Numeric Analysis of Array Operations

Denis Gopan[†] Thomas Reps[†] Mooly Sagiv[‡]

[†]Comp. Sci. Dept., University of Wisconsin; {gopan,reps}@cs.wisc.edu
[‡]School of Comp. Sci., Tel-Aviv University; msagiv@post.tau.ac.il

Abstract

We present a numeric analysis that is capable of reasoning about array operations. In particular, the analysis is able to establish that all elements of an array have been initialized ("an array kill"), as well as to discover numeric constraints on values of initialized array elements, and to verify the correctness of comparison-based sorting algorithms. The analysis is based on the combination of canonical abstraction and summarizing numeric domains. We present a prototype implementation of the analysis and discuss our experience with applying this prototype to several kernel examples.

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 [7, 11], 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 relationships involving the quantity associated with an array must be shared by all array elements. This approach, mentioned in [2], resolves the unboundedness issue. In previous work, we introduced a systematic way to extend standard numeric domains to be able to reason about such summary quantities [6]. The problem with this approach, as with any approach that uses aggregation of this sort, 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 attempts to combine the good features of the two approaches sketched above. An analysis partitions a potentially *unbounded* set of array elements into a *bounded number* of disjoint groups. Some groups will contain only a single array element and will be treated similarly to scalar variables. Others will contain multiple elements and will be modeled as *summary* quantities. To maintain numeric states associated with such partitions, the analysis employs *summarizing numeric domains*, which we introduced in our previous work [6].

Intuitively, the analysis attempts to partition array elements into groups about which stronger assertions

[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.

can be established and maintained. For example, if an array element is assigned to, it is beneficial to have that element alone in its group, so that a strong update may be performed. Also, when analyzing array-initialization code, it is advantageous to keep elements that were already initialized in a separate group from the uninitialized ones. For sorting routines, it makes sense to separate portions of arrays that have been sorted from portions that have not, and so on.

The partitioning is done based on numeric relationships among array-element indices and values of scalar variables. We use the technique of *canonical abstraction* [14, 8] to formalize the partitioning. Canonical abstraction of an unbounded set of concrete objects is performed by selecting a finite set of unary predicates whose free variables range over the concrete objects, and grouping together elements for which all predicates evaluate to the same values. The number of groups in each partitioning, as well as the number of possible partitionings, is guaranteed to be finite.

In essence, canonical abstraction allows us to group together concrete array elements with similar properties. As concrete execution progresses, the properties of the elements change, so a particular element may travel from group to group, while the partitioning still remains the same from the point of view of the analysis. To keep track of array elements contained in each group, our analysis directly models the indices of array elements. That is, two numeric quantities are associated with each array element: the actual value of the element and its index.

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.

Given a program that uses multiple arrays and non-array variables, an interesting 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 them is very effective for establishing simpler properties, such as discovering constraints on the values of array elements after an initialization loop. More complex properties, such as (i) verifying comparison-based sorting algorithms, and (ii) establishing that after an array-copy loop the source and destination arrays are equal element-wise, 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. In Sect. 7, we give several examples of such predicates and and illustrate how such predicates can be used to establish properties for several challenging examples.[2]

To implement a prototype of the analysis, we extended the TVLA tool [8] 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 [6]. 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.

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 surveys related work. Sect. 9 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

[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.

−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.

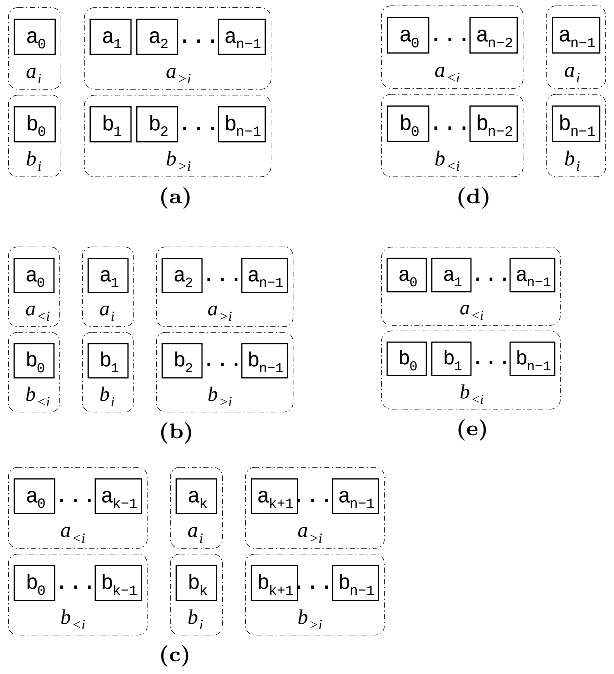
Our technique operates by partitioning the unbounded number of concrete array elements into a bounded number of groups. Each group is represented by an abstract array element. The partitioning is done by introducing relations between the indices of array elements and the value of loop variable i. In 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 on 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 on 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.



