the part of $c(a)$ that should be extracted. The back-tracer $(\!|t|\!)^{\mathcal{H}}_{cc'}$ runs in three steps. Suppose that $(\!|t|\!)^{\mathcal{H}}_{cc'}(g)$ is given an abstract value a_0. Note that this value a_0 is now about the input states, not the output states. Back-tracer $(\!|t|\!)^{\mathcal{H}}_{cc'}(g)$ first computes the set A_1 of abstract values a_1 such that $c'(a_1)$ is influenced by $c(a_0)$: while t is computing $c'(a_1)$, it has used $c(a_0)$. Then, for each obtained a_1 in A_1, $(\!|t|\!)^{\mathcal{H}}_{cc'}(g)$ makes a subroutine call $(\!|t|\!)^{\mathcal{F}}_{c(a_0)c'(a_1)}(g(a_1))$, in order to find out which part of $c(a_0)$ is used to compute the $g(a_1)$-part of $c'(a_1)$. Finally, $(\!|t|\!)^{\mathcal{H}}_{cc'}(g)$ combines all those parts of $c(a_0)$, and returns an extractor that picks the combined part.

Lemma 5.5 (Correctness) *The defined constructor $\mathfrak{H}$ satisfies the requirements for the abstract-value slicers.*

Proof: The first requirement about the finite lattice structure of $\mathcal{H}_c$ follows from two facts: every $\mathcal{F}_b$ is a finite lattice and poset $\mathcal{A}$ is finite. The second requirement is the monotonicity and boundedness of $\mathsf{ex}^{\mathcal{H}}_c$, and it follows from the monotonicity and boundedness of $\mathsf{ex}^{\mathcal{F}}_b$. Finally, the last requirement about back-tracers holds, because of the definition of the cardinal power and its abstract-value slicer, as shown in the following derivation:

$$\begin{array}{ll} & ([\![t]\!]^{\mathcal{C}} \circ \mathsf{ex}^{\mathcal{H}}_c \circ (\!|t|\!)^{\mathcal{H}}_{cc'})(g)(a_1) \\ = & \bigsqcup\Big\{[\![t]\!]^{\mathcal{B}}\Big((\mathsf{ex}^{\mathcal{H}}_c \circ (\!|t|\!)^{\mathcal{H}}_{cc'})(g)(a_0)\Big) \mid [\![t]\!]^{\mathcal{A}} a_0 \sqcap a_1 \neq \bot\Big\} & \text{(by the definition of } [\![t]\!]^{\mathcal{C}}) \\ = & \bigsqcup\Big\{[\![t]\!]^{\mathcal{B}}\Big(\mathsf{ex}^{\mathcal{H}}_c((\!|t|\!)^{\mathcal{H}}_{cc'} g)(a_0)\Big) \mid [\![t]\!]^{\mathcal{A}} a_0 \sqcap a_1 \neq \bot\Big\} & \\ = & \bigsqcup\Big\{[\![t]\!]^{\mathcal{B}}\Big(\mathsf{ex}^{\mathcal{F}}_{c(a_0)}\big((\!|t|\!)^{\mathcal{H}}_{cc'}(g)(a_0)\big)\Big) \mid [\![t]\!]^{\mathcal{A}} a_0 \sqcap a_1 \neq \bot\Big\} & \text{(by the definition of } \mathsf{ex}^{\mathcal{H}}_c) \\ = & \bigsqcup\Big\{([\![t]\!]^{\mathcal{B}} \circ \mathsf{ex}^{\mathcal{F}}_{c(a_0)})\Big(\prod\{(\!|t|\!)^{\mathcal{F}}_{c(a_0)c'(a'_1)}(g(a'_1)) \mid [\![t]\!]^{\mathcal{A}} a_0 \sqcap a'_1 \neq \bot\}\Big) \mid [\![t]\!]^{\mathcal{A}} a_0 \sqcap a_1 \neq \bot\Big\} & \\ & & \text{(by the definition of } (\!|t|\!)^{\mathcal{H}}) \\ \sqsubseteq & \bigsqcup\Big\{([\![t]\!]^{\mathcal{B}} \circ \mathsf{ex}^{\mathcal{F}}_{c(a_0)})\Big((\!|t|\!)^{\mathcal{F}}_{c(a_0)c'(a_1)}(g(a_1))\Big) \mid [\![t]\!]^{\mathcal{A}} a_0 \sqcap a_1 \neq \bot\Big\} & \\ & & \text{(since } [\![t]\!]^{\mathcal{A}} a_0 \sqcap a_1 \text{ is not } \bot, \text{ and } [\![t]\!]^{\mathcal{B}} \circ \mathsf{ex}^{\mathcal{F}}_{c(a_0)} \text{ is monotone)} \\ \sqsubseteq & \bigsqcup\Big\{\mathsf{ex}^{\mathcal{F}}_{c'(a_1)}(g(a_1)) \mid [\![t]\!]^{\mathcal{A}} a_0 \sqcap a_1 \neq \bot\Big\} & \text{(since } \mathfrak{F} \text{ is an abstract-value slicer for } (\mathcal{B}, [\![-]\!]^{\mathcal{B}})) \\ \sqsubseteq & \mathsf{ex}^{\mathcal{F}}_{c'(a_1)}(g(a_1)) & \\ & & \text{(since } a_1 \text{ is a given free value so } [\![t]\!]^{\mathcal{A}} a_0 \sqcap a_1 \neq \bot \text{ does not affect } \mathsf{ex}^{\mathcal{F}}_{c'(a_1)}(g(a_1))) \\ = & (\mathsf{ex}^{\mathcal{H}}_{c'} g)(a_1) & \text{(by the definition of } \mathsf{ex}^{\mathcal{H}}_{c'}). \end{array}$$

