Numeric Domains with Summarized Dimensions

Denis Gopan[1], Frank DiMaio[1], Nurit Dor[2], Thomas Reps[1], and Mooly Sagiv[2]

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

Abstract. We introduce a systematic approach to designing *summarizing abstract numeric domains* from existing numeric domains. Summarizing domains use *summary* dimensions to represent potentially unbounded collections of numeric objects. Such domains are of benefit to analyses that verify properties of systems with an unbounded number of numeric objects, such as shape analysis, or systems in which the number of numeric objects is bounded, but large.

1 Introduction

Verifying the correctness of complex software systems requires reasoning about numeric quantities. In particular, an analysis technique may have to discover certain relationships among values of *numeric objects*, such as numeric variables, numeric array elements, or numeric-valued fields of heap-allocated structures [2]. For example, to verify that there are no buffer overruns in a particular C program, an analysis needs to make sure that the value of an index variable does not exceed the length of the buffer at each program point where the buffer is accessed [16].

Numeric analyses have been a research topic for several decades, and a number of numeric domains that allow to approximate numeric state of a system have been designed over the years. These domains exhibit varying precision/cost tradeoffs, and target different types of numeric properties. The list of existing numeric domains includes: **non-relational domains**: intervals [7, 15], congruences [5]; **weakly relational domains**: difference constraints [4], octagons [11]; **relational domains**: polyhedra [2, 6], trapezoidal congruences [10].

Existing numeric domains are able to keep track of only a fixed number of numeric objects. Traditionaly, a finite set of stack-allocated numeric variables deemed *important* for the property to be verified is identified for the analysis. The remaining numeric objects, e.g., numeric array elements or heap-allocated numeric objects, are modeled conservatively.

Two problems that plague existing numeric domains are:

- It may be impossible to verify certain numeric properties by considering only a fixed number of the numeric objects in the system. For example, in programs that use collections (or, in general, dynamic memory allocation), it is impossible to determine statically the set of memory locations used by a program.
- The resources required for higher-precision relational numeric domains, such as polyhedra, are subject to combinatorial explosion. This is due to the representation of elements of the numeric domain; for instance, the number of elements in the frame representation of a polyhedron grows exponentially with the number of tracked objects.

The typical approach to reasoning about an unbounded number of objects (or simply a very large number of objects) is by employing abstraction. The set of objects is divided into a fixed number of groups based on certain criteria. Each group is then represented (*summarized*) by a single abstract object. The groups themselves need not be of bounded size. As an example, consider TVLA, a framework for shape analysis, which

```
(1) for(i = 0; i < n; i++) {
(2)    t = 0;
(3)    for(k = ia[i]; k < ia[i+1]; k++) {
(4)       j = ja[k];
(5)       t += a[k] * x[j];
(6)    }
(7)    y[i] = t;
(8) }
```

(a)

$0 \le ja[\cdot] < n$
$0 \le ia[\cdot] \le nnz$

(b)

$0 \le ja[\cdot] < n$
$0 \le ia[\cdot] \le nnz$
$0 \le k < nnz$
$0 \le i < n$
$0 \le j < n$

(c)

Fig. 1. Multiplication of a sparse matrix by a vector: (a) multiplication code; (b) initial constraints, imposed by *CSR* format; (c) constraints that represent the abstract numeric state at line 5.

concerns the problem of determining "shape invariants" for programs that perform destructive updating on dynamically allocated storage. TVLA uses so-called "canonical abstraction" to create bounded-size representations of memory states [14, 9]. To reason about numeric properties in such a *summarizing* framework, an analysis needs to be able to capture the relationships among values of *groups* of numeric objects, rather than relationships among values of *individual* numeric objects. Moreover, what appears to be a natural way of applying existing techniques is, in fact, unsound.

In this paper, we define a systematic approach for extending existing numeric domains to support arbitrary summarization. The domain to be extended must support four extra operations: *expand*, *fold*, *add*, and *drop* which will be disccused in detail in later sections. The extended numeric domain exposes a standard numeric-domain interface. For purposes of this paper, we assume that a client analysis has the responsibility of indicating which abstract objects are summary ones. At one extreme, if the analysis does not do any summarization, the extended domain operates exactly as the original domain. In the presence of summarization, the extended domain produces safe answers, with minimal loss of precision.

The program in Fig. 1(a) multiplies an $(n \times n)$ sparse matrix stored in the *compressed sparse row (CSR)* format [13] by a vector x of size n. In the *CSR* format, a sparse matrix is represented using three arrays:

- the array a, of size nnz, stores the matrix elements row by row (nnz is the number of non-zero elements in the matrix);
- the array ja, of size nnz, stores the column indices for the elements in a; thus, *CSR* format imposes the constraint $0 \le ja[\cdot] < n$ on each element of the array ja;
- the array ia, of size $n+1$, stores the offsets to the beginning of each row in the arrays a and ja; thus, *CSR* format imposes the constraint $0 \le ia[\cdot] \le nnz$ on each element of the array ia.

Note that, if the matrix is properly represented in *CSR* format, both array acceses on line 5 (shown in bold) will never be out of bounds. Yet, most existing analyses are not able to verify this property because array indices k and j are computed through the use of *indirection* arrays, ia and ja.

We used the summarizing extension of the polyhedral numeric domain [2, 6] to verify that both array accesses in line 5 are, indeed, always in bounds, if the constraints imposed by *CSR* format hold initially. We used two *summary* dimensions to represent all elements of the arrays ia and ja, respectively. Such summarization allowed us to represent the abstract numeric states of the program as 7-dimensional polyhedra. It follows directly from the constraints shown in Fig. 1(c) that the values of both indices j and k at line 5 are within the bounds of the corresponding arrays.

The contributions this paper makes are

– The extended numeric domains constructed using our technique support arbitrary summarization, which makes them suitable for a wide range of client analyses: on the one hand, the analysis could be as simple as summarizing the values of all elements of an array; on the other hand, it could be as involved as using canonical abstraction for summarization (which would allow multiple, dynamically changing segments of an array to be summarized separately—see Sect. 7).
– The requirements that we place on the numeric domain to be extended are minimal: a safe implementation of four operations must be provided. Because of this flexibility, the numeric domain that is most suitable for a problem can be employed by a client analysis.
– When coupled with a suitable client, such as TVLA, our extended numeric domains are able to operate on unbounded collections of numeric objects.
– For large fixed-size collections of numeric objects, our extended numeric domains allow trading some precision for analysis efficiency (in terms of both memory usage and running time).

As will be illustrated in Ex. 1, what looks like a natural approach to performing operations over values in an extended numeric domain is actually unsound. The formalization of a sound approach is the major technical contribution of this paper.