a_0 a_i a_1 a_2 a_{n-1} $a_{>i}$
b_0 b_i b_1 b_2 b_{n-1} $b_{>i}$
(a)
a_0 a_{n-2} $a_{<i}$ a_{n-1} a_i
b_0 b_{n-2} $b_{<i}$ b_{n-1} b_i
(d)
a_0 $a_{<i}$ a_1 a_i a_2 a_{n-1} $a_{>i}$
b_0 $b_{<i}$ b_1 b_i b_2 b_{n-1} $b_{>i}$
(b)
a_0 a_1 a_{n-1} $a_{<i}$
b_0 b_1 b_{n-1} $b_{<i}$
(e)
a_0 a_{k-1} $a_{<i}$ a_k a_i a_{k+1} a_{n-1} $a_{>i}$
b_0 b_{k-1} $b_{<i}$ b_k b_i b_{k+1} b_{n-1} $b_{>i}$
(c)

Figure 2: Partitioning of the array elements.

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 $b_{<i}$ in 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 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 the values stored in `b` range from -5 to 5.

The analysis is also able to establish a more general property 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 the abstract state with auxiliary predicates.

The analysis is performed exactly as before. On each iteration, property δ is established for the pair of concrete array elements that are represented by a_i and b_i. At the end of the iteration, the array elements represented by a_i and b_i move into the groups of concrete array elements represented by $a_{<i}$ and $b_{<i}$, for which property δ has already been established on the previous iteration. As a result, the analysis establishes that property δ holds for the concrete elements represented by $a_{<i}$ and $b_{<i}$ at the end of each iteration. In this fashion, the analysis establishes that after exiting the loop, property δ holds for all of the 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 state of the program assigns a value to each scalar variable. It also specifies a size for each array and assigns a value and an index position to each array element.

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

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

respectively. Each array $A_i \in Array$ itself denotes a sequence of elements.

Let $\mathbb{V}$ denote a set of possible numeric values (such as $\mathbb{Z}$ or $\mathbb{Q}$). For each concrete state S we define the following functions:

- $Value^S : Scalar \to \mathbb{V}$ maps each scalar variable to its corresponding value,
- $Size^S : Array \to \mathbb{V}$ maps each array to its size,
- $Value^S[A_i] : A_i \to \mathbb{V}$ maps an element of an array $A_i \in Array$ to its corresponding value,
- $Index^S[A_i] : A_i \to \mathbb{V}$ maps an element of an array $A_i \in Array$ 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 of the program, denoted by S as follows:*

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

$$Values^S = [i \mapsto 4, j \mapsto 7],\ Size^S = [a \mapsto 10]$$

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

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

Following the traditional approach of numeric analysis, we associate each scalar variable and each array element with a dimension in a multidimensional space, and encode a concrete numeric state of the program as a point in that space. Each scalar variable is associated with a single dimension, which represents the value of corresponding variable. Each array is associated with a dimension that represents its size. Each array element is associated with two dimensions: one represents its value and the other represents its index position in the corresponding array. In general, for a concrete state S we have

$$S \in \mathbb{V}^m, \quad \text{where } m = n + k + 2 \times \sum_{A_i \in Array} Size^S(A_i)$$

Given a concrete state $S \in \mathbb{V}^m$, functions $Value^S$ and $Size^S$, as well as families of functions $Value^S[A_i]$ and $Index^S[A_i]$, for each $A_i \in Array$, are defined as projections of the m-dimensional point S onto the corresponding dimensions.

We will denote the set of *all* possible concrete states by Σ. There is a somewhat subtle point concerning Σ that deserves mention: even though each concrete state is modeled as a point in a space of fixed dimensionality, different concrete states may require different numbers of dimensions. For a particular concrete state, none of the transformers can change its dimensionality[3]; it is only the initial state that can have concrete states of different dimensionalities; as we propagate that set

[3]This statement applies to programming languages that support arrays of unchanging length. Our approach can also be used to analyze programs written in languages that support flexible-length arrays.

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

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

Figure 3: A simple programming language.

through the program, the dimensionality for each concrete state does not change. The difference in the number of dimensions is due to our need to model arrays of unspecified size. Thus, Σ is a set of points from spaces with different dimensionality, i.e.,

$$
\Sigma \subseteq \mathbb{V}^+, \quad \text{where } \mathbb{V}^+ = \bigcup_{i=1}^{\infty} \mathbb{V}^i
$$

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. The language does not allow expressions to be used to access elements of arrays. Programs that use expressions to index elements of an array can be trivially transformed into our language by introducing temporary variables.

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 partitions*. Each abstract partition $S^\#$ is a triple $\langle P^\#, \Omega^\#, \Delta^\# \rangle$, where $P^\#$ maps each array to an array partition, $\Omega^\#$ is the associated numeric state, and $\Delta^\#$ is the valuation of the auxiliary predicates. We denote the set of all possible abstract partitions by $\Sigma^\#$.

