# Non-disjunctive Numerical Domain for Array Predicate Abstraction

Xavier Allamigeon[1,2]

[1] EADS Innovation Works, SE/CS – Suresnes, France
[2] CEA, LIST MeASI – Gif-sur-Yvette, France
firstname.lastname@eads.net

**Abstract.** We present a numerical abstract domain to infer invariants on (a possibly unbounded number of) consecutive array elements using array predicates. It is able to represent and compute affine equality relations over the predicate parameters and the program variables, without using disjunctions or heuristics. It is the cornerstone of a sound static analysis of one- and two-dimensional array manipulation algorithms. The implementation shows very good performance on representative benchmarks. Our approach is sufficiently robust to handle programs traversing arrays and matrices in various ways.

## 1 Introduction

Program analysis now involves a large variety of methods able to infer complex program invariants, by using specific computer-representable structures, such as intervals [1], octagons [2], linear (more exactly affine) equality constraints [3], or affine inequality constraints [4]. Each abstract domain induces an equivalence relation: two abstract elements are equivalent if and only if they represent the same concrete elements. In this context, an *equivalence class* corresponds to a set of equivalent abstract elements, called *representatives*. Although all representatives are equivalent, they may not be identically treated by abstract operators or transfer functions, which implies that the choice of a "bad" representative may cause a loss of precision. Most numerical domains (for instance, reduced product [5]) are provided with a reduction operator which associates each abstract element to a "good" equivalent element, which will allow gaining precision.

Unfortunately, in some abstract domains, it may not be possible to define a precise reduction operator, because for some equivalence classes, the notion of "good" representatives may depend on further analysis steps, or on parts of the program not yet analyzed. This difficulty appears in abstract domains based on universally quantified predicates ranging over (a possibly unbounded number of) consecutive array elements (first introduced in [6]). The abstract elements of these domains consist of a predicate $\mathbf{p}$ and two parameters $u$ and $v$: $\mathbf{p}(u, v)$ means that all the elements whose index is between $u$ and $v$ (both included) contain values for which the statement $\mathbf{p}$ holds. These predicates are then combined with classic numerical abstractions to bind their parameters to the values of the program variables.

```
1:  int i, n, p; bool t[n];
2:  assert 0 <= p <= n;
3:  i := 0;
4:  while i < n do
5:    t[i] := 0;
6:    i := i+1;
7:  done;
8:  while i > p do
9:    t[i-1] := 1;
10:   i := i-1;
11: done;
12:
```

**Fig. 1.** Incrementing then decrementing array manipulations

```
int i, n; bool t[n];
i := 0;
while i < n do
  t[i] := 0;
  i := i+1;
done;
while ... do
  if ... then
    write_one();
  else
    write_zero();
  end;
done;
```

```
write_one() {
  if i > 0 then
    t[i-1] := 1;
    i := i-1;
  end;
}

write_zero() {
  if i < n then
    t[i] := 0;
    i := i+1;
  end;
}
```

**Fig. 2.** Both incrementing and decrementing array manipulations. The notation ... stands for a non-deterministic condition.