The remainder of the paper is organized as follows: Sect. 2 discusses concrete semantics. Sect. 3 introduces the concept of reducing dimensionality via summarization. Sect. 4 describes what is involved in extending a standard numeric domain (i.e., as long as it meets certain assumptions about various primitive operations). Sect. 5 describes how to perform safe computations on values of extended numeric domains. Sect. 6 discusses related work. Sect. 7 sketches the application of this technique to a situation in which multiple, dynamically changing segments of an array are summarized separately.

2 Concrete Semantics

Our goal is to perform static analysis of systems in which the number of numeric objects (i) may change as the system operates, and (ii) cannot be bounded statically. A concrete numeric state of the system, denoted $S^\natural$, is an assignment of a value to each numeric object. In each particular state the number of numeric objects is finite, and will be denoted as $N^{S^\natural}$, or N when $S^\natural$ is clear from context. We denote the set of objects in state $S^\natural$ by $Obj^{S^\natural} = \{v_1, ..., v_N\}$. Following the traditional numeric analysis approach, we associate each numeric object with a dimension of an N-dimensional space, and encode a concrete numeric state of the system as an N-dimensional point. We use a function $dim^{S^\natural} : Obj^{S^\natural} \to \{1, ..., N\}$ to map numeric objects to corresponding dimensions.

Let $\mathbb{V}$ denote a set of possible numeric values (such as $\mathbb{Z}$ or $\mathbb{Q}$). Because the number of numeric objects, and hence the number of dimensions, is not bounded a priori, a concrete numeric state may be a point in a space of arbitrary (finite) dimension:

$$S^\natural \in \mathbb{V}^+, \quad \text{where } \mathbb{V}^+ = \bigcup_{k=1}^{\infty} \mathbb{V}^k$$

Given an expression $e(w_1, ..., w_k)$, where $w_i \in Obj^{S^\natural}$, we evaluate it at a concrete numeric state $S^\natural$ as follows: let $x[i]$ denote the i-th component of a vector $x \in \mathbb{V}^N$; we define

$$[\![e(w_1, ..., w_k)]\!]_\natural(S^\natural) = e(S^\natural[dim^{S^\natural}(w_1)], ..., S^\natural[dim^{S^\natural}(w_k)]).$$

To each program point we attach a set of concrete states $D^\natural \subseteq \mathbb{V}^+$. We define the program's concrete collecting semantics using the following transformers:

– **Numeric tests** filter the set of concrete states:

$$[\![e(w_1, ..., w_k)?]\!]_\natural(D^\natural) = \left\{S^\natural \in D^\natural : [\![e(w_1, ..., w_k)]\!]_\natural(S^\natural) = true\right\}$$

– **Assignments** change the value of a single numeric object (in each concrete state):

$$\begin{aligned}&[\![v_i \leftarrow e(w_1, ..., w_k)]\!]_\natural(D^\natural) = \\ &\quad = \left\{ \bar{S}^\natural : \exists S^\natural \in D^\natural \text{ s.t. } \begin{bmatrix} Obj^{\bar{S}^\natural} = Obj^{S^\natural},\ dim^{\bar{S}^\natural} \equiv dim^{S^\natural} \\ \bar{S}^\natural[dim^{S^\natural}(v_i)] = [\![e(w_1, ..., w_k)]\!]_\natural(S^\natural), \\ \bar{S}^\natural[j] = S^\natural[j] \text{ for } j \neq dim^{S^\natural}(v_i) \end{bmatrix} \right\}\end{aligned}$$

– **Union** collects the sets at control flow merge points.

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

Existing numeric analyses identify overapproximations of the sets of concrete states that arise at program points using a set representation that can be easily stored and manipulated in a computer. Such value spaces are called *numeric domains*. The assumption that existing numeric domains make is that the number of numeric objects is fixed throughout the analysis, thus allowing sets of concrete states to be represented as subsets of a space with a fixed number of dimensions.

In the semantics formulated above, however, the concrete numeric states arising at a given program point belong to spaces of possibly different numbers of dimensions. Therefore, existing numeric domains cannot be used directly. In the next section, we show how a set of points in spaces of different dimensionalities can be abstracted by a subset of a space with a fixed, smaller number of dimensions.

3 Summarizing Numeric Domains

In this section, we introduce a numeric domain that uses a subset of a space with a fixed number of dimensions to overapproximate a set of points in spaces of different dimensionalities. The idea behind the abstraction is that some dimensions of the fixed-dimensional space will represent the values of potentially unbounded collections of numeric objects, rather than the values of individual objects. The numeric domain is not able to differentiate between the individual objects that are members of such collections, and preserves only overall properties of the collection, e.g., lower and upper bounds on the values of its members.

We call the process of grouping numeric objects into a collection *summarization*. In this paper, for the sake of simplifying the presentation, we assume that each client analysis that uses one of our summarizing numeric domains is responsible for defining

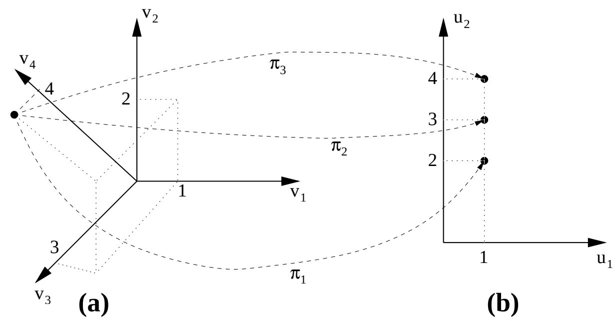


Fig. 2. Abstraction example: (a) concrete numeric state $S^\natural$; (b) abstraction of $S^\natural$. $F_{sum} = [v_1 \mapsto u_1, v_2 \mapsto u_2, v_3 \mapsto u_2, v_4 \mapsto u_2]$ and $\Pi_{F_{sum}} = \{\pi_1, \pi_2, \pi_3\}$.

which numeric objects are to be summarized, and which collections are to be formed. In the simplest case, as illustrated in the introduction, all elements of an array may be summarized by a single dimension. Ultimately, our goal is to implement more complicated summarizations, including canonical abstraction [14], which would allow parts of an array to be summarized, and would let the summarization partition on array elements change during the course of the analysis; see Sect. 7 for more on combining a client analysis that uses canonical abstraction with a summarizing numeric domain.

Formally, we assume that the concrete numeric objects are separated into M groups, where M is determined by the client analysis. Some groups may contain just one numeric object, while others may contain a set of objects of a priori unbounded size. In the abstract numeric state S, each group is represented by an abstract numeric object. We denote the set of abstract numeric objects by $Obj^S = \{u_1, ...u_M\}$. The abstract objects that represent groups with more than one element are called *summary* objects. For describing and justifying our techniques, we will refer to (conceptual) mappings that map numeric objects of a concrete state $S^\natural$, to the corresponding numeric objects of an abstract state S; such a mapping is called a *summarization* function, e.g., $F_{sum} : Obj^{S^\natural} \to Obj^S$.