□

6 Experiments

We designed an abstract-value slicer for the "full" zone analysis [Min01], and tested the efficiency of the resulting slicer in the context of proof generation.

6.1 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:

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

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 $a^* \sqsubseteq b$,

$$\begin{array}{rcl} \beta_{ab} & : & \mathcal{E}_b \to \mathcal{E}_a \\ \beta_{ab}(e) & \stackrel{\text{def}}{=} & \textbf{if } (\mathsf{hasNegCycle}(a) = \mathit{true}) \\ & & \textbf{then } \mathsf{edges}(\mathsf{pickNegCycle}(a)) \\ & & \textbf{else } \bigcup\Big\{\Big(\textbf{if } a_{kl} \le b_{kl} \textbf{ then } \{kl\} \textbf{ else } \mathsf{edges}(\mathsf{mPath}(a,k,l))\Big) \mid kl \in e \wedge b_{kl} \neq \infty\Big\} \end{array}$$

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 used the slicer for interval analysis (Example 14), and changed the back-tracer $(\!|x := E|\!)$ of the old slicer as follows:

$$\begin{array}{rcl} (\!|x_i := E|\!)_{ab} & \stackrel{\text{def}}{=} & \textbf{let} \quad e' = e - \{kl \mid b_{kl} = \infty\} \\ & & \qquad\ f = (\!|E|\!)_{(\mathsf{prj}(a^*))[-b_{i0}, b_{0i}]}(\{\mathsf{u} \mid 0i \in e'\} \cup \{\mathsf{l} \mid i0 \in e'\}) \\ & & \textbf{in } \beta_{aa^*}\Big(\{0k \mid \mathsf{u} \in f(x_k)\} \cup \{k0 \mid \mathsf{l} \in f(x_k)\} \cup (e' - \{0i, i0\})\Big) \end{array}$$

where $(\!|E|\!)$ is the expression back-tracing in the slicer for interval analysis.

6.2 Experimental Results

We implemented the full zone analysis, the abstract-value slicer, and the proof construction algorithm using our previous work [SYY03]. 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.

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 1: Number of Sliced DBM Entries

Table 1 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 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 2. 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 2, 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.

7 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. We have also presented the construction of abstract-value slicers for combined abstract interpreters such as the cartesian product and the reduction. For this direction, we plan to consider the known techniques for constructing abstract interpretations systematically [CC79, GRS00, GR99, GS98], and provide corresponding systematic methods for building abstract-value slicers.

The motivating application of the slicer is to reduce the proof size in the proof construction method [SYY03] 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

program	before slicing		after slicing		(1)-(3)/(1)	reduction in proof size[e]
	(1)FOL[a]	(2)formulas[b]	(3)FOL[c]	(4)formulas[d]		
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 2: Reduction in the Proof Size

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 [SYY03].

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,[14] 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.

The question about simplifying or compressing abstract domains has already been studied in the theory of abstract domain transformations [FGR96, GR97, GRS00, CFW98]. It would be

[14]Domain $\mathcal{D}$ amounts to $\prod_{n \in V} \mathcal{A}$ in Section 2, and d_0 and d correspond to f and $\mathsf{ex}_f(\epsilon_0)$ in Section 3.3.

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.

ACKNOWLEDGMENTS

We would like to thank David Schmidt, Alan Mycroft, and Daejun Park for their helpful comments.

References

[AF00] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 243–253, New York, January 2000. ACM Press.

[App01] A. W. Appel. Foundational proof-carrying code. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 247–258, Los Alamitos, June 2001. IEEE Computer Society Press.

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, New York, June 2001. ACM Press.

[Bou93] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 46–55, New York, June 1993. ACM Press.

[BR01] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the SPIN Workshop on Model Checking Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer-Verlag, 2001.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, New York, January 1977. ACM Press.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, New York, 1979. ACM Press.

[CC92] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2–3):103–179, 1992.

[CC99] P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6(1):69–95, 1999.

[CFW98] A. Cortesi, G. Filé, and W. H. Winsborough. The quotient of an abstract interpretation. *Theoretical Computer Science*, 202(1-2):163–192, 1998.

