A Framework for Numeric Analysis of Array Operations*

Denis Gopan† University of Wisconsin gopan@cs.wisc.edu

Thomas Reps‡ University of Wisconsin reps@cs.wisc.edu

Mooly Sagiv§ Tel-Aviv University msagiv@post.tau.ac.il

ABSTRACT

Automatic discovery of relationships among values of array elements is a challenging problem due to the unbounded nature of arrays. We present a framework for analyzing array operations that is capable of capturing numeric properties of array elements. In particular, the analysis is able to establish that all array elements are initialized by an array-initialization loop, as well as to discover numeric constraints on the values of initialized elements.

The analysis is based on the combination of canonical abstraction and summarizing numeric domains. We describe a prototype implementation of the analysis and discuss our experience with applying the prototype to several examples, including the verification of correctness of an insertion-sort procedure.

Categories and Subject Descriptors

D.3.1 [**Formal Definitions and Theory**]: Semantics; F.3.2 [**Semantics of Programming Languages**]: Program analysis

General Terms

Languages, Verification

Keywords

Program analysis, array analysis, abstract numeric domains, canonical abstraction, summarization

*Supported in part by ONR under contract N00014-01-1-0796.

†Supported in part by a Cisco Systems Wisconsin Distinguished Graduate Fellowship.

‡Supported in part by NSF under grant CCR-9986308, by ONR under contract N00014-01-1-0708, and by the Alexander von Humboldt Foundation.

§Supported in part by a grant from the Israeli Academy of Science and by the Alexander von Humboldt Foundation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'05, January 12–14, 2005, Long Beach, California, USA.

Copyright 2005 ACM 1-58113-830-X/05/0001 ...$5.00.

1. INTRODUCTION

An array is a simple and efficient data structure that is heavily used. In many cases, to verify the correctness of programs that use arrays an analysis needs to be able to discover relationships among values of array elements, as well as their relationships to scalar variables and constants. For example, in scientific programing, a sparse matrix is represented with several arrays, and indirect indexing is used to access matrix elements. In this case, to verify that all array accesses are in bounds, an analysis has to discover upper and lower bounds on the elements stored in the index arrays. Mutual-exclusion protocols, such as the Bakery and Peterson algorithms [10, 16], use certain relationships among the values stored in a shared integer array to decide which processes may enter their critical section. To verify the correctness of these protocols, an analysis must be capable of capturing these relationships.

Static reasoning about array elements is problematic because of the unbounded nature of arrays. Array operations tend to be implemented without having a particular fixed array size in mind. Rather, the code is parametrized by scalar variables that have certain numeric relationships to the actual size of the array. The proper verification of such code requires establishing the desired property for any values of those parameters with which the code may be invoked. These symbolic constraints on the size of the array preclude the analysis from modeling each array element as an independent scalar variable and using standard numeric analysis techniques to verify the property.

Alternatively, an entire array may be treated as a single *summary* numeric quantity. In this case, numeric properties established for the summary quantity that represents an array must be shared by all array elements. This approach, known as *array smashing* [2], resolves the unboundedness issue. However, the problem with this approach, as with any approach that uses such aggregation, is the inability to perform *strong updates* when assigning to individual array elements;[1] this can lead to significant precision loss.

In this paper, we present a static-analysis framework that combines *canonical abstraction* [19, 11] and *summarizing numeric domains* [7], and is more precise than array smashing. The analysis uses canonical abstraction to partition an unbounded set of array elements into a bounded number of

[1]A strong update corresponds to a kill of a scalar variable; it represents a definite change in value to all concrete objects that the abstract object represents. Strong updates cannot generally be performed on summary objects because a (concrete) update only affects one of the summarized concrete objects.

groups. Partitioning is done based on certain properties of array elements, in particular, on numeric relationships between their indices and values of scalar variables. The elements that have the same properties are grouped together. Each group is represented by an abstract array element. The abstraction is *storeless* in the sense that there is no static connection between a concrete array element and an abstract array element that represents it: after the properties of a concrete array element change, it may be represented by a different abstract element.

The analysis uses summarizing numeric domains to keep track of the values and indices of array elements. Indices of array elements are modeled explicitly; that is, two numeric quantities are associated with each array element: the actual value of the element and its index.

Intuitively, the analysis attempts to partition array elements into groups about which stronger assertions can be established and maintained. For example, if an array element is assigned to, the analysis tries to isolate that element in a separate group, so that a strong update may be performed. When analyzing array-initialization code, the analysis attempts to keep array elements that were already initialized in a separate group from the uninitialized ones. For the verification of sorting routines, the analysis tries to separate portions of arrays that have been sorted from the unsorted portions, and so on.

Given a program that uses multiple arrays and non-array variables, an important question is how to partition each array to verify the desired property. We found that a simple heuristic of partitioning arrays with respect to variables that are used to access them is very effective in practice. The array elements that are indexed by scalar variables are placed by themselves into individual groups, which are represented by non-summary abstract elements. Array segments in between the indexed elements are also placed into individual groups, but these groups are represented by summary abstract elements. Such a partitioning heuristic allows the analysis to discover constraints on the values of array elements after simple initialization loops; this can be done fully automatically.

More complex properties, such as verifying comparison-based sorting algorithms, require extra care. To this end, in Sect. 4.3, we introduce auxiliary predicates that are attached to each abstract partition element and encompass numeric properties that are beyond the capabilities of summarizing numeric domains. Throughout the paper, we give several examples of such auxiliary predicates and illustrate how they can be used to establish the desired properties for several challenging examples.[2]

The goal of the analysis is to collect an overapproximation of the set of reachable states at each program point. We use the abstract-interpretation framework [3] to formalize the analysis. The abstract states that are obtained for each program point are a set of triples; each triple consists of an array partition, an element of a summarizing abstract numeric domain, and a valuation of auxiliary predicates.

To implement a prototype of the analysis, we extended the TVLA tool [11] to provide it with the capability to reason about numeric quantities. TVLA uses *three-valued logical structures* to describe states of a program. We had to associate an element of a summarizing numeric domain with each three-valued structure, to extend TVLA's internal machinery to maintain numeric states correctly, and to extend the specification language to incorporate predicates that include numeric comparisons. The summarizing numeric domain was implemented by wrapping the Parma library for polyhedral analysis [1] in the manner described in [7]. We then defined a set of predicates necessary for describing and partitioning arrays. With this prototype implementation, we were able to analyze successfully several kernel examples, including verifying the correctness of an insertion-sort implementation.

The contributions we make in this paper are:

- We introduce an abstract domain suitable for analyzing properties of complex array operations.
- More generally, we show how two previously described techniques, canonical abstraction and summarizing numeric domains, can be combined to work together.
- We describe a working prototype of the analysis, and illustrate it with several non-trivial examples.

1.1 Related work

The problem of reasoning about values stored in arrays has been addressed in previous research. Here we review some of the related approaches that we are aware of.

Masdupuy, in his dissertation [13], uses numeric domains to capture relationships among values and index positions of elements of *statically initialized* arrays. In contrast, our framework allows to discover such relationships for *dynamically initialized* arrays. In particular, canonical abstraction lets our approach retain precision by handling assignments to array elements using strong updates.

Blanchet et al., while building a *special-purpose static program analyzer* [2], recognized the need for handling values of array elements. They proposed two practical approaches: (i) *array expansion*, i.e., introducing an abstract element for each index in the array; and (ii) *array smashing*, i.e., using a single abstract element to represent all array elements. Array expansion is precise, but in practice can only be used for arrays of small size, and is not able to handle unbounded arrays. Array smashing allows handling arbitrary arrays efficiently, but suffers precision losses due to the need to perform weak updates. Our approach combines the benefits of both array expansion and array smashing by dynamically expanding the elements that are read or written to so as to avoid weak updates, and smashing together the remaining elements.

Flanagan and Qadeer used predicate abstraction to infer universally-quantified loop invariants [6]. To handle unbounded arrays, they used predicates over *Skolem constants*, which are synthetically introduced variables with unconstrained values. These variables are then quantified out from the inferred invariants. Our approach is different in that we model the values of all array elements directly and use *summarization* to handle unbounded arrays. Also, our approach uses abstract numeric domains to maintain the numeric state of the program, which obviates the need for calls to a theorem prover.

Černý introduced the technique of *parametric predicate abstraction* [20], in which special-purpose abstract domains

[2] The process of identifying auxiliary predicates and their abstract transformers is not, at present, performed automatically. Sect. 6.2 discuses some possibilities for carrying out these steps automatically.

```
void array_copy(int a[], int b[], int n) {
   i ← 0; (⋆)
   while(i < n) {
      b[i] ← a[i];
      i ← i + 1;
   } (⋆⋆)
}
```

Figure 1: Array-copy function.

are designed to reason about properties of array elements. The domains are parametrized with numeric quantities that designate the portion of an array for which the desired property holds. The abstract transformers are manually defined for each domain. The analysis instantiates the domains by explicitly modeling their parameters in the numeric state. Our approach differs in two respects. First, in our approach numeric states directly model array elements; which allows the analysis to automatically synthesize certain invariants that involve values of array elements. Second, our approach separates the task of array partitioning from the task of establishing the properties of array elements. This separation allows the user to concentrate directly on formulating auxiliary predicates that capture the properties of interest.