A *summarizing abstract numeric domain* represents an abstract numeric state S as a subset of M-dimensional space, where each dimension corresponds to an abstract numeric object. Function $dim^S : Obj^S \to \{1, ..., M\}$ maps an abstract object to its corresponding dimension.

The *abstraction* of a concrete numeric state is an M-dimensional subset constructed by *folding* together the *summarized* dimensions. For example, as illustrated in Fig. 2, if $S^\natural = (1, 2, 3, 4) \in \mathbb{V}^4$ and the summarization function is defined as

$$F_{sum} = [v_1 \mapsto u_1, v_2 \mapsto u_2, v_3 \mapsto u_2, v_4 \mapsto u_2]$$

the abstraction of $S^\natural$ is $S = \{(1, 2), (1, 3), (1, 4)\} \subseteq \mathbb{V}^2$. Note that the abstraction loses the distinction between the values of summarized numeric objects; e.g., under the above summarization function, S is also an abstraction for the concrete states $(1, 3, 2, 4)$ and $(1, 4, 3, 2)$.

Let $S^\natural \in \mathbb{V}^N$ be a concrete numeric state, whose abstraction is $S \subseteq \mathbb{V}^M$. Each point $x \in S$ is constructed by taking an *orthogonal projection* $\pi : \mathbb{V}^N \to \mathbb{V}^M$ of $S^\natural$, such that

$$x[dim^S(u)] = \pi(S^\natural)[dim^S(u)] = S^\natural[dim^{S^\natural}(v)]$$

where $u \in Obj^S$, $v \in Obj^{S^\natural}$ and $u = F_{sum}(v)$. We denote the set of all such projections as $\Pi_{F_{sum}}$. As shown in Fig. 2, given the above summarization function, the set $\Pi_{F_{sum}}$ contains three projections: π_1 projects the dimensions corresponding to v_1 and v_2, π_2 projects the dimensions corresponding to v_1 and v_3, and π_3 projects the dimensions corresponding to v_1 and v_4.

Formally, given a summarization function F_{sum}, we say that an abstract state $S \subseteq \mathbb{V}^M$ *represents* a concrete state $S^\natural$ (denoted by $S^\natural \sqsubseteq^{F_{sum}} S$) iff:

$$S \supseteq \left\{ x \in \mathbb{V}^M : x = \pi(S^\natural) \text{ for some } \pi \in \Pi_{F_{sum}} \right\}$$

Sometimes, when F_{sum} is not important for the discussion, we will omit it from the notation and write $S^\natural \sqsubseteq S$. It is easy to see that abstract states form an infinite-height lattice ordered by set inclusion, where the *bottom* element is the empty set, and the *join* and the *meet* operations correspond to set union and set intersection, respectively.

Given a set of concrete states $D^\natural \subseteq \mathbb{V}^+$, and an abstract state $S \subseteq \mathbb{V}^M$, we say that $D^\natural \sqsubseteq S$ if S abstracts every element in $D^\natural$, i.e., $S^\natural \sqsubseteq S$ for all $S^\natural \in D^\natural$.

We use existing numeric domains to store and manipulate M-dimensional subsets that correspond to abstract states. Numeric domains impose certain restrictions on the subsets they are able to represent; therefore, it may be impossible to represent an abstract numeric state precisely. In such cases, an overapproximation of that abstract state is represented. For example, the abstract state $\{(1,2),(1,3),(1,4)\} \subseteq \mathbb{V}^2$ cannot be represented as a polyhedron directly, but can be approximated by the polyhedron

$$\left\{ x \in \mathbb{V}^2 : x[1] = 1 \text{ and } 2 \leq x[2] \leq 4 \right\}.$$

Because these sets are conceptually different from the sets that occur in the existing numeric domain, we need to extend the semantics of the existing numeric domain in order to be able to compute safely with the abstract states of the extended numeric domain. In the next section, we describe the operations that a given numeric domain must support to be used with our abstraction technique.

4 Extending Numeric Domains

As was pointed out in the introduction, a number of numeric domains have been designed, and undoubtedly more will be proposed in the future. These domains target different numeric properties, and exhibit different precision/cost tradeoffs. What is common among all of them is that they use a compact representation for a subset of a multidimensional space, and define a number of operations that allow a client analysis to (i) evaluate certain kinds of numeric conditions, and (ii) transform an underlying subset according to the semantics of a program statement.

The assumption that all existing numeric domains make is that each point within the represented subset corresponds to a concrete numeric state of the system, and vice versa. This assumption is relied on when proving the correctness of the implementations of operations for transforming the set, and for evaluating conditions. In contrast, in our abstraction a concrete state corresponds to a *collection* of points within the subset represented by a numeric domain, and each individual point within the represented subset may belong to the abstraction of *multiple* concrete states.

To provide for sound evaluation of conditions and set transformations induced by program statements, the numeric domain needs to be extended with several extra operations that map between subsets of spaces of different dimensionality. In this section, we give a detailed description of these operations, and show how they can be implemented for several existing numeric domains.

4.1 Standard semantics of a numeric domain

Let us define the standard *interface* of a numeric domain. The operations that the numeric domain exposes are abstract state transformers that manipulate subsets of N-dimensional space. As mentioned above, the semantics of the state transformers is defined under the assumption that each concrete numeric state corresponds to a single point within the abstract state and vice versa.

Let d_i denote the i-th dimension of $\mathbb{V}^N$. Let formula $e(w_1, ..., w_k)$, where $w_i \in \{d_1, ..., d_N\}$, denote either a numeric condition or a numeric computation. We will use the following notation to denote the standard numeric domain operations

- **Numeric tests:** $[\![e(w_1, ..., w_k)?]\!]_{std}$
- **Assignments:** $[\![d_i \leftarrow e(w_1, ..., w_k)]\!]_{std}$
- **Join:** $\sqcup_{std}$
- **Widening operator:** ∇_{std}

The detailed definitions of the abstract state transformers can be found in the corresponding papers [7, 15, 4, 11, 5, 2, 10].