Expressions:

$[\![c]\!]_\natural(S) = c, \quad \text{where } c \in \mathbb{V}$

$[\![v]\!]_\natural(S) = \mathit{Value}^S(v), \quad \text{where } v \in \mathit{Scalar}$

$$
[\![A[v]]\!]_\natural(S) = \begin{cases} \mathit{Value}^S[A](u) & \text{if } \exists u \in A : \mathit{Index}^S[A](u) = \mathit{Value}^S(v) \\ \bot & \text{otherwise} \end{cases}
$$
where $v \in \mathit{Scalar}, A \in \mathit{Array}$

$[\![\mathit{expr}_1 \odot \mathit{expr}_2]\!]_\natural(S) = [\![\mathit{expr}_1]\!]_\natural(S) \odot [\![\mathit{expr}_2]\!]_\natural(S)$,
where $\odot \in \{+,-,\times\}$

Conditions:

$[\![\mathit{expr}_1 \bowtie \mathit{expr}_2]\!]_\natural(S) = [\![\mathit{expr}_1]\!]_\natural(S) \bowtie [\![\mathit{expr}_2]\!]_\natural(S)$,
where $\bowtie \in \{<,\leq,=,\geq,>\}$

Assignments:

$[\![v \leftarrow \mathit{expr}]\!]_\natural(S) = S\left[v \mapsto [\![\mathit{expr}]\!]_\natural(S)\right]$

$$
[\![a[v] \leftarrow \mathit{expr}]\!]_\natural(S) = \begin{cases} S\left[u \mapsto [\![\mathit{expr}]\!]_\natural(S)\right] & \text{if } \exists u \in A : \mathit{Index}^S[A](u) = \mathit{Value}^S(v) \\ \bot & \text{otherwise} \end{cases}
$$

Errors:

$[\![\cdot]\!]_\natural(\bot) = \bot$

Set transformers:

$[\![v \leftarrow \mathit{expr}]\!]_\natural(D) = \left\{[\![v \leftarrow \mathit{expr}]\!]_\natural(S) : S \in D\right\} \qquad (\mathit{Assign}_s)$

$[\![a[v] \leftarrow \mathit{expr}]\!]_\natural(D) = \left\{[\![a[v] \leftarrow \mathit{expr}]\!]_\natural(S) : S \in D\right\} \qquad (\mathit{Assign}_a)$

$[\![\mathit{cond}]\!]_\natural(D) = \left\{S : S \in D \text{ and } [\![\mathit{cond}]\!]_\natural(S) = \mathit{true}\right\} \qquad (\mathit{Cond})$

$D_1 \sqcup D_2 = D_1 \cup D_2 \qquad (\mathit{Join})$

Figure 4: Concrete Semantics.

4.1 Array partitioning

Given an array and a set of scalar variables, the goal of array partitioning is to separate array elements into several disjoint groups based on numeric relationships between the indices of the elements and the values of the scalar variables. In particular, we would like to partition the array so that each element whose index is equal to the value of any of the scalar variables in the set is a single element in its group. We represent such groups by *non-summary* abstract array elements. The consecutive segments of array elements 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 partitioning function is parameterized by an array and a single scalar variable. Partitioning functions are defined as follows:

$$
\pi_{A,v} : \Sigma \rightarrow A \rightarrow \{-1, 0, 1\},
$$

where $A \in \mathit{Array}$ and $v \in \mathit{Scalar}$. Given a concrete program state S, partitioning function $\pi_{A,v}$ is evaluated as follows:

$$\pi_{A,v}(S)(u) = \begin{cases} -1 & \text{if } Index^S[A](u) \le Value^S(v) - 1 \\ 0 & \text{if } Index^S[A](u) = Value^S(v) \\ 1 & \text{if } Index^S[A](u) \ge Value^S(v) + 1 \end{cases}$$

where $u \in A$. The choice of function values is completely arbitrary as long as the function evaluates to a different value for each of the three cases. We chose -1, 0, and 1 for convenience. We denote the set of partitioning functions parameterized with array A by Π_A.

For each array $A \in Array$, and a set of partitioning functions Π_A, we group together elements of A for which *all* partitioning functions evaluate to the same values. If the set Π_A is empty, all of the elements of array A are grouped together into a single summary element.

Imposing a global ordering on the set Π_A allows us to uniquely characterize each abstract array element by a vector of values of partitioning functions of length $|\Pi_A|$. We refer to such vectors as *canonical names* of abstract array elements. Such naming gives us an effective procedure for testing equality between partitions. We say that two partitions are equal if for every abstract array element in one partition there is an abstract element with the same canonical name in the other partition, and vice versa. We use the notation $cn(u)$ to refer to the canonical name of abstract array element u.

We identify abstract array elements as summary or non-summary as follows: if the canonical name for the abstract array element contains a value 0, the element is non-summary; otherwise, it is summary.

Example 2 *Assume the same situation as in Ex. 1. Let the ordered 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 (the subscript of each abstract array element corresponds to its canonical name):*
(i) $\{b_0, b_1, b_2, b_3\}$, represented by $b_{(-1,-1)}$;
(ii) $\{b_4\}$, represented by $b_{(0,-1)}$;
(iii) $\{b_5, b_6\}$, represented $b_{(1,-1)}$;
(iv) $\{b_7\}$, represented by $b_{1,0}$;
(v) $\{b_8, b_9\}$, represented by $b_{1,1}$.
The abstract array elements $b_{(0,-1)}$ and $b_{(1,0)}$ are non-summary, while $b_{(-1,-1)}$, $b_{(1,-1)}$, and $b_{(1,1)}$ are summary.