Canonical abstraction [19, 11] was first introduced for the purpose of determining "shape invariants" for programs that perform destructive updating on dynamically allocated storage. However, it lacked the ability to explicitly represent numeric quantities. Also, [7] introduced a method for extending existing numeric domains with the capability of reasoning about unbounded collections of numeric quantities. However, static partitioning of numeric quantities was assumed. Our work combines these techniques and shows how their combination can be used for verifying properties of array operations.

1.2 Paper organization

The paper is structured as follows: Sect. 2 gives an overview of the analysis. Sect. 3 discusses concrete semantics. Sect. 4 introduces the abstraction. Sect. 5 details the analysis of the running example. Sect. 6 outlines a prototype analysis implementation. Sect. 7 describes our experiences with the analysis prototype. Sect. 8 concludes the presentation.

2. RUNNING EXAMPLE

In this section, we illustrate our technique using a simple example. The procedure in Fig. 1 copies the contents of array `a` into array `b`. Both arrays are of size `n`, which is specified as a parameter to the procedure. Let us assume that the analysis has already determined some facts about values stored in array `a`. For instance, assume that the values of elements in array `a` range from -5 to 5. At the exit of the procedure, we would like to establish that the values stored in array `b` also range from -5 to 5. Furthermore, we would like to establish this property for any reasonable array size, i.e., for all values of `n` greater than or equal to one.

Our technique operates by partitioning the unbounded number of concrete array elements into a bounded number of groups. Each group is represented by an abstract array element. The partitioning is done by introducing relations between the indices of array elements and the value of loop variable `i`. In particular, for the example in Fig. 1, our technique will group the elements of the two arrays with indices less than `i` into two *summary* array elements (denoted by $a_{<i}$ and $b_{<i}$, respectively). Array elements with indices greater than `i` are grouped into two other *summary* array elements ($a_{>i}$ and $b_{>i}$).

Array elements `a[i]` and `b[i]` are not grouped with any other array elements, and are represented by *non-summary* abstract array elements a_i and b_i. Such partitioning allows the analysis to perform a strong update when it processes the assignment statement in the body of the loop.

Fig. 2(a) shows how the elements of both arrays are partitioned during the first iteration of the loop. Each of the abstract array elements a_i and b_i represents a single concrete array element of the corresponding array. This allows the analysis to conclude that the value of the concrete array element `b[0]` that is represented by b_i ranges from -5 to 5 after the assignment `b[i] ← a[i]`.

As variable `i` gets incremented, the grouping of concrete array elements changes. The element `b[0]`, represented by b_i, moves into the group of the concrete array elements that are represented by $b_{<i}$. The abstract element b_i represents the array element `b[1]` that is extracted from the group of concrete array elements that is represented by $b_{>i}$. The elements of array `a` are treated similarly. Fig. 2(b) shows how the arrays `a` and `b` are partitioned during the second iteration.

The analysis reflects the change in grouping of array elements by *combining* the numeric properties associated with b_i with the numeric properties associated with $b_{<i}$. The new numeric properties for the abstract element b_i are obtained by duplicating the numeric properties associated with $b_{>i}$. As a result, at the beginning of the second iteration, the analysis establishes that the value of the concrete array element represented by $b_{<i}$ ranges from -5 to 5. Numeric properties associated with abstract elements of array `a` are treated similarly.

As the value of `i` increases with each iteration, more and more of the concrete array elements of both arrays move from the two groups subscripted by "$> i$", to the two groups subscripted by "i", and finally, to the two groups subscripted by "$< i$". Fig. 2(c) shows how the arrays are partitioned on the k-th iteration. The concrete array elements that are represented by $b_{<i}$ are the elements that have been initialized. Suppose that the analysis has established that the values of the elements represented by $b_{<i}$ at the beginning of the k-th iteration range from -5 to 5. After interpreting the assignment in the body of the loop, the analysis establishes that the value of the element `b[k]`, represented by b_i, also ranges from -5 to 5. After the increment of variable `i`, the numeric properties associated with b_i are combined with the properties associated with $b_{<i}$. As a result, the analysis establishes that the values of the concrete elements represented by $b_{<i}$ at the beginning of the $k+1$-st iteration range from -5 to 5.

An important thing to observe is that, even though the partitions shown in Fig. 2 (b) and (c) describe different groupings of concrete array elements, both partitions have the same sets of abstract array elements. Therefore, from the point of view of the analysis these partitions are the same. To establish which concrete array elements are represented by a particular abstract element, the analysis directly models the values of indices of array elements in the numeric state associated with each partition.

Fig. 2(e) shows how the array elements are partitioned

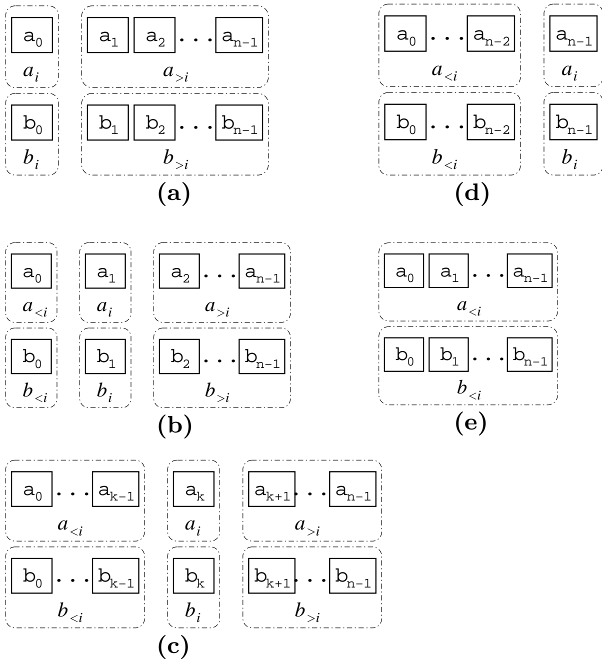


Figure 2: Partitionings of array elements at different points in the execution of the array-copy function: (a) on the 1-st loop iteration; (b) on the 2-nd iteration; (c) on the k-th iteration; (d) on the last iteration; (e) after exiting the loop.

after exiting from the loop. We have just shown that, on each iteration, the analysis established that the values of the concrete array elements that are represented by $b_{<i}$ range from -5 to 5. After the loop, as shown in Fig. 2(e), $b_{<i}$ represents all of the concrete elements of array `b`. Therefore, the analysis is able to conclude that all the values stored in `b` range from -5 to 5.

The analysis is also able to establish a more general property, namely, that the value of each element of array `b` is equal to the value of the element of array `a` with the same index. Unfortunately, the numeric domains that are used by the analysis are not capable of maintaining the numeric relationships of this kind for the concrete array elements that have been summarized together. To capture such relationships, we augment each abstract array element of array `b` with an extra value that indicates whether the property holds for (i) all, (ii) some, or (iii) none of the concrete array elements represented by that abstract element. This is done by introducing an auxiliary three-valued unary logic predicate δ, which evaluates to the values 1, 1/2, and 0 on the abstract elements of array `b` to represent cases (i), (ii), and (iii), respectively.

The analysis proceeds as follows: elements of array `b` start with δ evaluating to 1/2, which indicates that the analysis has no knowledge about the values stored in array `b`. On each iteration, the property is established for the array element b_i, i.e., $\delta(b_i)$ is set to 1. At the end of the iteration, the concrete array element represented by b_i is merged into the group of elements represented by $b_{<i}$. On the first iteration, no $b_{<i}$ exists, so the properties of b_i are directly transfered to $b_{<i}$, i.e., $\delta(b_{<i}) = 1$. On the following iterations, the new value for $\delta(b_{<i})$ is determined by joining its current value,

$$
\begin{array}{llll}
\mathit{expr} ::= c & & \mathit{stmt} ::= v \leftarrow \mathit{expr} & \\
\quad \mid v & & \quad \mid a[v] \leftarrow \mathit{expr} & \\
\quad \mid a[v] & & \quad \mid \textbf{if}(\mathit{cond})\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} & \\
\quad \mid \mathit{expr} \odot \mathit{expr} & & \quad \mid \textbf{while}(\mathit{cond})\ \mathit{stmt} & \\
\mathit{cond} ::= \mathit{expr} \bowtie \mathit{expr} & & \quad \mid \mathit{stmt};\ \mathit{stmt} &
\end{array}
$$

$$
c \in \mathbb{V},\ \ v \in \mathit{Scalar},\ \ a \in \mathit{Array}
$$
$$
\odot \in \{+, -, \times\},\ \ \bowtie \in \{<, \leq, =, \geq, >\}
$$

Figure 3: An array-manipulation language.

which is 1, with the value $\delta(b_i)$, which also equals to 1. As a result, the analysis establishes that $\delta(b_{<i}) = 1$ after each iteration, which indicates that the property of interest holds for all initialized array elements.

3. CONCRETE SEMANTICS

Our goal is to analyze programs that operate on a fixed, finite set of scalar variables and arrays. A concrete program state stores a corresponding value for each scalar variable and each array element. We denote the set of all possible concrete program states by *Sigma*.