The operations defined below insert and remove arbitrary dimensions of the multidimensional space. After each such operation, dimensions with indices above that of the inserted or removed dimension have to be renumbered. To somewhat simplify the presentation by avoiding recomputation of dimension indices within the operation definitions, we introduce a function $[\cdot]'$ that maps the dimensions of the original space to the corresponding dimensions of the resulting space after the j-th dimension has been removed (used by operations *fold* and *drop*). Similarly, we introduce a dimension mapping $[\cdot]''$ for inserting the new dimension j (used by operations *expand* and *add*).

$$k' = \begin{cases} k-1 & \text{if } k > j \\ k & \text{if } k < j \\ \text{undefined} & \text{if } k = j \end{cases} \qquad\qquad k'' = \begin{cases} k+1 & \text{if } k \geq j \\ k & \text{if } k < j \end{cases}$$

4.2 The *fold* operation

The *fold* operation formalizes the concept of folding dimensions together. Let $S \subseteq \mathbb{V}^N$. The $\mathit{fold}_{i,j}$ transforms S into a subset of $\mathbb{V}^{N-1}$ by folding dimension j into dimension i. For an arbitrary subset of $\mathbb{V}^N$, $\mathit{fold}_{i,j}$ is defined as follows:

$$\mathit{fold}_{i,j}(S) = \left\{ x \in \mathbb{V}^{N-1} : \exists y \in S \text{ s.t. } \begin{bmatrix} (x[i'] = y[i]) \vee (x[i'] = y[j]) \\ \wedge\ \forall k \neq i, j\ \left[x[k'] = y[k]\right] \end{bmatrix} \right\}$$

If multiple dimensions need to be folded together, we construct the corresponding transformation by composing several *fold* operations.

Note that the *fold* operation as defined above is not closed in most existing numeric domains. For example, consider a two-dimensional polyhedron

$$P = \left\{ x \in \mathbb{V}^2 : 1 \leq x[1] \leq 3 \text{ and } 7 \leq x[2] \leq 12 \right\}$$

Clearly, $\mathit{fold}_{1,2}(P) = \{x \in \mathbb{V} : (1 \leq x \leq 3) \vee (7 \leq x \leq 12)\}$ is not a polyhedron. For such domains, we define $[\![\mathit{fold}_{i,j}]\!]_{std}(S)$ to be an overapproximation of the set $\mathit{fold}_{i,j}(S)$ that is representable in that domain, e.g., for polyhedral domain, $[\![\mathit{fold}_{1,2}]\!]_{std}(P) = \{x \in \mathbb{V} : 1 \leq x \leq 12\}$.

4.3 The *expand* operation

The *expand* operation is essentially the opposite of the *fold* operation. Let $S \subseteq \mathbb{V}^N$. The $\mathit{expand}_{i,j}$ transforms S into a subset of $\mathbb{V}^{N+1}$ by creating an exact copy of the i-th dimension and inserting it as a new dimension j. For an arbitrary subset of $\mathbb{V}^N$, the $\mathit{expand}_{i,j}$ operation is defined as follows:

$$\mathit{expand}_{i,j}(S) = \left\{ x \in \mathbb{V}^{N+1} : \exists y, z \in S \text{ s.t. } \left[\begin{array}{c} x[i''] = y[i] \wedge x[j] = z[i] \\ \wedge\, \forall\, k \neq i \big[x[k''] = y[k] = z[k] \big] \end{array} \right] \right\}$$

For instance, for the example from Sect. 3, $\mathit{expand}_{2,3}(\{(1,2),(1,3),(1,4)\})$ is equal to $\{(1,2,2),(1,2,3),(1,2,4),(1,3,2),(1,3,3),(1,3,4),(1,4,2),(1,4,3),(1,4,4)\}$. More complex expansions can be constructed by composing several $\mathit{expand}_{i,j}$ operations.

Let $S \subseteq \mathbb{V}^N$. Note that the $\mathit{expand}_{i,j}(S)$ constructs the *maximal* subset $S' \subseteq \mathbb{V}^{N+1}$, such that $fold_{i,j}(S')$ is S. In fact, *fold* and *expand* form a Galois insertion:

$$\mathit{expand}_{i,j} \circ \mathit{fold}_{i,j}(S) \supseteq S \quad \text{and} \quad \mathit{fold}_{i,j} \circ \mathit{expand}_{i,j}(S) = S$$

Unlike *fold*, the *expand* operation is closed on all of the existing numeric domains we have experimented with so far, and is likely to be closed for any numeric domain. Our conviction is based on the fact that the values along the i-th dimension of the original subset are precisely represented by the numeric domain. Therefore, the values along the newly introduced j-th dimension can also be precisely represented in that domain.

4.4 The *add* and *drop* operations

The *add* and *drop* are two auxiliary operations that add new dimensions and remove specified dimensions from a multidimensional space. Let $S \subseteq \mathbb{V}^N$. The $\mathit{add}_j(S)$ embeds S into $\mathbb{V}^{N+1}$ after inserting a new dimension, j, into $\mathbb{V}^N$.

$$\mathit{add}_j(S) = \big\{ x \in \mathbb{V}^{N+1} : \exists y \in S \text{ s.t. } x[j] \in \mathbb{V} \text{ and } \forall\, k \big[x[k''] = y[k] \big] \big\}$$

The $\mathit{drop}_j(S)$ computes the projection of S onto $\mathbb{V}^{N-1}$, removing the dimension j, from $\mathbb{V}^N$.

$$\mathit{drop}_j(S) = \big\{ x \in \mathbb{V}^{N-1} : \exists y \in S \text{ s.t. } \forall\, k \neq j \big[x[k'] = y[k] \big] \big\}$$

The add_j operation is closed in all numeric domains and can be implemented precisely. The drop_j operation is closed in the numeric domains that we have experimented with; furthermore, we were able to design precise implementations of $[\![\mathit{drop}_j]\!]_{std}$ for those domains. Thus, for the sake of the minimality arguments given in Sect. 5, we assume that the numeric domain to be extended must furnish a precise implementation of the *drop* operation.

4.5 Implementation examples

In this section, we show how to implement the above operations for several existing numeric domains. For this discussion we have chosen numeric domains that use a diverse set of representations: the non-relational *interval* domain [7, 15], the fully relational *polyhedral* domain [2, 6], and a family of *weakly-relational* domains constructed in [12].