Formally, a partition $P^\#$ is defined as a mapping from *Array* to individual array partitions:

$$P^\# = \left[A \mapsto P^\#_A : A \in Array\right],$$

where each $P^\#_A$ denotes the set of abstract elements of array A. We define the equality of two individual array partitions as follows:

$$P^\#_A = Q^\#_A \iff \wedge \begin{array}{l} \forall u \in P^\#_A \ \exists t \in Q^\#_A \ [cn(u) = cn(t)] \\ \forall u \in Q^\#_A \ \exists t \in P^\#_A \ [cn(u) = cn(t)] \end{array}$$

The equality of two global partitions is defined as follows:

$$P^\# = Q^\# \iff \forall A \in Array \ \left[P^\#_A = Q^\#_A\right]$$

Given a set of partitioning functions Π_A, the elements of array A may be partitioned into at most $2 \times |\Pi_A| + 1$ groups. Therefore, each abstract state has a finite number of abstract elements. The number of possible ways to partition an array is finite, although combinatorially large. However, our observations show that only a small fraction of these partitions actually occur in practice.

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. Non-summary dimensions of the summarizing numeric domain are used to model scalar variables, array sizes, and values and indices of non-summary abstract array elements. Summary dimensions are used to model values and indices of summary abstract array elements. The element of the summarizing numeric domain associated with the partition is constructed by *folding together* the dimensions associated with the concrete array elements represented by the same abstract element.

We use the following notations to refer to the dimensions of the summarizing numeric domain. Let $A \in Array$ denote an arbitrary array, let $P^\#_A$ denote the partition of array A, and let $u \in P^\#_A$ denote an abstract element of an array A. Then, 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; $u.value$ denotes the dimension that represents the value of abstract array element u; and $u.index$ denotes the dimension that represents the index of abstract array element u.

Example 3 *Assume the same situation as in Ex. 2. The partition of array B is*

$$P^\# = \left[B \mapsto \left\{b_{(-1,-1)}, b_{(0,-1)}, b_{(1,-1)}, b_{(1,0)}, b_{(1,1)}\right\}\right].$$

The numeric state associated with $P^\#$ is described by the following set of constraints (we assume that the employed summarizing numeric domains are based on polyhedra):

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

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

To manipulate numeric states, we use abstract transformers defined in [6]. Given two numeric states, denoted $\Omega_1^\#$ and $\Omega_2^\#$, we say that $\Omega_1^\# \sqsubseteq \Omega_2^\#$ if both numeric states are defined on the same set of dimensions and $\Omega_1^\# \subseteq \Omega_2^\#$. Note, that both numeric states are defined on the same set of dimensions if and only if they are associated with the same array partition. The definition of the join operation for a summarizing numeric domain depends on the implementation of the domain. In case of a polyhedral domain, the join is performed by taking a convex hull of both numeric states.

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 $b_{(-1,-1)}$ 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 an auxiliary set of predicates, denoted by Δ. The predicate values are attached to the abstract array elements. We specify the semantics of the predicates in Δ by supplying a formula over concrete states. Each predicate is parameterized by an array. Let $A \in \mathit{Array}$; a predicate $\delta_A \in \Delta$ has the type

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

Generally, in each concrete state, δ_A maps each element of array A to either 1 or 0, depending on whether the corresponding property holds for that array element or not. When the elements of A are summarized in the abstract state, we *join* the values to which δ_A evaluates on the array elements that are summarized together. The join is performed in a *3-valued logic lattice*.[4] The resulting value is attached to the corresponding abstract array element.

Let $P_A^\#$ denote the partition of array A, and let $\Gamma^\#$ denote the set of all possible abstract valuations of predicates in Δ. We extract predicate values associated with abstract elements of A by using a function $\delta_A^\#$ of type:

$$\delta_A^\# : \Gamma^\# \to P_A^\# \to \{0, 1, 1/2\}$$

If $\delta_A^\#(\Delta^\#)(u)$ evaluates to 1 for some abstract array element $u \in P_A^\#$, then the property holds for each of the concrete elements of A that are represented by u. Similarly, the value 0 means that the property does not hold for each of the concrete elements of A that are represented by u. The value $1/2$ implies that the property holds for some concrete elements that are represented by u and does not hold for the rest of the elements that are represented by u.