We denote the set of scalar variables and the set of arrays used in the program by

$$
\mathit{Scalar} = \{v_1, ..., v_n\} \quad \text{and} \quad \mathit{Array} = \{A_1, ..., A_k\},
$$

respectively. These sets are the same across all concrete states that may arise as a result of program execution. Because we would like to reason about arrays of arbitrary size, the set of elements of a particular array may differ from state to state. We use the notation A^S to denote the set of elements of array $A \in \mathit{Array}$ in state S. To ensure proper sequencing of array elements, we assume that a concrete state explicitly assigns to each array element its proper index position in the corresponding array.

Let $\mathbb{V}$ denote a set of possible numeric values (such as $\mathbb{Z}$, $\mathbb{Q}$, or $\mathbb{R}$). We encode a concrete state S with the help of the following four functions:

- $\mathit{Value}^S : \mathit{Scalar} \to \mathbb{V}$ maps each scalar variable to its corresponding value,
- $\mathit{Size}^S : \mathit{Array} \to \mathbb{N}$ maps each array to its size,
- $\mathit{Value}^S_A : A^S \to \mathbb{V}$ maps an element of an array $A \in \mathit{Array}$ to its corresponding value,
- $\mathit{Index}^S_A : A^S \to \mathbb{N}$ maps an element of an array $A \in \mathit{Array}$ to its index position in the array.

EXAMPLE 1. *Let program P operate on two scalar variables,* `i` *and* `j`*, and an array* `B` *of size 10. Suppose that at some point in the execution of the program, the values of variables* `i` *and* `j` *are* 4 *and* 7*, respectively, and the values that are stored in array* `B` *are* $\{1, 3, 8, 12, 5, 7, 4, -2, 15, 6\}$*. We encode the concrete state S of the program as follows:*

$$
\mathit{Scalar} = \{i, j\},\ \mathit{Array} = \{B\},\ B^S = \{b_0, \ldots, b_9\}
$$

$$
\mathit{Value}^S = [i \mapsto 4, j \mapsto 7],\ \mathit{Size}^S = [B \mapsto 10]
$$

Notation:
$c \in \mathbb{V},\ v \in Scalar,\ A \in Array, \quad S \in \Sigma,\ D \subseteq \Sigma$
$\odot \in \{+, -, \times\},\ \bowtie \in \{<, \leq, =, \geq, >\}$
$elem(S, A, v) = \{u \in A : Index^S_A(u) = Value^S(v)\}$

Expressions:
$[\![c]\!](S) = c, \quad [\![v]\!](S) = Value^S(v)$
$[\![A[v]]\!](S) = \begin{cases} Value^S_A(u) & \text{if } \exists u \in elem(S, A, v) \\ \bot & \text{otherwise} \end{cases}$
$[\![expr_1 \odot expr_2]\!](S) = [\![expr_1]\!](S) \odot [\![expr_2]\!](S)$

Conditions:
$[\![expr_1 \bowtie expr_2]\!](S) = [\![expr_1]\!](S) \bowtie [\![expr_2]\!](S)$

Assignments:
$[\![v \leftarrow expr]\!](S) = S\,[v \mapsto [\![expr]\!](S)]$

$[\![a[v] \leftarrow expr]\!](S) = \begin{cases} S\,[u \mapsto [\![expr]\!](S)] & \text{if } \exists u \in elem(S, A, v) \\ \bot & \text{otherwise} \end{cases}$

Errors:
$[\![.]\!](\bot) = \bot$

Set transformers:
$[\![v \leftarrow expr]\!](D) = \{[\![v \leftarrow expr]\!](S) : S \in D\} \quad (Assign_s)$
$[\![a[v] \leftarrow expr]\!](D) = \{[\![a[v] \leftarrow expr]\!](S) : S \in D\} \quad (Assign_a)$
$[\![cond]\!](D) = \{S : S \in D \text{ and } [\![cond]\!](S) = true\} \quad (Cond)$
$D_1 \sqcup D_2 = D_1 \cup D_2 \quad (Join)$

Figure 4: Concrete collecting semantics for the language shown in Fig. 3.

$$Value^S_B = [b_0 \mapsto 1, b_1 \mapsto 3, b_2 \mapsto 8, \ldots, b_9 \mapsto 6]$$

$$Index^S_B = [b_0 \mapsto 0, b_1 \mapsto 1, b_2 \mapsto 2, \ldots, b_9 \mapsto 9]$$

In Fig. 3, we define a simple language suitable for expressing array operations. The language consists of assignment statements, conditional statements, and while loops. Values can be assigned to both scalar variables and array elements. However, the language does not allow to use arbitrary expressions explicitly to access array elements. Only scalar variables are allowed to index into arrays.

We define the program's concrete collecting semantics as follows. To each program point we attach a set of concrete states, D. The set transformers shown in Fig. 4 are used to propagate the sets of concrete states through the program. Set transformers are defined for assigning to scalar variables ($Assign_s$) and array elements ($Assign_a$), interpreting numeric conditionals of if-statements and while-statements ($Cond$), and joining sets of concrete states at control merge points ($Join$).

The goal of the analysis is to collect the set of reachable program states at each program point. Determining the exact sets of concrete states is, in general, undecidable. We use the framework of abstract interpretation [3] to collect at each program point an overapproximation of the set of states that may arise there.

4. ABSTRACT DOMAIN

In this section, we define the family of abstract domains that is the main contribution of this paper. The elements of the abstract domains are sets of *abstract memory configurations*. Each abstract memory configuration $S^\#$ is a triple $\langle P^\#, \Omega^\#, \Delta^\# \rangle$, in which $P^\#$ specifies how arrays are partitioned, $\Omega^\#$ represents the corresponding abstract numeric state, and $\Delta^\#$ stores the values of auxiliary predicates. We denote the set of all possible abstract partitions by $\Sigma^\#$.

4.1 Array partitioning

The goals of array partitioning are twofold. First, we would like to isolate in separate groups the array elements that are assigned to. This allows the analysis to perform strong updates when assigning to these elements. Second, we would like to group elements with similar properties together to minimize the precision loss due to summarization.

In this paper, we explore an array-partitioning scheme based on numeric relationships among indices of array elements and values of scalar variables. In particular, given a set of scalar variables, we partition an array so that each element whose index is equal to the value of any of the variables in the set is placed in a group by itself. Such groups are represented by *non-summary* abstract array elements. The consecutive array segments in between the indexed elements are grouped together. Such groups are represented by *summary* abstract array elements.

We define array partitions by using a fixed set of *partitioning functions*, denoted by Π. Each function is parametrized by an array and a single scalar variable. Let $A \in Array$ and $v \in Scalar$. In a concrete state S, a function $\pi_{A,v}$ is interpreted as:

$$\pi_{A,v} : A^S \to \{-1, 0, 1\},$$

and is evaluated as follows:

$$\pi_{A,v}(u) = \begin{cases} -1 & \text{if } Index^S_A(u) < Value^S(v) \\ 0 & \text{if } Index^S_A(u) = Value^S(v) \\ 1 & \text{if } Index^S_A(u) > Value^S(v) \end{cases}$$

The choice of values is completely arbitrary as long as the function evaluates to a different value for each of the three cases. We denote the set of partitioning functions parameterized with array A by Π_A.

In a given concrete state, we partition each array $A \in Array$ by grouping together elements of A for which *all* partitioning functions in Π_A evaluate to the same values. Each group is represented by an abstract array element: a non-summary element if at least one partitioning function evaluates to 0 for the array elements in the group; a summary element otherwise. If the set Π_A is empty, all of the elements of array A are grouped together into a single summary element.

The values to which partitioning functions evaluate on the array elements in a group uniquely determine the abstract element that is used to represent that group. We continue using the intuitive abstract-element naming introduced in Sect. 2, e.g., $b_{>i,<j}$ denotes the group of array elements whose indices are greater than the value of `i`, but less than the value of `j`.

Formally, array partition $P^\#$ maps each array in $Array$ to a corresponding set of abstract array elements. We say that two array partitions are equal if they map all arrays in $Array$ to the same sets:

$$P^\#_1 = P^\#_2 \iff \forall A \in Array \left[P^\#_1(A) = P^\#_2(A)\right]$$

The following example illustrates array partitioning.

Example 2. *Assume the same situation as in Ex. 1. Let the set of partitioning functions Π be $(\pi_{B,i}, \pi_{B,j})$. The elements of array B are partitioned into five groups, each of*

which is represented by an abstract array element:
(i) $\{b_0, b_1, b_2, b_3\}$*, represented by* $b_{<i,<j}$*;*
(ii) $\{b_4\}$*, represented by* $b_{i,<j}$*;*
(iii) $\{b_5, b_6\}$*, represented* $b_{>i,<j}$*;*
(iv) $\{b_7\}$*, represented by* $b_{>i,j}$*;*
(v) $\{b_8, b_9\}$*, represented by* $b_{>i,>j}$*.*
The abstract array elements $b_{i,<j}$ *and* $b_{>i,j}$ *are non-summary, while* $b_{<i,<j}$*,* $b_{>i,<j}$*, and* $b_{>i,>j}$ *are summary. Thus,*

$$P^{\#} = [B \mapsto \{b_{<i,<j}, b_{i,<j}, b_{>i,<j}, b_{>i,j}, b_{>i,>j}\}]$$

Note, that since each abstract element of array A corresponds to a valuation of partitioning functions in Π_A, there can be at most $3^{|\Pi_A|}$ abstract array elements of A. Thus, the number of ways to partition array A (i.e., the number of sets of abstract array elements that consistently represent the corresponding array) is finite, although combinatorially large. However, our observations show that only a small fraction of these partitions actually occur in practice.