*Overview of the Problem.* As an example, let us try to analyze the first loop of the program given in Fig. 1, which initializes the array `t` with the boolean 0. For that purpose, we introduce the predicate **zero** (which means that the associated array elements contain the value 0), combined with the affine inequality domain. Informally, the loop invariant consists in joining the abstract representations $\Sigma_k$ of the concrete memory states arising after exactly $k$ loop iterations. For example, after one loop iteration ($k = 1$), the instruction `t[i] := 0` has assigned a zero to the array element of index 0, so that $\mathbf{zero}(u, v)$, with $u = v = 0$, $i = 1$ and $n \geq 1$. Similarly, after ten loop iterations, the ten first array elements have been initialized, thus $\mathbf{zero}(u, v)$, with $u = 0$, $v = 9$, $i = 10$ and $n \geq 10$. It can be shown that joining all the abstract states $\Sigma_k$ with $k \geq 1$, *ie* which have entered the loop at least once, yields the invariant $\mathbf{zero}(u, v)$, with $u = 0$, $v = i - 1$, and $1 \leq i \leq n$. We now have to join this invariant with $\Sigma_0$ to obtain the whole loop invariant. The abstract state $\Sigma_0$ represents the concrete memory states which have not entered the loop. Since the array $t$ is not initialized, $\Sigma_0$ is necessarily represented by a *degenerate* predicate, *ie* a predicate $\mathbf{zero}(l, m)$ such that $l > m$, which ranges over an empty set of array elements. Degenerate predicates naturally form an equivalence class, containing an infinite number of representatives, while non-degenerate predicates form classes containing a unique representative. Now, choosing the degenerate predicate $\mathbf{zero}(u, v)$ with $u = 0$, $v = -1$, $i = 0$, and $n \geq 0$, to represent $\Sigma_0$, yields the expected loop invariant $u = 0$, $v = i$, and $0 \leq i \leq n$. On the contrary, if we choose $\mathbf{zero}(u, v)$ with $u = 10$, $v = 9$, $i = 0$, and $n \geq 0$, we obtain an invariant $\mathbf{zero}(u, v)$ with much less precise affine inequality relations, in which, in particular, the value of $u$ is not known exactly anymore (it ranges between 0 and 10). Therefore, the representative $\mathbf{zero}(0, -1)$ is a judicious choice in the first loop analysis. But choosing the same representative for the second loop analysis will lead to a major loss of precision. The second loop partly initializes the array with the boolean 1 between from the index $n-1$ to the index $p$. Using a predicate **one** to

represent array elements containing the value 1, the analysis yields the expected invariant only if the representative $\mathbf{one}(t, s)$ with $t = n$ and $s = n - 1$ is chosen to represent the class of degenerate predicates **one**.

This example illustrates that the choice of right representatives for the degenerate classes to avoid loss of precision, is not an obvious operation, even for simple one-dimensional array manipulations. In [6, 7], some solutions are proposed to overcome the problem: (i) use heuristics to introduce the right degenerate predicates. This solution is clearly well-suited for the analysis of programs involving very few different natures of loops, such as incrementing loops always starting from the index 0 of the arrays, but is not adapted for more complex array manipulations. In particular, we will see in Sect. 4 that even classic matrix manipulation algorithms involve various different configurations for degenerate predicates. (ii) partition degenerate and non-degenerate predicates, instead of merging them in a single (and convex) representation. However such a disjunction may lead to an algorithmic explosion, since at least one disjunction has to be preserved for each predicate, including at control points located after loops: for example, the expected invariant at the control point 12 in Fig. 1 is $\mathbf{zero}(u, v) \wedge \mathbf{one}(s, t)$ with $u = 0$, $v = p - 1$, $s = p$, and $t = n - 1$. Without further information on $n$ and $p$, this invariant contains non-degenerate and degenerate configurations of both predicates $\mathbf{zero}(u, v)$ and $\mathbf{one}(s, t)$. Partitioning these configurations yields the disjunction $(n = p = 0) \vee (n > p = 0) \vee (p = n > 0) \vee (0 < p < n)$. And, if the program contains instructions after control point 12, the disjunction must be propagated through the rest of the program analysis. Therefore, this approach may not scale up to programs manipulating many arrays.[3] (iii) partition traces [8], for instance unroll loops, in order to distinguish traces in which non-degenerate predicates are inferred, from others. This solution is adapted to simple loops: as an example, for the loop located at control point 4 in Fig. 1, degenerate predicates occur only in the trace which does not enter the loop. But, in general, it may be difficult to automatically discover well-suited trace partitions: for example, in Fig. 2, traces in which the functions `write_one` and `write_zero` are called the same number of times, or equivalently, $i = n$, should be distinguished from others, since they contain a degenerate form of the predicate **one**. Besides, if traces are not ultimately merged, trace partitioning may lead to an algorithmic explosion for the same reasons as state partitions, while merging traces amounts to the problem of merging non-degenerate and degenerate predicates in a non-disjunctive way.

As we aim at building an efficient and automatic static analysis, we do not consider any existing solution as fully satisfactory.