In the *interval* domain, the subset of a multidimensional space is represented by maintaining upper and lower bounds for values along each dimension, i.e., an N-dimensional subset is represented by an ordered set I of N intervals. The $[\![\mathit{add}_j]\!]_{int}(I)$ operation is defined by inserting interval $[-\infty, \infty]$ as the new j-th interval into I. The $[\![\mathit{drop}_j]\!]_{int}(I)$ operation removes the j-th interval from I. The operation $\mathit{expand}_{i,j}$ is defined as follows:

$$[\![\mathit{expand}_{i,j}]\!]_{int}(I) = J \text{ where } J[j] = I[i] \text{ and } \forall k \big[J[k''] = I[k]\big]$$

The $\mathit{fold}_{i,j}$ operation is not closed on the interval domain, hence we define $[\![\mathit{fold}_{i,j}]\!]_{int}(I)$ to overapproximate the resulting set as follows:

$$[\![\mathit{fold}_{i,j}]\!]_{int}(I) = J \text{ where } J[i'] = I[i] \sqcup_{int} I[j] \text{ and } \forall k \neq i, j \big[J[k'] = I[k]\big]$$

In the *polyhedral* domain, the subset of the multidimensional space is represented as an intersection of a finite set of linear constraints, i.e., by a polyhedron. Most polyhedral libraries, such as Parma [1], provide routines for adding and removing dimensions. The operation $[\![\mathit{add}_j]\!]_{poly}$ and $[\![\mathit{drop}_j]\!]_{poly}$ may be implemented by direct invocation of those routines. A little extra care may be necessary to maintain the proper numbering of the dimensions. Also, note that the operation $[\![\mathit{drop}_j]\!]_{poly}$ directly corresponds to *Fourier-Motzkin elimination* [3] of dimension j.

The $[\![\mathit{expand}_{i,j}]\!]_{poly}(P)$ operations is implemented by augmenting the set of constraints that represent P: for each linear constraint involving dimension i, we add a similar constraint with dimension j substituted for dimension i (and all constraints are remapped according to $[\cdot]''$). The $fold_{i,j}$ operation is not closed for polyhedra, therefore we define $[\![\mathit{fold}_{i,j}]\!]_{poly}(P)$ to compute an overapproximation of the set $fold_{i,j}(P)$ as follows:

$$[\![\mathit{fold}_{i,j}]\!]_{poly}(P) = [\![\mathit{drop}_j]\!]_{poly}(P \sqcup_{poly} [\![d_i \leftarrow d_j]\!]_{poly}(P))$$

In *weakly-relational* domains, a subset of $\mathbb{V}^N$ is represented by a set of constraints of the form $d_j - d_i \in C$, where d_j and d_i refer to the dimensions j and i, respectively. C is an element of a non-relational *base* numeric domain, such as the interval domain. A weakly-relational domain maintains an $N \times N$ matrix m, where each element $m_{ij} \in C$ encodes a constraint $d_j - d_i \in m_{ij}$. A *closure* operation $[\cdot]^\star$ that propagates the constraints through the matrix is defined. For more details, see [12].

The $[\![\mathit{add}_j]\!]_{wr}$ operation is implemented by inserting the j-th row and j-th column into the matrix m. The elements of the inserted row and column are initialized to the $\top$ element of the corresponding non-relational base domain. The $[\![\mathit{drop}_j]\!]_{wr}$ operation is implemented by removing the j-th row and j-th column from the *closed* matrix $m^\star$.

The operation $[\![expand_{i,j}]\!]_{wr}$ is implemented by inserting copies of the i-th row and i-th column into matrix $m^\star$ as the (new) j-th row and j-th column, respectively; the elements m_{ij}, m_{ji}, and m_{jj} are set to $\top$. The operation $[\![fold_{i,j}]\!]_{wr}$ is implemented by using $m^\star$ and recomputing the elements of its i-th row and i-th column by taking the join of their value with the value of the corresponding elements in the j-th row and j-th column. The $[\![drop_j]\!]_{wr}$ operation is applied to the resulting matrix.

5 Abstract Semantics

In existing numeric domains, each point in the abstract state corresponds to a unique concrete state. Therefore, each point in the abstract state contains enough information to compute how the corresponding concrete state is transformed; consequently it can be transformed independently of other points in the abstract space. Hence, using appropriate finite representations for subsets of N-dimensional space (e.g., polyhedra), the abstract state transformer can be defined as an operation that is applied to all points in the subset simultaneously (e.g., a linear transform) to produce the resulting abstract state.

In our case, there is no such correspondence between the concrete states and individual points in the abstract state. Instead, each concrete state corresponds, in general, to a subset of the abstract state, and each point in the abstract state belongs to the abstractions of a (possibly infinite) set of concrete states. Hence, the points within the abstract state cannot be used independently when applying a transformation. However, we will now show that by making appropriate calls to *expand* and *fold*, it is possible to use existing numeric domains to compute the results of transformations safely and precisely.

In general, a transformation consists of two steps. First, for each point in the abstract state we compute the values to which the transformation formula evaluates in the concrete states corresponding to this point. Second, given the values computed in the first step, we update each point. Ordinarily, the standard semantics of a numeric domain is able to combine the two steps, because each point in the abstract state corresponds to a single concrete state and vice versa, i.e., (i) a transformation formula evaluates to a single value for each point in the abstract state, (ii) each point in the original abstract state corresponds to a single point in the transformed abstract state.

In our situation, each point within the abstract state corresponds to the *projections* of one or more concrete states, rather than to a single concrete state, and may not contain all the values necessary to evaluate the transformation formula. Therefore, applying the standard semantics of the numeric domain may produce an *unsound* result. We illustrate this situation with the following example.

Example 1. Consider a concrete state $S^\natural$ with four numeric objects x, y_1, y_2, and y_3. Let $S^\natural = (1, 2, 3, 4)$. Consider an abstraction S in which y_1, y_2, and y_3 are folded into a summary numeric object y: $S = \{(1, 2), (1, 3), (1, 4)\}$. Let the transformation formula be $x \leftarrow y_2$. Evaluating this formula by binding y_2 to the summary dimension corresponding to y and treating each point in S independently, results in $S' = \{(2, 2), (3, 3), (4, 4)\}$. Now, applying the transformation to $S^\natural$ yields a concrete numeric state $(3, 2, 3, 4)$ whose abstraction $\{(3, 2), (3, 3), (3, 4)\}$ is clearly not a subset of S'.

Intuitively, the problem occurs because for a given point x in S, we failed to compute the set of *all* values to which the transformation formula evaluates in concrete states that have x as a projection. Hence, to be able to treat the points within the abstract state independently, we need to overcome the following problems:

- **The evaluation problem.** For each point in the abstract state S, we need to compute the set of all values to which the transformation formula evaluates, across all of the concrete states $S^\natural \sqsubseteq S$ whose abstraction includes that point. The problem is that to compute these values we may need information from other points within the abstract state.
- **The update problem.** Given the above set of values for a particular point in the abstract state, we need to define how to update that point. The problem is that the point needs to be updated differently for each value in the set. Therefore, a single point in the initial abstract state may produce a potentially infinite number of points within the transformed abstract state.