The approach that is presented in this section illustrates only one of the possibilities for partitioning an array. We found this partitioning to be useful when consecutive array elements share similar properties, e.g., when analyzing simple array-initialization loops and simple array-sorting algorithms, which constitutes a large portion of actual uses of arrays. However, in more complicated examples, e.g., when using double indexing to initialize an array and using an array to store a complex data structure (such as a tree), the above array partitioning is not likely to succeed. We plan to address the issue of improving array partitioning in future work.

4.2 Numeric states

To keep track of the numeric information associated with an array partition, we attach to each partition an element of a summarizing numeric domain [7]. Summarizing numeric domains are capable of modeling values associated with summary objects and are automatically constructed by extending standard relational numeric domains, such as octagons [14] and polyhedra [5, 8]. Each quantity associated with an abstract object is modeled by a dimension in the domain. Dimensions can be summary or non-summary depending on the type of the corresponding object.

In array analysis, we use non-summary dimensions to represent the values of scalar variables, array sizes, and the values and indices of non-summary abstract array elements. Summary dimensions are used to model values and indices of summary abstract array elements. Note that each abstract array element has two dimensions associated with it: one for the value, and one for its index position in the array.

We use the following notation to refer to the dimensions of summarizing numeric domain: v denotes the dimension that represents the values of scalar variable `v`; $A.size$ denotes the dimension that represents the size of an array A. Let $u \in P^{\#}(A)$ denote an arbitrary abstract element of array A. Then, $u.value$ denotes the dimension that represents the value of u, and $u.index$ denotes the dimension that represents its index. For simplicity, we assume that the summarizing numeric domain used in the analysis was constructed by extending polyhedral numeric domain; thus, we present numeric states as sets of linear constraints. A constraint on a quantity associated with a summary object is interpreted to hold for the quantities associated with *all* concrete objects represented by that abstract object.

Example 3. *Assume the same situation as in Ex. 2. The numeric state associated with* $P^{\#}$ *is described by the following set of linear constraints:*

$$i = 4,\ j = 7,\ B.size = 10$$

$$\begin{array}{cc} 0 \le b_{<i,<j}.index \le 3 & 1 \le b_{<i,<j}.value \le 12 \\ b_{i,<j}.index = 4 & b_{i,<j}.value = 5 \\ 5 \le b_{>i,<j}.index \le 6 & 4 \le b_{>i,<j}.value \le 7 \\ b_{>i,j}.index = 7 & b_{>i,j}.value = -2 \\ 8 \le b_{>i,>j}.index \le 9 & 6 \le b_{>i,>j}.value \le 15 \end{array}$$

The definitions of a partial-order relation and of a join operation for a summarizing domain depend on the underlying numeric domain. For example, in a polyhedral domain, the result of a join operation is computed by taking the convex hull of the union of the arguments. Only numeric states that have the same set of dimensions can be compared and joined together. The set of dimensions is determined by the array partition. Thus, only numeric states that are associated with similar array partitions can be compared and joined.

4.3 Beyond summarizing domains

Summarizing numeric domains can be used to reason about collective numeric properties of summarized array elements. However, the relationships among quantities that are summarized together are lost. This precludes summarizing numeric domains from being able to express certain properties of interest, e.g., it is impossible to express the fact that a set of array elements that are summarized together are in sorted order. In Ex. 3, the numeric state $S^{\#}$ is only able to capture the property that the values of the concrete array elements represented by $b_{(<i,<j)}$ range from 1 to 12, but not that those elements are sorted in ascending order.

To capture properties that are beyond the capabilities of summarizing numeric domains, we introduce a set of auxiliary predicates, denoted by Δ. In a concrete state S, a predicate in $\delta_A \in \Delta$ maps each element of array A to a boolean value: to 1 if the property of interest holds for that element, and to 0 otherwise:

$$\delta_A : A^S \to \{0, 1\}\,.$$

We specify the semantics of auxiliary predicates by supplying a formula that is evaluated in concrete states.

When the elements of array A are summarized, we *join* the corresponding values of δ_A in a *3-valued logic lattice* [19]. In 3-valued logic, an extra value, denoted by 1/2, is added to the set of Boolean values $\{0, 1\}$. The order is defined as follows:

$$l_1 \sqsubseteq l_2 \text{ iff } l_1 = l_2 \text{ or } l_2 = 1/2$$

Thus,

$$1/2 \sqcup 0 = 1/2 \sqcup 1 = 0 \sqcup 1 = 1/2.$$

The resulting value is attached to the corresponding abstract array element.

In an abstract memory configuration, we use an abstract counterpart of the predicate, denoted by $\delta_A^{\#}$, to map abstract array elements to corresponding values:

$$\delta_A^{\#} : P^{\#}(A) \to \{0, 1, 1/2\}$$

Let $u \in P^{\#}(A)$ be an arbitrary abstract array element. The value of $\delta_A^{\#}(u)$ is interpreted as follows: the value of 1 indicates that the property holds for all of the array elements

represented by u; the value of 0 indicates that the property does not hold for any of the array elements represented by u; and the value of $1/2$ indicates that property may hold for some of the array elements represented by u, but may not hold for the rest of the elements.

EXAMPLE 4. *Assume the same situation as in Ex. 3. We introduce a predicate δ_B that evaluates to 1 for array elements that are in ascending order, and to 0 for the elements that are not:*

$$\delta_B(u) = \forall t \in B \quad Index^S_B(t) < Index^S_B(u) \Rightarrow Value^S_B(t) \le Value^S_B(u)$$

In the concrete state shown in Ex. 1, δ_B evaluates to 1 for the elements b_0, b_1, b_2, b_3, and b_8; and to 0 for the remaining elements. The values associated with the abstract array elements are constructed as follows:

$\delta^{\#}_B(b_{<i,<j}) = \bigsqcup_{i=0}^{3} \delta_B(b_i) = 1 \sqcup 1 \sqcup 1 \sqcup 1 = 1$
$\delta^{\#}_B(b_{i,<j}) = \delta_B(b_4) = 0$
$\delta^{\#}_B(b_{>i,<j}) = \delta_B(b_5) \sqcup \delta_B(b_6) = 0 \sqcup 0 = 0$
$\delta^{\#}_B(b_{>i,j}) = \delta_B(b_7) = 0$
$\delta^{\#}_B(b_{>i,>j}) = \delta_B(b_8) \sqcup \delta_B(b_9) = 1 \sqcup 0 = 1/2$

The part of an abstract memory configuration that stores the interpretation of auxiliary predicates is denoted by $\Delta^{\#}$ and is defined as:

$$\Delta^{\#}(\delta_A, u) = \delta^{\#}_A(u)$$

It only makes sense to compare and join the interpretations that are defined for the same set of abstract array elements, i.e., the interpretations that are associated with the same array partition $P^{\#}$. We define a partial order relation for interpretations of auxiliary predicates as follows:

$$\Delta^{\#}_1 \sqsubseteq \Delta^{\#}_2 \iff \forall \delta_A \in \Delta \ \forall u \in P^{\#}(A) \ \left[\Delta^{\#}_1(\delta_A, u) \sqsubseteq \Delta^{\#}_2(\delta_A, u)\right].$$

The join operation for interpretations of auxiliary predicates is defined as follows: we say that $\Delta^{\#}_1 \sqcup \Delta^{\#}_2 = \Delta^{\#}$, where for all $\delta_A \in \Delta$ and for all $u \in P^{\#}(A)$

$$\Delta^{\#}(\delta_A, u) = \Delta^{\#}_1(\delta_A, u) \sqcup \Delta^{\#}_2(\delta_A, u).$$

4.4 Abstract states

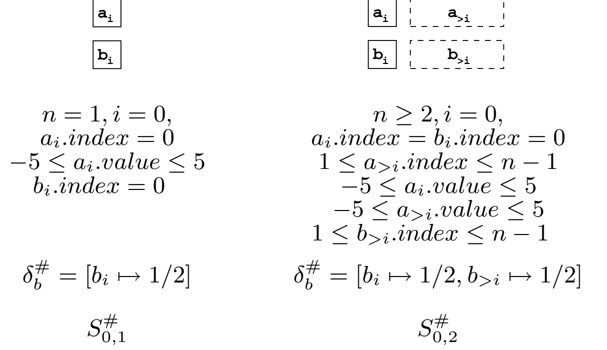
Examples 2, 3, and 4 illustrate the construction of an abstract memory configuration that represents an arbitrary concrete state. We use a function $\beta : \Sigma \to \Sigma^{\#}$ to refer to this process of abstracting a single concrete state.