*Contributions.* We present a numerical abstract domain to be combined with array predicates. It represents sets of equivalence classes of predicates, by inferring affine invariants on some representatives of each class. In particular, the right

[3] However, some techniques could allow merging disjunctions in certain cases. We will see at the end of Sect. 3 that these techniques coincide with the join operation that we develop in this paper.

representatives are automatically discovered, without any heuristics. As it is built as an extension of the affine equality constraint domain [3, 9], it does not use any disjunctive representations. Several abstract transfer functions are defined, all are proven to be sound. This domain allows the construction and the implementation of a sound static analysis of array manipulations. It is adapted to array predicates ranging over the elements of one-dimensional or two-dimensional arrays. Our work does not focus on handling a very large and expressive family of predicates relative to the content of the array itself, but rather on the complexity due to the automatic discovery of affine relations among program variables and predicate parameters, hence of right representatives for degenerate predicates. Therefore, the analysis has been experimented on programs traversing arrays and matrices in various ways. In all cases, the most precise invariants are discovered, which proves the robustness of our approach.

Section 2 presents the principles of the representation of equivalence classes of array predicates. Section 3 introduces the domain of *formal affine spaces* to abstract sets of equivalence classes of array predicates by affine invariants on some of their representatives. In Sect. 4, the construction of the array analysis and experiments are discussed. Finally, related work is presented in Sect. 5.

## 2 Principles of the Representation

As explained in Sect. 1, array predicates are related by an equivalence relation, depending on their nature (degenerate or non-degenerate): for an one-dimensional array predicate $\mathbf{p}$, two representations $\mathbf{p}(u, v)$ and $\mathbf{p}(u', v')$ are equivalent if and only if both are degenerate, *ie* $u > v \wedge u' > v'$, or they are equal ($u = u' \wedge v = v'$). More generally, given predicates with $\mathsf{p}$ parameters, we assume that there exists an equivalence relation $\sim$ over $\mathbb{R}^{\mathsf{p}}$, defining the equivalence of two numerical $\mathsf{p}$-tuples of predicate parameters.

Given a program with $\mathsf{n}$ scalar variables, a memory state can be represented by an element of $\mathbb{R}^{\mathsf{n}+\mathsf{p}}$, where each scalar variable is associated to one of the $\mathsf{n}$ first dimensions, and array predicate parameters are mapped to the $\mathsf{p}$ last ones. Then, the equivalence relation $\sim$ can be extended to $\mathbb{R}^{\mathsf{n}+\mathsf{p}}$ to characterize memory states which are provided with equivalent predicates: two memory states $M, N$ in $\mathbb{R}^{\mathsf{n}+\mathsf{p}}$ are equivalent, which is denoted by $M \simeq N$, if and only if $M$ and $N$ coincide on their $\mathsf{n}$ first dimensions, and if the $\mathsf{p}$-tuples formed by the $\mathsf{p}$ last dimensions are equivalent w.r.t. $\sim$. We adopt the notation $[M]$ to represent the equivalence class of $M$, *ie* the set of elements equivalent to $M$.

We have seen in Sect. 1 that the representation of equivalence classes by arbitrarily-chosen representative elements may lead to a very complex invariant, possibly not precisely representable in classic numerical domains. Our solution consists in representing an equivalence class by a *formal representative* instead: it consists in a $(\mathsf{n}+\mathsf{p})$-tuple, whose $\mathsf{n}$ first coordinates contain values in $\mathbb{R}$, while the $\mathsf{p}$ last ones (related to predicate parameters) contain *formal variables*, taken in a given set $\mathcal{X}$. A formal representative $R$ is provided with a set of valuations over $\mathcal{X}$: each valuation $\nu$ maps $R$ to a point $R\nu$ of $\mathbb{R}^{\mathsf{n}+\mathsf{p}}$, by replacing each

formal variable $x$ in $R$ by the value $\nu(x) \in \mathbb{R}$. Then, an equivalence class $C$ can be represented by a formal representative $R$ and a set of valuations $V$ such that for any $\nu \in V$, the element $R\nu$ is in the class $C$. In other words, a formal representative can represent several elements of a same equivalence class.