Example 2. Assume the same situation as in Ex. 1. Concrete numeric object y_2 is represented by a summary numeric object in the abstract domain. Therefore, for each point in S, the values that y_2 can take on are $\{2, 3, 4\}$. Transforming each point according to each value of y_2, we get the transformed abstract state $S' = \{(\alpha, \beta) : \alpha, \beta = 2, 3, \text{ or } 4\}$. It is easy to see that S' abstracts all possible concrete states that may arise as a result of the transformation of concrete states abstracted by S.

In the following discussion, we will often refer to concrete states whose abstraction includes a particular point within the abstract state. We introduce a concise notation to simplify the presentation: suppose that $S \subseteq \mathbb{V}^M$ is an abstract state; let x be a point in S, and let $S^\natural \sqsubseteq S$ be a concrete state whose abstraction contains x. We denote this relationship as $S^\natural \sqsubseteq_x S$.

5.1 General overview of the approach

Let $e(w_1, ..., w_k)$ be a numeric formula, where each w_i denotes a concrete numeric object. Each concrete numeric object w_i in the formula corresponds to either a summary or non-summary abstract numeric object. Without loss of generality, we assume that first $\hat{k}$ of the w_i's, where $0 \le \hat{k} \le k$, correspond to summary numeric objects.

Let $S \subseteq \mathbb{V}^M$ be an abstract numeric state, and let x be a point in S. We denote the set of values that $e(w_1, ..., w_k)$ evaluates to in all concrete numeric states $S^\natural \sqsubseteq_x S$ as

$$Values_e^S(x) = \left\{ [\![e(w_1, ..., w_k)]\!]_\natural (S^\natural) : S^\natural \sqsubseteq_x S \right\}$$

Given a test $e(w_1, ..., w_k)?$, or an assignment $v_i \leftarrow e(w_1, ..., w_k)$, we transform an abstract numeric state S in three steps:

- **Preparation step.** First, we construct a set $S_e \subseteq \mathbb{V}^{M+\hat{k}}$, which allows us to compute the set $Values_e^S(x)$ for each point $x \in S$ by using the standard semantics associated with the numeric domain. We construct S_e by creating exact copies of the dimensions of S that correspond to the summary abstract numeric objects $F_{sum}(w_1), ..., F_{sum}(w_{\hat{k}})$. The detailed description of this construction is given in Sect. 5.2.
- **Transformation step.** Next, we use the standard semantics of the numeric domain to perform the transformation of S_e. Certain care is necessary to handle the assignments to summary numeric objects, because these objects also represent the concrete numeric objects whose value is not changed by the assignment. For such

assignments, we introduce an extra dimension to capture the new values that are assigned to the numeric object, and then combine the new values with the old values by folding this extra dimension into the dimension that corresponds to the object. The details are covered in Sects. 5.3, 5.4, and 5.5.

– **Clean-up step.** Finally, we remove from S_e the dimensions introduced in the preparation step to produce an M-dimensional subset that corresponds to the updated abstract state, S'.

Because the clean-up step returns us to a situation in which $\Pi_{F_{sum}}$ defines a mapping from concrete states in $\mathbb{V}^N$ to abstract states (subsets of $\mathbb{V}^M$), the standard numeric domain join operation, $\sqcup_{std}$, can be used to combine numeric states at control flow merge points. The standard semantics of a widening operator, ∇_{std}, is safe with respect to the abstraction. Thus, for brevity, we will not discuss widening in this paper.

5.2 Evaluation of numeric formulas

Let $S \in \mathbb{V}^M$ be an abstract state, and let $e(w_1, ..., w_k)$ be a numeric formula. We will show how to construct the set S_e, so that it is possible to compute $Values_e^S(x)$ for all $x \in S$ by applying standard numeric domain operations to S_e.

Let $S^\natural \sqsubseteq S$. The value $[\![e(w_1, ..., w_k)]\!]_\natural(S^\natural)$ is completely determined by the values that the w_i's have in $S^\natural$. Thus, to be able to use the standard semantics of the numeric domain, we will extend S by adding k extra dimensions, and put all combinations of values w_i's have in $S^\natural \sqsubseteq_x S$ into the new dimensions for each $x \in S$,

$$S_e = \left\{ y \in \mathbb{V}^{M+k} : \begin{array}{c} x = (y[1], ..., y[M]) \in S,\ S^\natural \sqsubseteq_x S, \text{ and} \\ y[M+j] = S^\natural[dim^{S^\natural}(w_j)] \text{ for } j = \{1, ..., k\} \end{array} \right\}$$

Now we can compute $Values_e^S(x)$ for all points $x \in S$ in parallel by evaluating the formula $e(d_{M+1}, ..., d_{M+k})$ on S_e using the standard semantics of the $(M+k)$-dimensional numeric domain.

We can simplify the set S_e somewhat. Let $x \in S$. It follows from the abstraction that if $F_{sum}(w_i)$ is a non-summary abstract numeric object, then for all $S^\natural \sqsubseteq_x S$,

$$S^\natural[dim^{S^\natural}(w_i)] = x[dim^S(F_{sum}(w_i))]$$

Therefore, we do not need to create extra dimensions for w_i that correspond to non-summary abstract objects (i.e., $i > \hat{k}$). The simplified definition of S_e is

$$S_e = \left\{ y \in \mathbb{V}^{M+\hat{k}} : \begin{array}{c} x = (y[1], ..., y[M]) \in S,\ S^\natural \sqsubseteq_x S \text{ and} \\ y[M+j] = S^\natural[dim^{S^\natural}(w_j)] \text{ for } j \in \{1, ..., \hat{k}\} \end{array} \right\}$$

This definition of S_e allows us to compute the set $Values_e^S(x)$ for all points $x \in S$ by evaluating the formula $e(d_{M+1}, ..., d_{M+\hat{k}}, d_{dim^S(F_{sum}(w_{\hat{k}+1}))}, ..., d_{dim^S(F_{sum}(w_k))})$ on S_e using the standard semantics of the $(M+\hat{k})$-dimensional numeric domain. For brevity, we will refer to the above formula as $e(\bar{d})$ in the remainder of the paper.