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:*

$$\begin{aligned}\delta_B(S)(u) = &\forall t \in B\\ &\quad \mathit{Index}^S[B](t) < \mathit{Index}^S[B](u) \Rightarrow\\ &\quad \mathit{Value}^S[B](t) \le \mathit{Value}^S[B](u)\end{aligned}$$

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:

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

Given two abstract valuations of auxiliary predicates $\Delta_1^\#$ and $\Delta_2^\#$, which are associated with the same array partition $P^\#$, we say that $\Delta_1^\# \sqsubseteq \Delta_2^\#$ iff

$$\forall \delta_A \in \Delta\ \forall u \in P^\#(A)\ \left[\delta_A^\#(\Delta_1^\#)(u) \sqsubseteq \delta_A^\#(\Delta_2^\#)(u)\right].$$

The join operation for the abstract valuations 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 $u \in P^\#(A)$

$$\delta_A^\#(\Delta^\#)(u) = \delta_A^\#(\Delta_1^\#)(u) \sqcup \delta_A^\#(\Delta_2^\#)(u).$$

[4] 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,

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

For more information see, for instance, [14].

4.4 Abstract states

In Ex. 2, 3, and 4, we showed how to construct an abstract partition that represents a given concrete state. We define the abstraction function β that maps a single concrete state to the corresponding abstract state as follows: we say that β maps a concrete state $S \in \Sigma$ to a singleton set containing the abstract partition that represents state S.

Let $S_1^\#$ and $S_2^\#$ denote two abstract partitions. We define a partial-order relation for the abstract partitions 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 partitions that partition arrays similarly, i.e., $P_1^\# = P_2^\# = P^\#$. The resulting abstract partition is defined by

$$S^\# = S_1^\# \sqcup S_2^\# = \left\langle P^\#,\ \Omega_1^\# \sqcup \Omega_2^\#,\ \Delta_1^\# \sqcup \Delta_2^\# \right\rangle$$

Given the join operation, shown above, we define the abstraction function for the set of concrete states as follows: let $D \subseteq \Sigma$ denote the set of concrete states

$$\alpha(D) = \bigsqcup_{S \in D} \beta(S).$$

Next, let us define partial-order relation and join operation for the abstract states. Let $D_1^\#, D_2^\# \subseteq \Sigma^\#$ denote two abstract states. The partial-order relation is defined by:

$$D_1^\# \sqsubseteq D_2^\# \iff \forall S_1^\# \in D_1^\# \ \exists S_2^\# \in D_2^\# \left[S_1^\# \sqsubseteq S_2^\# \right]$$

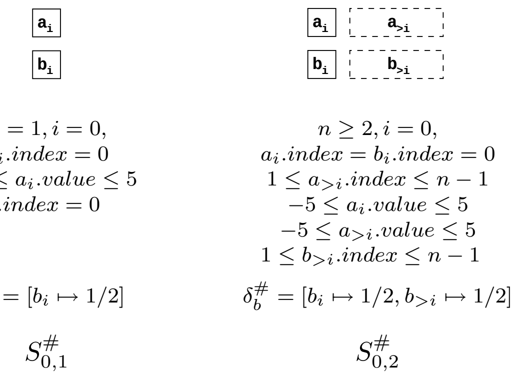
To join two abstract states we union together the sets of abstract partitions. The abstract state cannot contain multiple abstract partitions with the same $P^\#$'s. Thus, the abstract partitions with matching $P^\#$'s are joined together.

Given the partial-order for the abstract states, we define the concretization function as follows: let $D^\# \subseteq \Sigma^\#$ denote an abstract state

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

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 partitions that arise in the course of the analysis as follows. At the top the partition of the array is shown graphically: solid boxes represent non-summary



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

Figure 5: The abstract state before the loop.

abstract array elements; dashed boxes represent summary abstract array elements. Below, the numeric state is shown as a set of constraints. At the bottom, we show the values of the auxiliary predicates for each abstract array element.

Consider the program in Fig. 1. The set of scalar variables and the set of arrays are defined as follows: $Scalar = \{i, n\}$ and $Array = \{a, b\}$. We define the following set of partitioning predicates: $\Pi = \{\pi_{a,i}, \pi_{b,i}\}$. The property that for any index k, the value of `b[k]` is equal to the value of `a[k]` are equal, is beyond the capabilities of summarizing numeric domains alone. We introduce an auxiliary predicate δ_b, which is defined by

$$\delta_b(S)(u) = \exists t \in a \left[Index^S[b](u) = Index^S[a](t) \wedge \right. \\ \left. Value^S[b](u) = Value^S[a](t) \right]$$

to capture this property.

Fig. 5 shows the abstract state before entering the loop. The value of variable `i` is zero. The abstract state contains two abstract partitions: $S_{0,1}^\#$ and $S_{0,2}^\#$. Abstract partition $S_{0,1}^\#$ represents the degenerate case when each array contains only one element. This partition contains only one abstract array element for each array, namely a_i and b_i. 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 partition $S_{0,2}^\#$ represents the concrete states in which both arrays are of length greater than or equal to two. In these situations, each array is partitioned into 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.