Let $S^{\#}_1 = \left\langle P^{\#}_1, \Omega^{\#}_1, \Delta^{\#}_1 \right\rangle$ and $S^{\#}_2 = \left\langle P^{\#}_2, \Omega^{\#}_2, \Delta^{\#}_2 \right\rangle$ denote two abstract memory configurations. We define a partial-order relation for the abstract memory configurations as follows:

$$S^{\#}_1 \sqsubseteq S^{\#}_2 \iff P^{\#}_1 = P^{\#}_2 \wedge \Omega^{\#}_1 \sqsubseteq \Omega^{\#}_2 \wedge \Delta^{\#}_1 \sqsubseteq \Delta^{\#}_2$$

The join operation is only defined for the abstract memory configurations with similar array partitions, i.e., when $P^{\#}_1 = P^{\#}_2 = P^{\#}$. The resulting abstract partition is defined by

$$S^{\#} = S^{\#}_1 \sqcup S^{\#}_2 = \left\langle P^{\#},\ \Omega^{\#}_1 \sqcup \Omega^{\#}_2,\ \Delta^{\#}_1 \sqcup \Delta^{\#}_2 \right\rangle.$$



$n = 1, i = 0,$ $a_i.index = 0$ $-5 \le a_i.value \le 5$ $b_i.index = 0$	$n \ge 2, i = 0,$ $a_i.index = b_i.index = 0$ $1 \le a_{>i}.index \le n-1$ $-5 \le a_i.value \le 5$ $-5 \le a_{>i}.value \le 5$ $1 \le b_{>i}.index \le n-1$
$\delta^{\#}_b = [b_i \mapsto 1/2]$	$\delta^{\#}_b = [b_i \mapsto 1/2, b_{>i} \mapsto 1/2]$
$S^{\#}_{0,1}$	$S^{\#}_{0,2}$

Figure 5: The abstract state at the entrance of the loop (program point $(\star)$) in Fig. 1. The abstract state contains two abstract memory configurations: one represents arrays of length 1; the other represents arrays of length 2 and greater.

Given a set of abstract memory configurations $SS^{\#}$, we say that $C^{\#}$ is a *partition congruence class* iff it is a maximal-sized subset of $SS^{\#}$, all of whose members partition arrays similarly (i.e., there exists $P^{\#}$ such that for all $S^{\#}_i \in C^{\#}$, $P^{\#}_i = P^{\#}$). We define the abstraction function for a set of concrete states $D \subseteq \Sigma$ as follows:

$$\alpha(D) = \left\{ \bigsqcup C^{\#} : \begin{array}{c} C^{\#} \subseteq \{\beta(S) : S \in D\} \\ \text{is a partition congruence class} \end{array} \right\}$$

Thus, an abstract state $D^{\#}$ is a *set* of abstract memory configurations with distinct array partitions. The concretization function is defined as follows:

$$\gamma(D^{\#}) = \left\{ S \ : \ \exists S^{\#} \in D^{\#} \text{ s.t. } \beta(S) \sqsubseteq S^{\#} \right\}.$$

We define a partial-order and a join operation for abstract states as follows. Let $D^{\#}_1, D^{\#}_2 \subseteq \Sigma^{\#}$ denote two abstract states. The partial-order relation is defined by:

$$D^{\#}_1 \sqsubseteq D^{\#}_2 \iff \forall S^{\#}_1 \in D^{\#}_1 \ \exists S^{\#}_2 \in D^{\#}_2 \ \left[S^{\#}_1 \sqsubseteq S^{\#}_2\right]$$

The join of two abstract states is performed by computing a union of the corresponding sets of abstract memory configurations. The configurations that have similar array partitions are joined together. This is similar in spirit to the partially disjunctive heap abstraction described in [12].

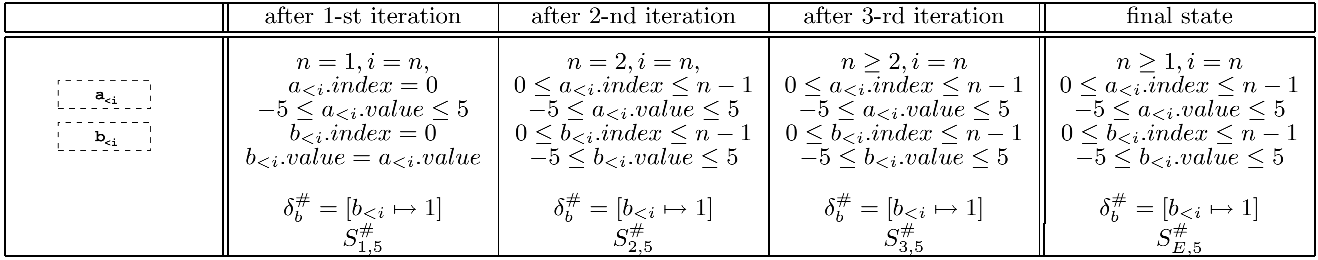
5. RUNNING EXAMPLE REVISITED

In this section, we flesh out the schematic illustration of the analysis that was given in Sect. 2. The analysis is applied to the code shown in Fig. 1. We depict the abstract memory configurations that arise in the course of the analysis as follows. The partition of the arrays is shown graphically: solid boxes represent non-summary abstract array elements; dashed boxes represent summary abstract array elements. Numeric states are shown as sets of linear constraints. Auxiliary predicates are shown as maps from sets of abstract array elements to corresponding values in $\{0, 1, 1/2\}$.

Consider the program in Fig. 1. The set of scalar variables and the set of arrays are defined as follows: $Scalar = \{i, n\}$ and $Array = \{a, b\}$. The analysis uses the set of partitioning functions $\Pi = \{\pi_{a,i}, \pi_{b,i}\}$. It is beyond the capabilities of

	a_i b_i	a_i $a_{>i}$ b_i $b_{>i}$	$a_{<i}$ a_i $b_{<i}$ b_i	$a_{<i}$ a_i $a_{>i}$ $b_{<i}$ b_i $b_{>i}$
1-st iteration	$i = 0, n = 1,$ $a_i.index = 0$ $-5 \leq a_i.value \leq 5$ $b_i.index = 0$ $\delta_b^\# = [b_i \mapsto 1/2]$ $S^\#_{1,1}$	$i = 0, n \geq 2,$ $a_i.index = b_i.index = 0$ $1 \leq a_{>i}.index \leq n-1$ $-5 \leq a_i.value \leq 5$ $-5 \leq a_{>i}.value \leq 5$ $1 \leq b_{>i}.index \leq n-1$ $\delta_b^\# = [b_i \mapsto 1/2, b_{>i} \mapsto 1/2]$ $S^\#_{1,2}$	(not present)	(not present)
2-nd iteration	(same as above)	(same as above)	$i = 1, n = 2,$ $a_i.index = b_i.index = 1$ $a_{<i}.index = 0$ $-5 \leq a_i.value \leq 5$ $-5 \leq a_{<i}.value \leq 5$ $b_{<i}.index = 0$ $b_{<i}.value = a_{<i}.value$ $\delta_b^\# = [b_{<i} \mapsto 1, b_i \mapsto 1/2]$ $S^\#_{2,3}$	$i = 1, n \geq 3,$ $a_i.index = b_i.index = 1$ $2 \leq a_{>i}.index \leq n-1$ $a_{<i}.index = 0$ $-5 \leq a_i.value \leq 5$ $-5 \leq a_{>i}.value \leq 5$ $-5 \leq a_{<i}.value \leq 5$ $2 \leq b_{>i}.index \leq n-1$ $b_{<i}.value = a_{<i}.value$ $\delta_b^\# = [b_{<i} \mapsto 1, b_i \mapsto 1/2, b_{>i} \mapsto 1/2]$ $S^\#_{2,4}$
3-rd iteration	(same as above)	(same as above)	$i = 2, n = 3$ $a_i.index = b_i.index = 2$ $0 \leq a_{<i}.index \leq 1$ $-5 \leq a_i.value \leq 5$ $-5 \leq a_{<i}.value \leq 5$ $0 \leq b_{<i}.index \leq 1$ $-5 \leq b_{<i}.value \leq 5$ $\delta_b^\# = [b_{<i} \mapsto 1, b_i \mapsto 1/2]$ $S^\#_{3,3}$	$i = 2, n \geq 4$ $a_i.index = b_i.index = 2$ $0 \leq a_{<i}.index \leq 1$ $3 \leq a_{>i}.index \leq n-1$ $-5 \leq a_i.value \leq 5$ $-5 \leq a_{>i}.value \leq 5$ $-5 \leq a_{<i}.value \leq 5$ $0 \leq b_{<i}.index \leq 1$ $3 \leq b_{>i}.index \leq n-1$ $-5 \leq b_{<i}.value \leq 5$ $\delta_b^\# = [b_{<i} \mapsto 1, b_i \mapsto 1/2, b_{>i} \mapsto 1/2]$ $S^\#_{3,4}$
3-rd iteration (after join)	(same as above)	(same as above)	$1 \leq i \leq 2$ $n = i + 1$ $a_i.index = i$ $0 \leq a_{<i}.index \leq i-1$ $-5 \leq a_i.value \leq 5$ $-5 \leq a_{<i}.value \leq 5$ $b_i.index = i$ $0 \leq b_{<i}.index \leq i-1$ $-5 \leq b_{<i}.value \leq 5$ $\delta_b^\# = [b_{<i} \mapsto 1, b_i \mapsto 1/2]$ $S^\#_{J,3}$	$1 \leq i \leq 2$ $i = a_i.index = b_i.index$ $0 \leq a_{<i}.index \leq i-1$ $i+1 \leq a_{>i}.index \leq n-1$ $-5 \leq a_i.value \leq 5$ $-5 \leq a_{>i}.value \leq 5$ $-5 \leq a_{<i}.value \leq 5$ $0 \leq b_{<i}.index \leq i-1$ $i+1 \leq b_{>i}.index \leq n-1$ $-5 \leq b_{<i}.value \leq 5$ $\delta_b^\# = [b_{<i} \mapsto 1, b_i \mapsto 1/2, b_{>i} \mapsto 1/2]$ $S^\#_{J,4}$
3-rd iteration (after widening)	(same as above)	(same as above)	$1 \leq i$ $n = i + 1$ $a_i.index = i$ $0 \leq a_{<i}.index \leq i-1$ $-5 \leq a_i.value \leq 5$ $-5 \leq a_{<i}.value \leq 5$ $b_i.index = i$ $0 \leq b_{<i}.index \leq i-1$ $-5 \leq b_{<i}.value \leq 5$ $\delta_b^\# = [b_{<i} \mapsto 1, b_i \mapsto 1/2]$ $S^\#_{W,3}$	$1 \leq i$ $i = a_i.index = b_i.index$ $0 \leq a_{<i}.index \leq i-1$ $i+1 \leq a_{>i}.index \leq n-1$ $-5 \leq a_i.value \leq 5$ $-5 \leq a_{>i}.value \leq 5$ $-5 \leq a_{<i}.value \leq 5$ $0 \leq b_{<i}.index \leq i-1$ $i+1 \leq b_{>i}.index \leq n-1$ $-5 \leq b_{<i}.value \leq 5$ $\delta_b^\# = [b_{<i} \mapsto 1, b_i \mapsto 1/2, b_{>i} \mapsto 1/2]$ $S^\#_{W,4}$