The dimensions $M+j$ of S_e, where $j \in \{1, ..., \hat{k}\}$, are constructed by creating exact copies of dimensions $dim^S(F_{sum}(w_j))$. We use a composition of *expand* operations, denoted *expand*$_e$, to construct them:

$$\begin{aligned} S_e &= [\![\textit{expand}_e]\!]_{std}(S) \\ &= [\![\textit{expand}_{dim^S(F_{sum}(w_{\hat{k}})),M+\hat{k}}]\!]_{std} \circ ... \circ [\![\textit{expand}_{dim^S(F_{sum}(w_1)),M+1}]\!]_{std}(S) \end{aligned}$$

After the transformation is applied to the set S_e, we need to project the transformed set S'_e back into M-dimensional space. We define the operation *drop*$_e$ as a composition of *drop* operations to remove the dimensions $d_{M+1}, ..., d_{M+\hat{k}}$,

$$S' = [\![\textit{drop}_e]\!]_{std}(S'_e) = [\![\textit{drop}_{M+1}]\!]_{std} \circ ... \circ [\![\textit{drop}_{M+\hat{k}}]\!]_{std}(S'_e)$$

5.3 Numeric tests

Let $S \in \mathbb{V}^M$ be an abstract state and let $e(w_1, ..., w_k)$ be a numeric condition. We want to construct the most-precise abstract state S', such that for any concrete state $S^\natural \sqsubseteq S$ in which $e(w_1, ..., w_k)$ holds, $S^\natural \sqsubseteq S'$. Let $Obj^{S'} = Obj^S$ and $dim^{S'} \equiv dim^S$. We define the abstract transformer as follows:

$$\begin{aligned} S' &= [\![e(w_1, ..., w_k)?]\!](S) = \left\{x : x \in S \text{ and } true \in Values_e^S(x)\right\} \\ &= [\![\textit{drop}_e]\!]_{std} \circ [\![e(\bar{d})?]\!]_{std} \circ [\![\textit{expand}_e]\!]_{std}(S) \end{aligned}$$

We argue that the above transformation is sound. Let $S^\natural \sqsubseteq S$ be an arbitrary concrete numeric state such that the condition $e(w_1, ..., w_k)$ holds in $S^\natural$. By the definition of set $Values_e^S(x)$, it follows that $true \in Values_e^S(x)$ for all points x in the abstraction of $S^\natural$. Hence, by the definition of the transformation, the entire abstraction of $S^\natural$ is in S'. Therefore, $S^\natural \sqsubseteq S'$. The equality on the second line of the definition is justified by the discussion of how to compute set $Values_e^S(x)$ in Sect. 5.2.

Also, we argue that the result of the transformation is minimal in the sense that no points can be excluded from S'. Note that, by construction, for all points $x \in S'$, the set $Values_e^S(x)$ contains the value $true$. Hence there exists at least one concrete state $S^\natural \sqsubseteq S$ in which the condition holds and whose abstraction contains x.

5.4 Assignments to non-summary objects

Let $S \in \mathbb{V}^M$ be an abstract state and let $v_i \leftarrow e(w_1, ..., w_k)$ be an assignment, such that $F_{sum}(v_i)$ is a non-summary abstract object. We want to construct the most precise abstract state $S' \in \mathbb{V}^M$, such that for any concrete state $S^\natural \sqsubseteq S$,

$$[\![v_i \leftarrow e(w_1, ..., w_k)]\!]_\natural(S^\natural) \sqsubseteq S'$$

Let $Obj^{S'} = Obj^S$ and $dim^{S'} \equiv dim^S$. Also let $m = dim^S(F_{sum}(v_i))$. We define the abstract transformer as follows:

$$\begin{aligned} S' &= [\![v_i \leftarrow e(w_1, ..., w_k)]\!](S) \\ &= \left\{y : \exists x \in S \text{ s.t. } y[m] \in Values_e^S(x) \text{ and } y[j] = x[j] \text{ for } j \neq m\right\} \\ &= [\![\textit{drop}_e]\!]_{std} \circ [\![d_m \leftarrow e(\bar{d})]\!]_{std} \circ [\![\textit{expand}_e]\!]_{std}(S) \end{aligned}$$

Let us show that this transformation is sound. Let $S^\natural \sqsubseteq^{F_{sum}} S$ be an arbitrary concrete numeric state, such that $[\![e(w_1, ..., w_k)]\!]_\natural(S^\natural) = \alpha$. We denote the concrete state, to which $S^\natural$ is transformed as the result of the assignment, by $\hat{S}^\natural$, where:

$$\hat{S}^\natural = [\![v_i \leftarrow e(w_1, ..., w_k)]\!]_\natural(S^\natural)$$

Both $S^\natural$ and $\hat{S}^\natural$ are points in N-dimensional space. By the definition of the concrete semantics, $S^\natural$ and $\hat{S}^\natural$ are equal component-wise, except for component v_i which is equal to α in $\hat{S}^\natural$. Let us pick an arbitrary projection $\pi \in \Pi_{F_{sum}}$. Let $x = \pi(S^\natural)$ and $\hat{x} = \pi(\hat{S}^\natural)$. Since $F_{sum}(v_i)$ is a non-summary abstract numeric object, the m-th component of $\hat{x}$ is equal to α, whereas other components are equal to corresponding components of x. Now, since $x \in S$ and $\alpha \in Values_e^S(x)$, it follows by construction that $\hat{x} \in S'$. Therefore, $\hat{S}^\natural \sqsubseteq^{F_{sum}} S'$. The equality on the third line of the definition is justified by the discussion of how to compute set $Values_e^S(x)$ in Sect. 5.2.

Also, the transformation is minimal in the sense that for every point $x' \in S'$, there exists a concrete state $S^\natural \sqsubseteq S$, such that x' is in the abstraction of a concrete state $\hat{S}^\natural$, where

$$\hat{S}^\natural = [\![v_i \leftarrow e(w_1, ..., w_k)]\!]_\natural(S^\natural)$$

By construction, the point x' is in S' if there exists a point $x \in S$, which is equal to x' component-wise with the exception of component $x'[m]$, whose value is in $Values_e^S(x)$. By definition of the set $Values_e^S(x)$, there exists a concrete state $S^\natural \sqsubseteq S$, such that its abstraction contains point x and $[\![e(w_1, ..., w_k)]\!]_\natural(S^\natural) = x'[m]$. Then, from the concrete semantics and the abstraction mechanism, it follows that x' is in the abstraction of concrete state $\hat{S}^\natural$.