Let us illustrate the principle of formal representative with the program in Fig. 1, with $\mathsf{n} = 3$ scalar variables $i$, $n$, and $p$. Consider the equivalence class of a memory state at control point 4 which has not yet entered the loop, thus in which the predicate $\mathbf{zero}(u, v)$ is degenerate, and in which, for instance, $i = 0$, $n = 10$, and $p = 5$. It can be represented by the formal representative $R = (0, 10, 5, x, y)$ (written as a row vector for reason of space) and the set of valuations $V = \{\nu \mid \nu(x) > \nu(y)\}$: indeed, each representative $R\nu$ corresponds to a predicate $\mathbf{zero}(u, v)$ such that $u > v$. In that case, all the equivalent numerical configurations for the degenerate predicate $\mathbf{zero}(u, v)$ are represented in the formal representative.

Therefore, formal representatives allow keeping several representatives for a given class $C$ instead on focusing on only one of them. In the following sections, we define *formal affine spaces*, which extend the affine equality domain to range over formal representatives. These formal affine spaces are combined with sets of valuations represented by affine inequality constraints over $\mathcal{X}$, giving the right values for the representatives. Besides, we describe how to compute the formal affine spaces, so as to automatically discover affine invariants on some representatives of distinct equivalence classes.

## 3 Formal Affine Spaces

We now formally introduce the abstract domain to represent sets of equivalence classes of array predicates. We follow the abstract interpretation methodology [1], by defining a concretization operator, and then abstract operators such as union.

Let $\Delta$ be the set of equivalence classes w.r.t the equivalence relation $\simeq$, and $\Delta(\mathcal{X})$ be the set of formal representatives. Formally, $\Delta(\mathcal{X})$ is isomorphic to the cartesian product of $\mathbb{R}^{\mathsf{n}}$, representing the set of memory states over scalar variables, with $\mathcal{X}^{\mathsf{p}}$. Given a formal representative $M$, $\pi_1(M)$ represent the $\mathsf{n}$-tuple consisting in the $\mathsf{n}$ first coordinates. This element of $\mathbb{R}^{\mathsf{n}}$ is called the *real component* of $M$. Besides, the $\mathsf{p}$ last coordinates of $M$ forms $\pi_2(M)$, called *formal component* of $M$. Similarly, the $\mathsf{i}$th coordinate of $M$ is said to be *real* (respectively *formal*) if $\mathsf{i} \leq \mathsf{n}$ (resp. $\mathsf{i} > \mathsf{n}$).

While the affine equality domain was initially introduced using conjunctions of equality constraints [3], affine spaces can be represented by means of generators as well [9]. An *affine generator system* $E + \Omega$ is given by a family $E = (e_i)_{1 \leq i \leq s}$ of linearly independent vectors of $\mathbb{R}^{\mathsf{n}}$, and a point $\Omega \in \mathbb{R}^{\mathsf{n}}$. It is associated to the affine space defined by:

$$\mathsf{Span}(E + \Omega) = \left\{ \Omega + \sum_{i=1}^{s} \lambda_i e_i \mid \lambda_1, \ldots, \lambda_s \in \mathbb{R} \right\} , \tag{1}$$

$$\begin{matrix}\pi_1\\ \pi_2\end{matrix}\left\{\left[\begin{pmatrix}0\\1\\0\\x_1\\x_2\end{pmatrix},\begin{pmatrix}0\\0\\1\\y_1\\y_2\end{pmatrix}\right]+\begin{pmatrix}0\\0\\0\\z_1\\z_2\end{pmatrix}\right. \mathrel{!} V \qquad \begin{matrix}\pi_1\\ \pi_2\end{matrix}\left\{\left[\begin{pmatrix}0\\1\\0\\x'_1\\x'_2\end{pmatrix},\begin{pmatrix}0\\0\\1\\y'_1\\y'_2\end{pmatrix}\right]+\begin{pmatrix}1\\0\\0\\z'_1\\z'_2\end{pmatrix}\right. \mathrel{!} V'$$

where $V = \{x_1 = x_2 \wedge y_1 = y_2 \wedge z_1 > z_2\}$

where $V' = \{x'_1 = x'_2 = 0 \wedge y'_1 = y'_2 = 0 \wedge z'_1 = z'_2 = 0\}$

**Fig. 3.** Two formal affine spaces for $\mathsf{n} = 3$ and $\mathsf{p} = 2$

corresponding to the set of the points generated by the addition of linear combinations of the vectors $e_i$ to the point $\Omega$. Affine generator systems are equivalent to sets of affine constraints. Indeed, the elimination of the $\lambda_i$ in the combinations given in Eq. (1) yields an equivalent set of affine constraints over the coordinates of the points.