	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	$i = 0, n = 1,$ $a_i.index = 0$ $-5 \leq a_i.value \leq 5$ $b_i.index = 0$ $\delta_b^{\#} = [b_i \mapsto 1/2]$ $S_{1,1}^{\#}$	$i = 0, n \geq 2,$ $a_i.index = b_i.index = 0$ $1 \leq a_{>i}.index \leq n-1$ $-5 \leq a_i.value \leq 5$ $-5 \leq a_{>i}.value \leq 5$ $1 \leq b_{>i}.index \leq n-1$ $\delta_b^{\#} = [b_i \mapsto 1/2, b_{>i} \mapsto 1/2]$ $S_{1,2}^{\#}$		
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	$i = 2, n = 3$ $a_i.index = b_i.index = 2$ $0 \leq a_{<i}.index \leq 1$ $-5 \leq a_i.value \leq 5$ $-5 \leq a_{<i}.value \leq 5$ $0 \leq b_{<i}.index \leq 1$ $-5 \leq b_{<i}.value \leq 5$ $\delta_b^{\#} = [b_{<i} \mapsto 1, b_i \mapsto 1/2]$ $S_{3,3}^{\#}$	$i = 2, n \geq 4$ $a_i.index = b_i.index = 2$ $0 \leq a_{<i}.index \leq 1$ $3 \leq a_{>i}.index \leq n-1$ $-5 \leq a_i.value \leq 5$ $-5 \leq a_{>i}.value \leq 5$ $-5 \leq a_{<i}.value \leq 5$ $0 \leq b_{<i}.index \leq 1$ $3 \leq b_{>i}.index \leq n-1$ $-5 \leq b_{<i}.value \leq 5$ $\delta_b^{\#} = [b_{<i} \mapsto 1, b_i \mapsto 1/2, b_{>i} \mapsto 1/2]$ $S_{3,4}^{\#}$
after joining	N/A	N/A	$1 \leq i \leq 2$ $n = i + 1$ $a_i.index = i$ $0 \leq a_{<i}.index \leq i-1$ $-5 \leq a_i.value \leq 5$ $-5 \leq a_{<i}.value \leq 5$ $b_i.index = i$ $0 \leq b_{<i}.index \leq i-1$ $-5 \leq b_{<i}.value \leq 5$ $\delta_b^{\#} = [b_{<i} \mapsto 1, b_i \mapsto 1/2]$ $S_{J,3}^{\#}$	$1 \leq i \leq 2$ $i = a_i.index = b_i.index$ $0 \leq a_{<i}.index \leq i-1$ $i+1 \leq a_{>i}.index \leq n-1$ $-5 \leq a_i.value \leq 5$ $-5 \leq a_{>i}.value \leq 5$ $-5 \leq a_{<i}.value \leq 5$ $0 \leq b_{<i}.index \leq i-1$ $i+1 \leq b_{>i}.index \leq n-1$ $-5 \leq b_{<i}.value \leq 5$ $\delta_b^{\#} = [b_{<i} \mapsto 1, b_i \mapsto 1/2, b_{>i} \mapsto 1/2]$ $S_{J,4}^{\#}$
after widening	N/A	N/A	$1 \leq i$ $n = i + 1$ $a_i.index = i$ $0 \leq a_{<i}.index \leq i-1$ $-5 \leq a_i.value \leq 5$ $-5 \leq a_{<i}.value \leq 5$ $b_i.index = i$ $0 \leq b_{<i}.index \leq i-1$ $-5 \leq b_{<i}.value \leq 5$ $\delta_b^{\#} = [b_{<i} \mapsto 1, b_i \mapsto 1/2]$ $S_{W,3}^{\#}$	$1 \leq i$ $i = a_i.index = b_i.index$ $0 \leq a_{<i}.index \leq i-1$ $i+1 \leq a_{>i}.index \leq n-1$ $-5 \leq a_i.value \leq 5$ $-5 \leq a_{>i}.value \leq 5$ $-5 \leq a_{<i}.value \leq 5$ $0 \leq b_{<i}.index \leq i-1$ $i+1 \leq b_{>i}.index \leq n-1$ $-5 \leq b_{<i}.value \leq 5$ $\delta_b^{\#} = [b_{<i} \mapsto 1, b_i \mapsto 1/2, b_{>i} \mapsto 1/2]$ $S_{W,4}^{\#}$

Figure 6: Abstract states at the begining of each iteration of the loop in Fig. 1.

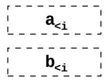
	after 1-st iteration	after 2-nd iteration	after 3-rd iteration	final state
	$n = 1, i = n,$ $a_{<i}.index = 0$ $-5 \leq a_{<i}.value \leq 5$ $b_{<i}.index = 0$ $b_{<i}.value = a_{<i}.value$ $\delta_b^{\#} = [b_{<i} \mapsto 1]$ $S^{\#}_{1,5}$	$n = 2, i = n,$ $0 \leq a_{<i}.index \leq n-1$ $-5 \leq a_{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

Figure 7: The abstract partitions that reach the program point just after the end of the loop for the first, second, and third iterations. The last column shows the combined abstract state at the end of the loop.

The auxiliary predicate $\delta_b^{\#}$ evaluates to 1/2 for all abstract array elements in the abstract partitions $S^{\#}_{0,1}$ 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. The top row shows four of the abstract partitions that arise during the analysis. The remaining rows show numeric states associated with these partitions on the first, second, and third iteration of the analysis. After widening is performed on the third iteration, a fixed point is reached and the analysis terminates. Fig. 7 shows the abstract partitions that reach the exit of the loop after each iteration.