5.5 Assignments to summary objects

Let $S \in \mathbb{V}^M$ be an abstract state and let $v_i \leftarrow e(w_1, ..., w_k)$ be an assignment, such that $F_{sum}(v_i)$ is a summary abstract object. We want to construct the most precise abstract state $S' \in \mathbb{V}^M$, such that for any concrete state $S^\natural \sqsubseteq S$,

$$[\![v_i \leftarrow e(w_1, ..., w_k)]\!]_\natural(S^\natural) \sqsubseteq S'$$

Let $Obj^{S'} = Obj^S$ and $dim^{S'} \equiv dim^S$. Also let $m = dim^S(F_{sum}(v_i))$. We define the abstract transformer as follows:

$$\begin{aligned} S' &= [\![v_i \leftarrow e(w_1, ..., w_k)]\!](S) \\ &= \left\{ y : \exists x \in S \text{ s.t. } y[m] \in Values_e^S(x) \cup \{x[m]\} \text{ and } y[j] = x[j] \text{ for } j \neq m \right\} \\ &\supseteq [\![\mathit{drop}_e]\!]_{std} \circ [\![\mathit{fold}_{d_m,d_n}]\!]_{std} \circ [\![d_n \leftarrow e(\bar{d})]\!]_{std} \circ [\![add_{d_n}]\!]_{std} \circ [\![\mathit{expand}_e]\!]_{std}(S) \end{aligned}$$

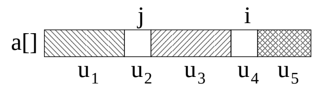
The soundness and minimality arguments are the same as in the Sect. 5.4. The only difference is that the abstract object $F_{sum}(v_i)$ corresponds to a collection of concrete numeric objects, only one of which is updated. Hence, for each point $x \in S$, the component $x[m]$ may preserve its old value. Note, that $\supseteq$ in the third line of the equation is due to the implementation $[\![\mathit{fold}]\!]_{std}$, which computes an overapproximation of *fold* in most numeric domains.

6 Related Work

The introduction already mentioned several numeric domains that have been investigated, including *non-relational domains*, such as intervals [7, 15] and congruences [5];

```
(1) for(i = 0; i < n; i++) {
(2)    for(j = 0; j < i; j++)
(3)       if(A[j] > A[i]) break;
(4)    t = A[i];
(5)    for(k = i; k > j; k--)
(6)       A[k] = A[k-1];
(7)    A[j] = t
(8) }
```

(a)



(b)

$$u_1 \leq u_4$$
$$u_4 < u_3$$
$$u_4 < u_2$$

(c)

Fig. 3. Insertion sort: (a) code for insertion sort; (b) array partitioning: abstract objects u_i represent array segments; (c) invariants captured by the abstract state at line 4.

weakly relational domains, such as difference constraints [4] and octagons [11]; and *relational domains*, such as polyhedra [2, 6] and trapezoidal congruences [10]. In all of this work, the assumption is made that there are a fixed number of numeric objects to track, where the number is known in advance. In contrast, our work provides techniques for performing static analysis in the presence of an unbounded number of concrete numeric objects (which are then collapsed into some number of summary objects).

Yavuz-Kahveci and Bultan present an algorithm for shape analysis in which numeric information is attached to summary nodes; the information on a summary node u of a shape-graph S bounds the number of concrete nodes that are mapped to u from any concrete memory configuration that S represents [17]. This represents a different approach to combining a collection of numeric quantities from the one pursued in our work: in [17], each combined object contributes 1 to a *sum* that labels the summary object; in our approach, when objects are combined together, the effect is to create a *set* of values.

7 Future Work

In this section, we sketch how the techniques developed in the paper could be applied in a situation in which multiple, dynamically changing segments of an array are summarized separately.

The goal would be to use static analysis to prove the correctness of an array-sorting program, such as the insertion-sort program shown in Fig. 3(a). The idea would be to use summary numeric objects to represent segments of the array. Fig. 3(b) depicts a situation that occurs during the execution of insertion sort. This would be represented using three summary numeric objects: u_1, u_3, and u_5. As the sort progresses, u_1, u_3, and u_5 must be associated with different segments of the array. In other words, the dimensionalities of the spaces that they represent have to vary.

To actually carry this out in a static-analysis algorithm, the main difficulty would be to find which dimensions/segments to merge, and when to adjust the pattern of merging during the course of the analysis. Fortunately, this is addressed by canonical abstraction [14]. The verification of sorting algorithms by means of shape analysis (using the TVLA system [9]) was described in [8]. However, TVLA does not provide a facility to describe numeric values directly; thus, in [8] it was necessary to introduce a "work-around"—an artificial binary predicate *dle* (for "data less-than or equal"), to record whether the value of one element is less than or equal to the value of another element.

The work presented in this paper would allow such problems to be addressed in a more straightforward manner: the numeric manipulations would be handled by the operations described in Sect. 5, and TVLA's normal mechanisms for summarizing elements, based on canonical abstraction, and unsummarizing elements (TVLA's "focus" operation) would drive how the summarization partition on array elements would change during the course of the analysis.

Some additional issues arise in supporting the operations of *new* and *delete*, which introduce and remove numeric objects, respectively. However, the mechanisms that have been presented provide all the machinery that is required. For instance, the abstract semantics for the *new* operation is to *add* a new dimension to an abstract state initialized to a range of possible initial values. If the client analysis indicates that the added object is to be summarized by one of the existing abstract objects, the new dimension is then *folded* into the corresponding existing dimension. *Deleting* a non-summary numeric object *drops* the corresponding dimension from the abstract state.

Space limitation precludes giving a full treatment of the remaining issues involved in using summarizing abstract numeric domains in conjunction with canonical abstraction.

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. P. Cousot and N. Halbwachs. Automatic discovery of linear constraints among variables of a program. In *Symp. on Princ. of Prog. Lang.*, 1978.
3. G. B. Dantzig and B. C. Eaves. Fourier-motzkin elimination and its dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.
4. D.L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, pages 197–212, 1989.
5. P. Granger. *Analyses Semantiques de Congruence*. PhD thesis, Ecole Polytechnique, 1991.
6. 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.
7. W.H. Harrison. Compiler analysis of the value ranges for variables. *Trans. on Softw. Eng.*, 3(3):243–250, 1977.
8. T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *Int. Symp. on Software Testing and Analysis*, pages 26–38, 2000.
9. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symp.*, pages 280–301, 2000.
10. F. Masdupuy. *Array Indices Relational Semantic Analysis using Rational Cosets and Trapezoids*. PhD thesis, Ecole Polytechnique, 1993.
11. A. Mine. The octagon abstract domain. In *Proc. Eighth Working Conf. on Rev. Eng.*, pages 310–322, 2001.
12. A. Mine. A few graph-based relational numerical abstract domains. In *Static Analysis Symp.*, pages 117–132, 2002.
13. Y. Saad. Sparsekit: A basic tool kit for sparse matrix computations, version 2. Tech. rep., Comp. Sci. Dept. Univ. of Minnesota, June 1994.
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. C. Verbrugge, P. Co, and L.J. Hendren. Generalized constant propagation: A study in C. In *Int. Conf. on Comp. Construct.*, volume 1060 of *Lec. Notes in Comp. Sci.*, pages 74–90, 1996.
16. D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Symp. on Network and Distributed Systems Security (NDSS)*, February 2000.
17. T. Yavuz-Kahveci and T. Bultan. Automated verification of concurrent linked lists with counters. In *Static Analysis Symp.*, pages 69–84, 2002.