Formal affine spaces are defined by extending affine generator systems of $\mathbb{R}^{\mathsf{n}}$ with $\mathsf{p}$ formal coordinates: generators are now elements of $\Delta(\mathcal{X})$, provided with a set of valuations.

**Definition 1.** *A* formal affine space $E + \Omega \mathrel{!} V$ *is given by a family* $E = (e_1, \ldots, e_s)$ *of vectors of* $\Delta(\mathcal{X})$*, a point* $\Omega$ *of* $\Delta(\mathcal{X})$ *verifying:*

- *the* $(\pi_1(e_i))_{1 \leq i \leq s}$ *are linearly independent,*
- *any two formal variables occurring in* $(\pi_2(e_i))_i$ *and* $\pi_2(\Omega)$ *are distinct,*

*and an affine inequality constraint system* $V$ *over the formal variables occurring in* $(\pi_2(e_i))_i$ *and* $\pi_2(\Omega)$*.*

Figure 3 gives an example of formal affine spaces. We abusively denote by $\nu \in V$ the fact that the valuation $\nu$ satisfies the constraint system $V$. Similarly to "classic" affine generator systems, a formal affine space $E + \Omega \mathrel{!} V$ generates a set of formal representatives, written as combinations $\Omega + \sum_i \lambda_i e_i$. As explained in Sect. 2, each formal representative $R$, provided with the set of valuations satisfying $V$, represents a set of several representatives which belong to a same equivalence class $C$: for any $\nu \in V$, $C = [R\nu]$. Following these principles, the concretization operator $\gamma$ maps any formal space $E + \Omega \mathrel{!} V$ to the set of the equivalence classes represented by the generated formal representatives:

$$\gamma(E + \Omega \mathrel{!} V) \stackrel{def}{=} \{C \mid R \in \mathsf{Span}(E + \Omega) \wedge \forall \nu \in V.\ C = [R\nu]\} \quad , \qquad (2)$$

where $\mathsf{Span}(E + \Omega)$ consists of the combinations $\Omega + \sum_{i=1}^{s} \lambda_i e_i$, for $\lambda_i \in \mathbb{R}$.

*Example 1.* Consider the formal affine space $E + \Omega \mathrel{!} V$ on the left-hand side of Fig. 3. Any combination in $\mathsf{Span}(E + \Omega)$ is a formal representative $R$ of the form $(0, \lambda, \mu, \lambda x_1 + \mu y_1 + z_1, \lambda x_2 + \mu y_2 + z_2)$ (written as a row vector for reason of space) where $\lambda, \mu \in \mathbb{R}$. Suppose that the dimensions respectively represent the

scalar variables $i$, $n$, $p$, and the parameters $u$ and $v$ of a predicate $\mathbf{zero}(u, v)$. Then $R$ represents the equivalence classes of memory states in which $i = 0$, $n$ and $p$ have independent values, and for any valuation $\nu \in V$,

$$u = \lambda\nu(x_1) + \mu\nu(y_1) + \nu(z_1) > \lambda\nu(x_2) + \mu\nu(y_2) + \nu(z_2) = v \ , \qquad (3)$$

or equivalently, the predicate $\mathbf{zero}(u, v)$ is degenerate. In particular, $E + \Omega \mathrel{!} V$ allows abstracting the memory states at control point 4 in Fig. 1 which have not yet entered the loop. Besides, it represents several representatives for the degenerate predicate $\mathbf{zero}(u, v)$, while a "classic" affine invariant would select only one of them. Similarly, the formal affine space $F + \Omega' \mathrel{!} V'$ on the right-hand side of Fig. 3 yields formal representatives $R'$ corresponding to classes of memory states such that $i = 1$, $n$ and $p$ are arbitrary, and $u = v = 0$, since for $i \in \{1, 2\}$, $\lambda\nu'(x'_i) + \mu\nu'(y'_i) + \nu'(z'_i) = 0$ for any valuation $\nu' \in V'$. Then, it is an abstraction of the memory states after the first iteration of the first body loop in Fig. 1: the first element of the array $t$ (index 0) contains the value 0. □

### 3.1 Joining Two Formal Spaces

We wish to define a union operator $\sqcup$ which provides an over-approximation of two formal affine spaces $E + \Omega \mathrel{!} V$ and $F + \Omega' \mathrel{!} V'$. Let us illustrate the intuition behind the definition of $\sqcup$ by sufficient conditions.