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 partitions $S^{\#}_{1,1}$ and $S^{\#}_{1,2}$, which are equal to $S^{\#}_{0,1}$ and $S^{\#}_{0,2}$, respectively. After the assignment "`b[i]` $\leftarrow$ `a[i]`", two changes happen to both abstract partitions: (i) the constraint $a_i.value = b_i.value$ is added to their numeric states, and (ii) the value of auxiliary predicate $\delta_b^{\#}(b_i)$ changes to 1.

As variable `i` is incremented, partition $S^{\#}_{1,1}$ is transformed into partition $S_{1,5}$. The loop condition does not hold in $S^{\#}_{1,5}$, and this partition is propagated to the program point after the exit of the loop. Partition $S_{1,2}$, after the `i` gets incremented, gives rise to two new partitions: $S^{\#}_{2,3}$ and $S^{\#}_{2,4}$. These partitions, along with the partitions $S^{\#}_{1,1}$ and $S^{\#}_{1,2}$, form the abstract state at the beginning of the second iteration.

At the end of the second iteration, the abstract partition $S^{\#}_{2,3}$ is transformed into the abstract partition $S^{\#}_{2,5}$, which is propagated to the program point after the exit of the loop. The abstract partition $S^{\#}_{2,4}$ is transformed into two abstract partitions: $S^{\#}_{3,3}$ and $S^{\#}_{3,4}$. Because the abstract state accumulated at the head of the loop already contains similar abstract partitions, the numeric states and the values of the auxiliary predicate $\delta_b^{\#}$ for these partitions are joined. In particular, the abstract partition $S^{\#}_{2,3}$ is joined with the abstract partition $S^{\#}_{3,3}$ to produce partition $S^{\#}_{J,3}$. Similarly, the abstract partition $S^{\#}_{2,4}$ is joined with the abstract partition $S^{\#}_{3,4}$, resulting in $S^{\#}_{J,4}$. Furthermore, a widening operation is applied: the abstract partition $S^{\#}_{J,3}$ is widened with respect to the abstract partition $S^{\#}_{2,3}$, producing $S^{\#}_{W,3}$; and the abstract partition $S^{\#}_{J,4}$ is widened with respect to the abstract partition $S^{\#}_{2,4}$, resulting in $S^{\#}_{W,4}$.

At the end of the third iteration, the abstract partition $S^{\#}_{3,5}$, resulting from $S^{\#}_{W,3}$, is propagated to the program point just after the exit of the loop. The abstract partition $S^{\#}_{W,4}$ is transformed into two partitions, which are propagated into the body of the loop. These partitions are equal to $S^{\#}_{W,3}$ and $S^{\#}_{W,4}$, which the analysis has encountered before. Thus, the analysis reaches a fixed point after the third iteration.

All of the abstract partitions that reach the exit of the loop are joined together to produce the abstract state at the exit of the loop. The abstract state contains a single abstract partition: $S^{\#}_{E,5}$. It is easy to see that this abstract partition represents only the concrete states in which (i) the values stored in the array `b` range from -5 to 5 (follows from 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 (because predicate δ_b evaluates to 1 for each element of `b`).

6 Abstract Semantics

To make the abstraction domains 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 [8], 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 reason about 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 formulae, and to specify numeric updates.

The extended abstract states are transformed as follows. Numeric comparison and numeric updates are performed directly on the numeric domain element associated with the structure by using the corresponding numeric domain operations. Node merging and node duplicating which are essential for the implementation of focus and blur operations are implemented by *expand* and *fold* operations of summarizing numeric domain. The coerce operation is extended to handle numeric integrity 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. Resulting three-valued structures directly correspond to the abstract partitions 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, which was indexed by the variable previously, into the appropriate summary node.

To update the interpretation of auxiliary predicates, the programmer must supply incremental 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 predicate-maintenance formulas for the auxiliary predicates, the conversion of an arbitrary program into TVLA specification language can be performed fully automatically. In the future, we plan to extend the technique for *differencing logical formulas*, described in [12], 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 [13, 16].

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

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

Figure 8: Array-initialization loop.

7 Experiments

In this section, we describe the application of the analysis prototype to a handful of kernel examples. We used a simple heuristic to obtain the set of partitioning functions for each array in the analyzed examples. In particular, for each occurrence `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 an 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 analysis established that after the code is executed the values stored in array `a` range from 3 to $2 \times n + 3$.

The array-partitioning heuristic produces a single partitioning function $\pi_{a,i}$ for this example. No auxiliary predicates are required. Thus, the analysis is able to handle this example fully automatically.