[CGJ+00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer-Verlag, 2000.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. The MIT Press, 1999.

[CLRS01] H. T. Cormen, E. C. Leiserson, L. R. Rivest, and C. Stein. *Introduction to algorithms*. The MIT Press, second edition, 2001.

[Cou81] P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[Cou99] P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[Cou05] P. Cousot. Abstract interpretation. MIT course 16.399, `http://web.mit.edu/16.399/www/`, Feb.–May 2005.

[DGG97] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.

[DGS95] E. Duesterwald, R. Gupta, and M. L. Soffa. Demand-driven computation of interprocedural data flow. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 37–48, New York, January 1995. ACM Press.

[DP90] D. A. Davey and H. A. Priestley. *Introduction to lattices and order*. Cambridge University Press, 1990.

[DW90] K. Davis and P. L. Wadler. Backwards strictness analysis: Proved and improved. In *Functional Programming: Proceedings of the 1989 Glasgow Workshop, 21-23 August 1989*, pages 12–30. Springer-Verlag, 1990.

[FGR96] G. Filé, R. Giacobazzi, and F. Ranzato. A unifying view of abstract domain design. *ACM Computing Surveys*, 28(2):333–336, 1996.

[GM04] R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 186–197, New York, January 2004. ACM Press.

[GR97] R. Giacobazzi and F. Ranzato. Refining and compressing abstract domains. In *International Colloquium on Automata, Languages and Programming*, volume 1256 of *Lecture Notes in Computer Science*, pages 771–781. Springer-Verlag, 1997.

[GR99] R. Giacobazzi and F. Ranzato. The reduced relative power operation on abstract domains. *Theoretical Computer Science*, 216(1-2):159–211, 1999.

[GRS00] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag, 1997.

[GS98] R. Giacobazzi and F. Scozzari. A logical model for relational abstract domains. *ACM Transactions on Programming Languages and Systems*, 20(5):1067–1109, 1998.

[HJMS02] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 58–70, New York, January 2002. ACM Press.

[HJMS03] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *Proceedings of the SPIN Workshop on Model Checking Software*, volume 2548 of *Lecture Notes in Computer Science*, pages 235–239. Springer-Verlag, 2003.

[HKL04] J. M. Howe, A. King, and L Lu. Analysing logic programs by reasoning backwards. In *Program Development in Computational Logic*, volume 3049 of *Lecture Notes in Computer Science*, pages 152–188. Springer-Verlag, 2004.

[HL92] J. Hughes and J. Launchbury. Reversing abstract interpretations. In *European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, pages 269–286. Springer-Verlag, February 1992.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[HST+02] N. Hamid, Z. Shaoi, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 89–100, Los Alamitos, June 2002. IEEE Computer Society Press.

[Hug88] J. Hughes. Backwards analysis of functional programs. In *Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, pages 187–208, October 1988.

[KL02] A. King and L. Lu. A backward analysis for constraint logic programs. *Theory and Practice of Logic Programming*, 2(4-5):517–547, 2002.

[Mas01] D. Massé. Combining backward and forward analyses of temporal properties. In *Proceedings of the Second Symposium PADO'2001, Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 155–172. Springer-Verlag, 21 – 23 May 2001.

[Min01] A. Miné. A new numerical abstract domain based on difference-bound matrices. In *Proceedings of the Second Symposium PADO'2001, Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 155–172. Springer-Verlag, May 2001.

[MWCG98] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, New York, January 1998. ACM Press.

[Nec97] G. C. Necula. Proof-carrying code. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, New York, January 1997. ACM Press.

[NL97] G. C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In G. Vigna, editor, *Special Issue on Mobile Agent Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer-Verlag, 1997.

[NR01] G. C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 142–154, New York, January 2001. ACM Press.

[NS02] G. C. Necula and R. Schneck. Proof-carrying code with untrusted proof rules. In *Software Security – Theories and Systems*, volume 2609 of *Lecture Notes in Computer Science*, pages 283–298. Springer-Verlag, November 2002.

[Riv05a] X. Rival. Abstract dependences for alarm diagnosis. In *Asian Symposium on Programming Languages and Systems*, volume 3780 of *Lecture Notes in Computer Science*, pages 347–363. Springer-Verlag, November 2005.

[Riv05b] X. Rival. Understanding the origin of alarms in ASTRÉE. In *Static Analysis Symposium*, volume 3672 of *Lecture Notes in Computer Science*, pages 303–319. Springer-Verlag, 2005.

[SYY03] S. Seo, H. Yang, and K. Yi. Automatic construction of Hoare proofs from abstract interpretation results. In *Asian Symposium on Programming Languages and Systems*, volume 2895 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, November 2003.

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[WH87] P. Wadler and R. J. M. Hughes. Projections for Strictness Analysis. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 385–407. Springer, Berlin, September 1987.