Figure 6: Abstract memory configurations at the begining of the 1-st, 2-nd and 3-d iterations of the loop in Fig. 1. The sets of abstract memory configurations in each row form corresponding abstract states. Last two rows show the transformation of the abstract state on the third iteration, as is is joined and widened with respect to the abstract state obtained on the second iteration.



	after 1-st iteration	after 2-nd iteration	after 3-rd iteration	final state
a<i b<i	$n = 1, i = n,$ $a_{<i}.index = 0$ $-5 \leq a_{<i}.value \leq 5$ $b_{<i}.index = 0$ $b_{<i}.value = a_{<i}.value$ $\delta_b^\# = [b_{<i} \mapsto 1]$ $S_{1,5}^\#$	$n = 2, i = n,$ $0 \leq a_{<i}.index \leq n-1$ $-5 \leq a_{<i}.value \leq 5$ $0 \leq b_{<i}.index \leq n-1$ $-5 \leq b_{<i}.value \leq 5$ $\delta_b^\# = [b_{<i} \mapsto 1]$ $S_{2,5}^\#$	$n \geq 2, i = n$ $0 \leq a_{<i}.index \leq n-1$ $-5 \leq a_{<i}.value \leq 5$ $0 \leq b_{<i}.index \leq n-1$ $-5 \leq b_{<i}.value \leq 5$ $\delta_b^\# = [b_{<i} \mapsto 1]$ $S_{3,5}^\#$	$n \geq 1, i = n$ $0 \leq a_{<i}.index \leq n-1$ $-5 \leq a_{<i}.value \leq 5$ $0 \leq b_{<i}.index \leq n-1$ $-5 \leq b_{<i}.value \leq 5$ $\delta_b^\# = [b_{<i} \mapsto 1]$ $S_{E,5}^\#$

Figure 7: The abstract state (consisting of a single abstract memory configuration) that reaches program point ($\star\star$) after the 1-st, 2-nd, and 3-d iterations of the loop in Fig. 1. The last column shows the stabilized abstract state at ($\star\star$).

summarizing numeric domains alone to express the property "for every index k, the value of `b[k]` is equal to the value of `a[k]`". To capture this property, we introduce an auxiliary predicate δ_b, whose semantics in a concrete state S is defined by

$$\delta_b(u) = \exists t \in a^S \ \big[Index_b^S(u) = Index_a^S(t) \wedge Value_b^S(u) = Value_a^S(t)\big].$$

Fig. 5 shows the abstract state at program point ($\star$). The abstract state contains two abstract memory configurations: $S_{0,1}^\#$ and $S_{0,2}^\#$. Configuration $S_{0,1}^\#$ represents the degenerate case of each array containing only one element. Thus, each array is represented by a single abstract array element, a_i and b_i, respectively. The indices of both a_i and b_i are equal to zero, and the value of a_i ranges from -5 to 5.

Abstract memory configuration $S_{0,2}^\#$ represents the concrete states in which both arrays are of length greater than or equal to two. In these situations, each array is represented two abstract elements: a_i and b_i represent the first elements of the corresponding arrays, while $a_{>i}$ and $b_{>i}$ represent the remaining elements. The numeric state associated with this partition indicates that the indices of the concrete array elements represented by a_i and b_i are equal to zero, the indices of the concrete array elements represented by $a_{>i}$ and $b_{>i}$ range from 1 to $n-1$, and the values of all concrete elements of array `a` range from -5 to 5.

The auxiliary predicate $\delta_b^\#$ evaluates to 1/2 for all array elements in the abstract memory configurations $S_{0,1}^\#$ and $S_{0,2}^\#$. This means that, in the concrete states represented by $S_{0,1}^\#$ and $S_{0,2}^\#$, the values of the concrete elements of array `b` may either be equal to the values of the corresponding elements of array `a` or not.

Fig. 6 shows the abstract states that the analysis encounters at the beginning of each iteration. Each entry in the table represents an abstract memory configuration. The abstract state at a particular iteration is the set of abstract memory configurations shown in the corresponding row of the table. Table columns correspond to distinct array partitions that arise during the analysis. Fig. 7 shows the evolution, on successive iterations of the analysis, of the single abstract memory configuration that reaches the exit of the loop.

The analysis proceeds as follows. Both $S_{0,1}^\#$ and $S_{0,2}^\#$ satisfy the loop condition and are propagated into the body of the loop. The abstract state at the beginning of the first iteration contains abstract memory configurations $S_{1,1}^\#$ and $S_{1,2}^\#$, which are similar to $S_{0,1}^\#$ and $S_{0,2}^\#$, respectively. After the assignment "`b[i]` ← `a[i]`", two changes happen to both abstract memory configurations: (i) the constraint $a_i.value = b_i.value$ is added to their numeric states, and (ii) the value of auxiliary predicate $\delta_b^\#(b_i)$ is changed to 1.

At the end of the first iteration, as variable `i` is incremented, abstract memory configuration $S_{1,1}^\#$ is transformed into configuration $S_{1,5}^\#$. The loop condition does not hold in $S_{1,5}^\#$, thus, this memory configuration is propagated to the program point after the exit of the loop. Abstract memory configuration $S_{1,2}^\#$ is transformed into two new abstract memory configurations: $S_{2,3}^\#$ and $S_{2,4}^\#$. These memory configurations, along with $S_{1,1}^\#$ and $S_{1,2}^\#$, form the abstract state at the beginning of the second iteration.

At the end of the second iteration, the abstract memory configuration $S_{2,3}^\#$ is transformed into configuration $S_{2,5}^\#$, which violates the loop condition, and is, thus, propagated to program point ($\star\star$). The abstract memory configuration $S_{2,4}^\#$ is transformed into two new abstract memory configurations, $S_{3,3}^\#$ and $S_{3,4}^\#$, in which loop condition holds. Because the abstract state accumulated at the head of the loop already contains memory configurations with similar array partitions, the numeric states and the values of auxiliary predicates for these abstract memory configurations are joined. In particular, abstract memory configuration $S_{2,3}^\#$ is joined with $S_{3,3}^\#$ to produce $S_{J,3}^\#$. Similarly, $S_{2,4}^\#$ is joined with $S_{3,4}^\#$, resulting in $S_{J,4}^\#$. Furthermore, widening is applied: $S_{J,3}^\#$ is widened with respect to $S_{2,3}^\#$, producing abstract memory configuration $S_{W,3}^\#$; and $S_{J,4}^\#$ is widened with respect to $S_{2,4}^\#$, resulting in $S_{W,4}^\#$.

On the third iteration, abstract memory configuration $S_{W,3}^\#$ is transformed into $S_{3,5}^\#$, which is propagated to program point ($\star\star$). Abstract memory configuration $S_{W,4}^\#$, is transformed into memory configurations $S_{W,3}^\#$ and $S_{W,4}^\#$, which were previously encountered by the analysis. At this stage, the abstract state at the head of the loop stabilizes and the analysis terminates.