Suppose that $G + O \mathrel{!} W$ is the resulting formal space. A good start is to require $\sqcup$ to be sound w.r.t. the underlying real affine generator systems: if $\pi_1(G + O)$ denotes the real affine generator system obtained by applying $\pi_1$ on each vector $g_i$ of $G$ and on the origin, then $\pi_1(G + O)$ has to represent a larger affine space than those generated by $\pi_1(E + \Omega)$ and $\pi_1(F + \Omega')$. To ensure this condition, let us build $G + O \mathrel{!} W$ by extending the *sum system* of the two real systems $\pi_1(E + \Omega)$ and $\pi_1(F + \Omega')$.[4] More precisely, if $G_r + O_r$ denotes the sum system, we add $\mathsf{p}$ fresh formal variables to each vector of $G_r$ and to $O_r$, which yields $G + O$.

Then, to ensure $\gamma(E + \Omega \mathrel{!} V) \subseteq \gamma(G + O \mathrel{!} W)$, we require $\mathsf{Span}(E + \Omega)$ to be "included" in $\mathsf{Span}(G + O)$. Although the inclusion already holds for their real components ($\mathsf{Span}(\pi_1(E+\Omega)) \subseteq \mathsf{Span}(\pi_1(G+O))$), $\mathsf{Span}(E+\Omega)$ and $\mathsf{Span}(G+O)$ can not be directly compared since they may contain different formal variables. Therefore, we build a substitution $\sigma_P$ over the formal variables occurring in $\pi_2(E + \Omega)$, such that for any $R \in \mathsf{Span}(E + \Omega)$, we have $R\sigma_P \in \mathsf{Span}(G + O)$. This substitution is induced by the *change-of-basis matrix* $P$ from $\pi_1(E + \Omega)$ to $\pi_1(G+O)$, which verifies $\mathsf{mat}(\pi_1(E+\Omega)) = \mathsf{mat}(\pi_1(G+O)) \times P$ ($\mathsf{mat}(\pi_1(E+\Omega))$ is the matrix whose columns are formed by the vectors $(\pi_1(e_i))_i$ and $\pi_1(\Omega)$). The matrix $P$ expresses the coefficients of the (unique) decomposition of each $\pi_1(e_i)$ and $\pi_1(\Omega)$ in terms of the $\pi_1(O)$ and $(\pi_1(g_k))_k$. It allows to express the

[4] The *sum system* is obtained by extracting a free family $G_r$ from the vectors $(\pi_1(e_i))_i$, $(\pi_1(f_i))_j$, and $\pi_1(\Omega') - \pi_1(\Omega)$, and choosing $O_r = \pi_1(\Omega)$. Then, $G_r + O_r$ generates the smallest affine space greatest than the affine spaces represented by both $\pi_1(E + \Omega)$ and $\pi_1(F + \Omega')$.

$$P = \begin{pmatrix} 0\,0\,0 \\ 1\,0\,0 \\ 0\,1\,0 \\ 0\,0\,1 \end{pmatrix} \quad Q = \begin{pmatrix} 0\,0\,1 \\ 1\,0\,0 \\ 0\,1\,0 \\ 0\,0\,1 \end{pmatrix} \quad \forall i \in \{1,2\}. \begin{cases} \sigma_P(x_i) \mapsto \mathbf{y}_i \\ \sigma_P(y_i) \mapsto \mathbf{z}_i \\ \sigma_P(z_i) \mapsto \mathbf{t}_i \end{cases} \quad \begin{cases} \sigma_Q(x'_i) \mapsto \mathbf{y}_i \\ \sigma_Q(y'_i) \mapsto \mathbf{z}_i \\ \sigma_Q(z'_i) \mapsto \mathbf{t}_i + \mathbf{x}_i \end{cases}$$

**Fig. 4.** Change-of-basis matrices and their associated substitutions

$\pi_2(e_i)$ and $\pi_2(\Omega)$ in terms of the $\pi_2(O)$ and $(\pi_2(g_k))_k$ as well, by defining $\sigma_P$ by $\sigma_P(\mathsf{mat}(\pi_2(E+\Omega))) \stackrel{def}{=} \mathsf{mat}(\pi_2(G+O)) \times P$.