7.2 Partial array initialization

Fig. 9 contains a more complex array initialization example. In this example, the indices for which elements of arrays `a` and `b` are equal to each other are written into the initial segment of array `c`. The number of elements of array `c` that are initialized depends on the values stored in arrays `a` and `b`. It may be that (i) none of the elements are initialized; (ii) several consecutive elements in the beginning of the array are initialized; (iii) all of the elements are initialized. We would like to establish an invariant that after the code executes, the

```
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.

values stored in elements of array c whose indices are less than the value of j range from 0 to the value of n. The goal of this example is to illustrate how the analysis handles multiple loops and partial initialization of an array.

The array partitioning heuristic derives a set of three partitioning functions for this example, one for each array: $\Pi = \{\pi_{a,i}, \pi_{b,i}, \pi_{c,j}\}$. Again, no auxiliary predicates are necessary.

The abstract state that reaches the exit of the loop contains four abstract partitions. The first abstract partition represents the concrete states in which none of array c elements have been initialized. The value of j, in this abstract partition, is equal to zero, and the array partition of array c does not contain the abstract array element $c_{<j}$.

The second and third abstract partitions represent the concrete states in which only the initial segment of c was initialized. The reason that two different partitions are required to describe this case is that the analysis distinguishes the case of j indexing an element in the middle of the array from the case of j indexing the last element of the array. The array c is represented by abstract elements $c_{<j}$, c_j, and $c_{>j}$ in the second abstract partition, and by abstract elements $c_{<j}$ and c_j in the third abstract partition.

The last abstract partition represents the concrete states in which all elements of array c were initialized. In this partition the value of j is equal to the value of n, and all elements of array c are represented by the abstract array element $c_{<j}$. The numeric states associated with the second, third, and fourth partitions establish that the values of concrete array elements represented by $c_{<j}$ range from 0 to the value of n.

7.3 Insertion sort

Fig. 10 shows the procedure that implements the insertion sort of an array. Parameter n specifies the size of array a. The invariant for the outer loop is that array a is sorted up to the i-th element. The inner loop inserts the i-th element into the sorted portion of the array.

```
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.

An interesting detail about this implementation is that elements are inserted into the sorted portion of the array in reverse order. To verify this implementation, the analysis needs to establish that the inner loop preserves the invariant of the outer loop.

A subtle issue arises in this example. We would like to use variable i to partition the array: this would separate the sorted segment of the array from the unsorted segment. But, 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 had to manually inject $\pi_{a,i}$ into Π, resulting in $\Pi = \{\pi_{a,i}, \pi_{a,j}\}$.

Summarizing numeric domains are not able to capture "sortedness" of summarized array elements. An auxiliary predicate, defined similarly to the predicate δ_B in Ex. 4, is necessary for the analysis to succeed. Our prototype implementation requires user interaction to specify the explicit update formulas for the predicate for each of the program statements. For virtually all statements, supplying the update formula is trivial, because these statements do not affect the value of the predicate. The only non-trivial case is the assignment to an array element. But, even in this case, it required only an insignificant effort to supply the correct update formula.

7.4 Results

Fig. 11 shows the time it took the analysis prototype to analyze the examples presented in this section. We ran the analysis on an Intel-based Linux machine equipped with a Pentium 4, 2.4Ghz processor and 512Mb of memory.

The analysis times are severely affected by our choice of implementing the analysis prototype in TVLA. Because TVLA is a general framework, the structure of

Example	Time (in seconds)
Array initialization	1.6
Partial initialization	135.8
Array copy	370.1
Insertion sort	49.7

Figure 11: Results of the analysis.

an array has to be modeled explicitly by introducing a number of predicates and integrity constraints. The majority of the analysis time is spent on ensuring that the array structure is preserved. Building a dedicated analysis implementation, in which the abstract state transformers are array-aware, would greatly improve the analysis time.

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[10], 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 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 [9], shows how his numeric domains can be used to represent values stored in a *statically initialized* array, and how that information can be retrieved during analysis. In contrast, our approach allows arrays to be partitioned dynamically into disjoint segments, and values stored in each segment are represented separately. Such dynamic array partitioning, along with the simple partitioning schemes illustrated in this paper, allows our approach to handle certain cases of *dynamic* array initialization.

A team of French researchers, 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 expanding only 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 [5]. 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, in unpublished work [15], uses *parametric predicate abstraction* to define *problem-specific* abstract domains, which he uses to verify array kills and correctness of comparison-based sorting algorithms. His technique augments standard numeric domains by introducing extra numeric quantities that do not correspond to any variables in the program and that describe segments of an array for which certain properties of interest hold. Abstract transformers are then defined for the augmented numeric domain. In contrast, our approach directly separates array segments for which properties of interest hold from the remaining elements, and uses numeric domains to track numeric properties for each group of elements. For instance, in the case of an array initialization, our approach not only detects an array kill, but also automatically discovers the constraints on the initialized values.

Canonical abstraction [14, 8] 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 represent numeric quantities. Also, [6] 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.

9 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 performing 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.

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. October 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] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Symp. on Princ. of Prog. Lang.*, pages 191–202, New York, NY, January 2002. ACM Press.

[6] 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. Springer-Verlag, March 2004.

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

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

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

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

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

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

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

[14] 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.

[15] Pavol Černý. Verification by abstract interpretation of parametrized predicates, 2003. Available at "http://www.cis.upenn.edu/ cernyp/".

[16] 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.