All of the abstract memory configurations that reach program point ($\star\star$) partition arrays similarly. The analysis joins these memory configurations to produce the abstract state that contains a single abstract memory configuration $S_{E,5}^\#$ shown in the last column of Fig. 7. It is easy to see that this configuration represents only the concrete states in which (i) the values stored in the array `b` range from -5 to 5 (follows from "$-5 \leq b_{<i} \leq 5$" constraint in the numeric state); and

(ii) the value of each element of array `b` is equal to the value of the element of array `a` with the same index (follows from the fact that $\delta_b^{\#}(b_{<i})$ evaluates to definite value 1).

6. ABSTRACT SEMANTICS

To make the abstraction described in previous sections usable, we have to define the abstract counterparts for the concrete state transformers shown in Fig. 4.

In [4], it is shown that for a Galois connection defined by abstraction function α and concretization function γ, the best abstract transformer for a concrete transformer τ, denoted by $\tau^\sharp$, can be expressed as: $\tau^\sharp = \alpha \circ \tau \circ \gamma$. This defines the limit of precision obtainable using a given abstract domain; however, it is a non-constructive definition: it does not provide an *algorithm* for finding or applying $\tau^\sharp$.

We implemented a prototype of our analysis using the TVLA tool [11], and defined *overapproximations* for the best abstract state transformers by using TVLA mechanisms. Space considerations preclude us from giving a full account of the modifications that needed to be applied to TVLA, and the exact definitions of the predicates needed to support array operations. In the rest of this section, we give a brief overview of TVLA, and sketch the techniques for modeling arrays and defining abstract transformers.

6.1 An extension of TVLA

TVLA models concrete states by first-order logical structures. The elements of a structure's universe represent the concrete objects. Predicates encode relationships among the concrete objects. The abstract states are represented by *three-valued logical structures*, which are constructed by applying canonical abstraction to the sets of concrete states. The abstraction is performed by identifying a set of abstraction predicates and representing the concrete objects for which these abstraction predicates evaluate to the same values by a single element in the universe of a three-valued structure. In the rest of the paper, we refer to these abstract elements as *nodes*. A node that represents a single concrete object is called non-summary node, and a node that represent multiple concrete objects is called summary node.

TVLA distinguishes between two types of predicates: *core* predicates and *instrumentation* predicates. Core predicates are the predicates that are necessary to model the concrete states. Instrumentation predicates, which are defined in terms of core predicates, are introduced to capture properties that would otherwise be lost due to abstraction.

An abstract state transformer is defined in TVLA as a sequence of (optional) steps:

- A *focus step* replaces a three-valued structure by a set of more precise three-valued structures that represent the same set of concrete states as the original structure. Usually, focus is used to "materialize" a non-summary node from a summary node. The structures resulting from a focus are not necessarily images of canonical abstraction, in the sense that they may have multiple nodes for which the abstraction predicates evaluate to the same values.

- A *precondition step* filters out the structures for which a specified property does not hold from the set of structures produced by focus. Generally, preconditions are used to model conditional statements.

- An *update step* transforms the structures that satisfy the precondition, to reflect the effect of an assignment statement. This is done by changing the interpretation of core and instrumentation predicates in each structure.

- A *coerce step* is a cleanup operation that "sharpens" updated three-valued structures by making them comply with a set of globally defined integrity constraints.

- A *blur step* restores the "canonicity" of coerced three-valued structures by applying canonical abstraction to them, i.e., merging together the nodes for which the abstraction predicates evaluate to the same values.

We extended TVLA with the capability to explicitly model numeric quantities. In particular, we added the facilities to associate a set of numeric quantities with each concrete object, and equipped each three-valued logical structure with an element of a summarizing numeric domain to represent the values of these quantities in abstract states. Each node in a three-valued structure is associated with a dimension of a summarizing numeric domain. TVLA specification language was extended to permit using numeric comparisons in logical formulas, and to specify numeric updates.

The operations that affect numeric states, such as performing numeric updates and evaluating numeric comparisons, as well as creating, removing, merging, and duplicating nodes in the structure are handled by the corresponding abstract transformers of a summarizing numeric domain. The focus and coerce operations use the "assume" operation, provided by the domain, to augment a numeric state with extra constraints.

6.2 Modeling arrays

We encode concrete states of a program as follows. Each scalar variable and each array element corresponds to an element in the universe of the first-order logical structure. We define a *core* unary predicate for each scalar variable and for each array. These predicates evaluate to 1 on the elements of the first-order structure that represent the corresponding scalar variable or the element of corresponding array, and to 0 for the rest of the elements. Each element in the universe is associated with a numeric quantity that represents its value. Each array element is associated with an extra numeric quantity that represents its index position in the array.

To model the array structure in TVLA correctly, extra predicates are required. We model the adjacency relation among array elements by introducing a binary instrumentation predicate for each array. This predicate evaluates to 1 when evaluated on two adjacent elements of an array. To model the property that indices of array elements are contiguous and do not repeat, we introduce a unary instrumentation predicate for each array that encodes transitive closure of the adjacency relation.

Partitioning functions are defined by unary *instrumentation* predicates. Since a partitioning function may evaluate to three different values, whereas a predicate can only evaluate to 0 or 1, we use two predicates to encode each partitioning function. Auxiliary predicates directly correspond to unary *instrumentation* predicates.

To perform the abstraction, we select a set of abstraction predicates that contains the predicates corresponding

to scalar variables and arrays, the predicates that encode the transitive closure of adjacency relations for each array, and the predicates that implement the partitioning functions. The auxiliary predicates are non-abstraction predicates. The resulting three-valued structures directly correspond to the abstract memory configurations we defined in Sect. 4.

The transformers for the statements that do not require array repartitioning, e.g., conditional statements, and assignments to array elements and to scalar variables that are not used to index array elements, are modeled trivially. The transformers for the statements that cause a change in array partitioning, e.g., updates of scalar variables that are used to index array elements, are defined as follows: *focus* is applied to the structure to materialize the array element that will be indexed by the variable after the update; then, the value of the scalar variable, and the interpretation of the partitioning predicates are updated; finally, *blur* is used to merge the array element that was indexed by the variable previously, into the appropriate summary node.

To update the interpretation of auxiliary predicates, the programmer must supply predicate-maintenance formulas for each statement that may change the values of those predicates. Also, to reflect the numeric properties, encoded by the auxiliary predicates, in the numeric state as the grouping of the concrete array elements changes, a set of integrity constraints implied by the auxiliary predicates must be supplied.

Aside from the integrity constraints and update formulas for the auxiliary predicates, the conversion of an arbitrary program into a TVLA specification can be performed fully automatically. In the future, we plan to extend the technique for *differencing logical formulas*, described in [17], with the capability to handle numeric formulas. Such an extension will allow us to automatically compute safe abstract transformers for the auxiliary predicates. Another technique that may help to fully automate the analysis involves the use of decision procedures to symbolically compute best abstract transformers [18, 21].

7. EXPERIMENTS

In this section, we describe the application of the analysis prototype to four simple examples. We used a simple heuristic to obtain the set of partitioning functions for each array in the analyzed examples. In particular, for each array access “`a[i]`” in the program we added a partitioning function $\pi_{a,i}$ to the set Π. This approach worked well for all of the examples, except for the insertion-sort implementation, which required the addition of an extra partitioning function.

7.1 Array initialization

Fig. 8 shows a piece of code that initializes array `a` of size `n`. Each array element is assigned a value equal to twice its index position in the array plus 3. The purpose of this example is to illustrate that the analysis is able to automatically discover numeric constraints on the values of array elements.

The array-partitioning heuristic produces a single partitioning function $\pi_{a,i}$ for this example. The analysis establishes that after the code is executed the values stored in the array range from 3 to $2 \times n + 1$. No human intervention in the form of introducing auxiliary predicates is required.

```
int a[n], i, n;

i ← 0;
while(i < n) {
   a[i] ← 2 × i + 3;
   i ← i + 1;
}
```

Figure 8: Array-initialization loop.

```
int a[n], b[n], c[n], i, j, n;

i ← 0;
j ← 0;
while(i < n) {
   if(a[i] == b[i]) {
      c[j] ← i;
      j ← j + 1;
   }
   i ← i + 1;
}
```

Figure 9: Partial array initialization.

In contrast, other approaches that are capable of handling this example [6, 20] require that the predicate that specifies the expected bounds for the values of array elements is supplied explicitly, either by the user or by an automatic abstraction-refinement technique [9].

7.2 Partial array initialization

Fig. 9 contains a more complex array-initialization example. The code repeatedly compares elements of arrays `a` and `b` and, in case they are equal, writes their index position into the array `c`. The portion of array `c` that is initialized depends on the values stored in the arrays `a` and `b`. Three scenarios are possible: (i) none of the elements of `c` are initialized; (ii) an initial segment of `c` is initialized; (iii) all of `c` is initialized. The purpose of this example is to illustrate the handling of multiple arrays as well as partial array initialization.

The array-partitioning heuristic derives a set of three partitioning functions for this example, one for each array: $\Pi = \{\pi_{a,i}, \pi_{b,i}, \pi_{c,j}\}$. The analysis establishes that, after the loop, the elements of array `c` with indices between 0 and $j - 1$ were initialized to values ranging from 0 to $n - 1$. Again, no auxiliary predicates are necessary.

The abstract state that reaches the exit of the loop contains four abstract memory configurations. The first configuration represents concrete states in which none of the array elements are initialized. The value of `j`, in this domain element, is equal to zero, and, thus, the array partition does not contain the abstract element $c_{<j}$.

The second and the third memory configurations represent the concrete states in which only an initial segment of array `c` is initialized. Two different memory configurations are required to represent this case because the analysis distinguishes the case of variable `j` indexing an element in the middle of the array from the case of `j` indexing the last element of the array.

The last abstract memory configuration represents the concrete states in which all elements of array `c` are initialized. In the concrete states represented by this memory configuration, the value of variable `j` is equal to the value of variable `n`, and all elements of array `c` are represented by the abstract array element $c_{<j}$.

```
void sort(int a[], int n) {
   int i, j, k, t;

   i ← 1;
   while(i < n) {
      j ← i;
      while(j > 0) {
         k ← j - 1;
         if(a[j] ≥ a[k]) break;

         t ← a[j];
         a[j] ← a[k];
         a[k] ← t;
         j ← j - 1;
      }
      i ← i + 1;
   }
}
```

Figure 10: Insertion-sort procedure.

The initialized array elements are represented by the abstract array element $c_{<j}$. The array partition of the first memory configuration does not contain element $c_{<j}$, which indicates that no elements were initialized. The numeric states associated with the other abstract memory configurations capture the property that the values of initialized array elements range between 0 and $n-1$.

7.3 Insertion sort

Fig. 10 shows a procedure that sorts an array using an insertion sort. Parameter `n` specifies the size of array `a`. The invariant for the outer loop is that the array is sorted up to the i-th element. The inner loop inserts the i-th element into the sorted portion of the array. An interesting detail about this implementation is that elements are inserted into the sorted portion of the array in reverse order. The purpose of this example is to demonstrate the application of the analysis to a more challenging problem.

The application of the array-partitioning heuristic yields $\Pi = \{\pi_{a,j}\}$. Unfortunately, this partitioning is not sufficient. We also need to use variable `i` to partition the array so that the sorted segment of the array is separate from the unsorted segment. However, since `i` is never explicitly used to index array elements, our array-partitioning heuristic fails to add $\pi_{a,i}$ to the set of partitioning functions. To successfully analyze this example, we have to manually add $\pi_{a,i}$ to Π.

Summarizing numeric domains are not able to preserve the order of summarized array elements. An auxiliary predicate, defined similarly to the predicate δ_B in Ex. 4, needs to be introduced. Our prototype implementation requires user involvement to specify the explicit update formulas for this predicate for each of the program statements. Fortunately, the majority of the program statements do not affect this predicate. Thus, the corresponding update formula for such statements is the identity function. The only non-trivial case is the assignment to an array element.

The human involvement necessitated by the analysis is (i) minor, and (ii) problem-specific. In particular, only one auxiliary predicate needs to be introduced, Furthermore, this predicate is not specific to a given implementation of a sorting algorithm. Rather, it can be reused in the analysis of other implementations, and even in the analysis of other sorting algorithms.

Also, this example identifies some directions for future research: (i) designing better techniques for the automatic array partitioning, and (ii) automatically discovering and maintaining auxiliary predicates.

7.4 Results

We ran the analysis prototype on an Intel-based Linux machine equipped with a 2.4 GHz Pentium 4 processor and 512Mb of memory. Fig. 11 shows the measurements we collected while analyzing the examples discussed above.

The measurements are severely affected by our decision to implement the analysis prototype in TVLA. Because TVLA is a general framework, the structure of an array has to be modeled explicitly by introducing a number of instrumentation predicates and integrity constraints. Consequently, the majority of the analysis time is spent executing focus and coerce operations to ensure that the array structure is preserved. The measurements in Fig. 11 indicate that, on average, focus and coerce account for about 80% of the overall analysis time. Building a dedicated analysis implementation, in which the knowledge of the linear structure of arrays is built into the abstract state transformers, would recover the majority of that time.

Another shortcoming of the analysis prototype is that the number of nodes it uses to represent an array in TVLA is larger than the number of abstract objects created by array partitioning described in Sect. 4.1. For example, extra non-summary nodes are used to represent the first and the last elements of each array. As a result, the reported number of abstract objects in each abstract memory configuration as well as the number of memory configurations in an abstract state is greater than a dedicated analyzer would encounter.

Another factor that slows down the analysis is our use of the polyhedral numeric domain. While offering a superior precision, the polyhedral numeric domain does not scale well as the number of dimensions grow. This property is particularly apparent when a polyhedron that represents the abstract state is a multidimensional hypercube. In the array copy example, the constraints on the values of elements of both arrays form a 10-dimensional hypercube, which provides an explanation of why the analysis takes over 6 minutes. If the constraints on the values of array `a` are excluded from the initial abstract state, the analysis takes merely 8 seconds.

Observation of the numeric constraints that arise in the course of the analysis led us to believe that using less precise, but more efficient weakly-relational domains [15], may speed up the analysis of the above examples without sacrificing precision. We reran the analysis of the array copy example, using a summarizing extension of a weakly-relational domain. The analysis was able to verify the desired properties in 40 seconds, which is a significant improvement over the time it takes to perform the analysis with a polyhedral domain.

8. CONCLUSIONS

Canonical abstraction is a powerful technique that allows static analysis to represent a (potentially unbounded) set of concrete objects with a bounded number of abstract objects. The partitioning imposed on the set of the concrete objects is dynamic in a sense that the same abstract object may represent different groups of the concrete objects within the same abstract state. The net result is an ability to avoid per-

Example	Abstract Memory Configurations (AMCs)		Time for	Time
	Max AMCs per state	Max nodes per AMC	Coerce & Focus (%)	(sec)
Array initialization	7	8	68.3	1.7
Partial initialization	35	20	86.3	194.0
Array copy	7	13	94.2	338.1
Insertion sort	38	14	85.5	48.5

Figure 11: Analysis measurements: maximal number of abstract memory configurations in an abstract state, maximal number of abstract objects in an abstract memory configuration, percentage of the overall analysis time spent on focus and coerce operations, and the overall analysis time.

forming weak updates, which greatly improves the precision of the analysis. In this paper, we explore the possibilities for combining canonical abstraction with existing numeric analyses and applications of the combined analysis to the problem of analyzing array operations.

The analysis we define in this paper is capable of automatically establishing interesting array properties; in particular, we show how it is able to capture numeric constraints on the values of array elements after an array-initialization loop. More sophisticated properties, such as verifying the implementation of comparison-based sorting algorithms, require some human intervention to define necessary auxiliary predicates along with their abstract transformers. The auxiliary predicates that are introduced are *problem-specific*, rather then *program-specific*, which allows them to be reused for the analysis of other programs.

The prototype implementation of the analysis, although not very efficient, can be used to analyze interesting array operations in reasonable times.

9. REFERENCES

[1] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the parma polyhedra library. In *Static Analysis Symp.*, volume 2477, pages 213–229, 2002.

[2] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation: Complexity, Analysis, Transformation.*, pages 85–108. Springer-Verlag, 2002.

[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Symp. on Princ. of Prog. Lang.*, pages 238–252, New York, NY, 1977. ACM Press.

[4] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282, New York, NY, 1979. ACM Press.

[5] P. Cousot and N. Halbwachs. Automatic discovery of linear constraints among variables of a program. In *Symp. on Princ. of Prog. Lang.*, 1978.

[6] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Symp. on Princ. of Prog. Lang.*, pages 191–202, 2002.

[7] D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 512–529, 2004.

[8] N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[9] S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *Int. Conf. on Computer Aided Verification*, pages 135–147, 2004.

[10] L. Lamport. A new approach to proving the correctness of multiprocess programs. *Trans. on Prog. Lang. and Syst.*, 1(1):84–97, July 1979.

[11] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symp.*, pages 280–301, 2000.

[12] R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *Static Analysis Symp.*, pages 265–279, 2004.

[13] F. Masdupuy. *Array Indices Relational Semantic Analysis using Rational Cosets and Trapezoids*. PhD thesis, Ecole Polytechnique, 1993.

[14] A. Mine. The octagon abstract domain. In *Proc. Eighth Working Conf. on Rev. Eng.*, pages 310–322, 2001.

[15] A. Mine. A few graph-based relational numerical abstract domains. In *Static Analysis Symp.*, pages 117–132, 2002.

[16] G. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[17] T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *European Symp. on Programming*, pages 380–398, 2003.

[18] T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *Verification, Model Checking, and Abstract Interpretation*, pages 252–266, 2004.

[19] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Trans. on Prog. Lang. and Syst.*, 24(3):217–298, 2002.

[20] P. Černý. Vérification par interprétation abstraite de prédicats paramétriques. D.E.A. Report, Univ. Paris VII & École normale supérieure, September 2003.

[21] G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 530–545, 2004.