Now, it suffices that $W$ be a stronger system of constraints than $V\sigma_P$, the system obtained by applying the substitution $\sigma_P$ on $V$. Indeed, for any class $C \in \gamma(E+\Omega \restriction V)$, there exists $R \in \mathsf{Span}(E+\Omega)$ such that for any $\nu \in V$, $C = [R\nu]$. Then, for any $\nu' \in W$, we have $\nu' \in V\sigma_P$, so that there exists a valuation $\nu \in V$ such that $\forall x.(\sigma_P(x))\nu' = \nu(x)$. This implies $(R\sigma_P)\nu' = R\nu$, hence $C = [(R\sigma_P)\nu']$. A similar reasoning can be performed for $F+\Omega' \restriction V'$, which leads to the following definition of $\sqcup$:

**Definition 2.** *The union $(E+\Omega \restriction V) \sqcup (F+\Omega' \restriction V')$ is defined as the formal space $G+O \restriction W$ where $\pi_1(G+O)$ is the sum of $\pi_1(E+\Omega)$ and $\pi_1(F+\Omega')$, yielding two change-of-basis matrices $P$ and $Q$ respectively, and $W$ is the conjunction of the two systems of constraints $V\sigma_P$ and $V'\sigma_Q$.*

The following proposition states that the union operator is sound.

**Proposition 1.** *The union $(E+\Omega \restriction V) \sqcup (F+\Omega' \restriction V')$ over-approximates the union of the sets of classes represented by $E+\Omega \restriction V$ and $F+\Omega' \restriction V'$.*

*Example 2.* Consider the formal spaces $E+\Omega \restriction V$ and $F+\Omega' \restriction V'$ introduced in Ex. 1. The sum of the two real affine generator systems $\pi_1(E+\Omega)$ and $\pi_1(F+\Omega')$ is a system in which $i$, $n$, and $p$ are all independent, so that:

$$G+O \stackrel{def}{=} \begin{matrix} i \\ n \\ p \\ u \\ v \end{matrix} \begin{bmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \\ \mathbf{y}_1 \\ \mathbf{y}_2 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \\ \mathbf{z}_1 \\ \mathbf{z}_2 \end{pmatrix} \end{bmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ \mathbf{t}_1 \\ \mathbf{t}_2 \end{pmatrix} \quad . \tag{4}$$

The corresponding change-of-basis matrices $P$ and $Q$ are given in Fig. 4. In particular, these matrices represent the relation $\pi_1(\Omega') = \pi_1(O) + \pi_1(g_1)$, which generates the substitutions $z'_1 \mapsto \mathbf{t}_1 + \mathbf{x}_1$ and $z'_2 \mapsto \mathbf{t}_2 + \mathbf{x}_2$. The associated substitutions $\sigma_P$ and $\sigma_Q$ are then defined in Fig. 4. Applying them on the constraint systems $V$ and $V'$ yields: $V\sigma_P = \{\mathbf{y}_1 = \mathbf{y}_2 \wedge \mathbf{z}_1 = \mathbf{z}_2 \wedge \mathbf{t}_1 > \mathbf{t}_2\}$ and $V'\sigma_Q = \{\mathbf{y}_1 = \mathbf{y}_2 = 0 \wedge \mathbf{z}_1 = \mathbf{z}_2 = 0 \wedge \mathbf{t}_1 + \mathbf{x}_1 = \mathbf{t}_2 + \mathbf{x}_2 = 0\}$, so that:

$$W = \{\mathbf{x}_1 = -\mathbf{t}_1 \wedge \mathbf{x}_2 = -\mathbf{t}_2 \wedge \mathbf{y}_1 = \mathbf{y}_2 = 0 \wedge \mathbf{z}_1 = \mathbf{z}_2 = 0 \wedge \mathbf{t}_1 > \mathbf{t}_2\} \quad . \tag{5}$$

It can be intuitively verified that $G+O \restriction W$ contains the formal spaces $E+\Omega \restriction V$ and $F+\Omega' \restriction V'$:

– when $i = 0$, we have $u = \mathbf{t}_1 + \lambda\mathbf{y}_1 + \mu\mathbf{z}_1$ and $v = \mathbf{t}_2 + \lambda\mathbf{y}_2 + \mu\mathbf{z}_2$ for some $\lambda, \mu \in \mathbb{R}$, so that for any $\nu \in W$, $u\nu = \nu(\mathbf{t}_1) > \nu(\mathbf{t}_2) = v\nu$. Then the predicate $\mathbf{zero}(u, v)$ is degenerate.
– when $i = 1$, we have $u = \mathbf{t}_1 + \mathbf{x}_1 + \lambda\mathbf{y}_1 + \mu\mathbf{z}_1$ and $v = \mathbf{t}_2 + \mathbf{x}_2 + \lambda\mathbf{y}_2 + \mu\mathbf{z}_2$, hence $u\nu = v\nu = 0$ for any valuation $\nu \in W$. In that case, the predicate $\mathbf{zero}(u, v)$ ranges over the first element of the array.

The resulting formal space $G + O \mathbin{!} W$ is an over-approximation of the memory states arising at control point 4 in Fig. 1, after at most one loop iteration.

We could show that joining $E + \Omega \mathbin{!} V$ with the formal space resulting from the loop body execution on $G + O \mathbin{!} W$, yields the affine space $G + O \mathbin{!} W'$, where $W' = \{\mathbf{x}'_1 = 0 \wedge \mathbf{x}'_2 = 1 \wedge \mathbf{y}'_1 = \mathbf{y}'_2 = 0 \wedge \mathbf{z}'_1 = \mathbf{z}'_2 = 0 \wedge \mathbf{t}'_1 = 0 \wedge \mathbf{t}'_2 = -1\}$. It could be also verified that this affine space is a fixpoint of the loop transfer function. It represents the expected invariant $u = 0$ and $v = i - 1$. In particular, the computation automatically discovers the right representative $\mathbf{zero}(0, -1)$ (obtained with $i = 0$) among all the representatives $\mathbf{zero}(u, v)$ such that $u > v$ contained in $E + \Omega \mathbin{!} V$. □

Definition 2 and Ex. 2 raise some remarks. Firstly, when considering increasing formal affine spaces, the underlying real affine generators are logically growing, while the sets of valuations become smaller (the constraint system becomes stronger). Intuitively, this corresponds to an increasing determinism in the choice of the representatives in the equivalence classes abstracted by the formal space. In particular, when considering formal spaces obtained by iterating an increasing transfer function to compute a global invariant, two cases (among possibly more) are singular: when the set of valuations is reduced to a singleton, and when this set is empty. In the former, the formal affine space coincide with an affine generator system over $\mathbb{R}^{\mathsf{n}+\mathsf{p}}$: in other words, some representatives in the over-approximated equivalence classes are bound with program variables by an affine invariant. This situation happens at the end of Ex. 2, in which $u = 0$ and $v = i - 1$ in the affine space over-approximating the loop invariant. In the latter case, the discovery of an affine invariant failed: by definition of $\gamma$, the concretization of the formal space is the entire set $\Delta$.

Secondly, consider the two abstract memory states that we tried to join in Sect. 1 to compute an invariant of the first loop in Fig. 1: on the one hand, a degenerate predicate $\mathbf{zero}(u, v)$ with $i = 0$, and on the other hand, a non-degenerate one $\mathbf{zero}(u, v)$ with $u = 0$, $v = i - 1$, and $1 \leq i \leq n$. We could verify that joining the two representations by means of formal spaces, and in particular, computing the conjunction of the two corresponding constraint systems $V\sigma_P$ and $V'\sigma_Q$, exactly amounts to check whether the affine relations $u = 0$ and $v = i - 1$ match the degenerate condition $u > v$ when $i = 0$. More generally, when it succeeds, the approach based on matching degenerate condition coincides with the operations performed when joining two formal spaces. The major advantage of formal affine spaces is that it is adapted to any program or coding style, while matching degenerate conditions may fail. For example, let us consider the piece of program `i := n-1; if ... then t[i] := 0; i := i-1; fi;`. The matching approach would check if the non-degenerate invariant $\mathbf{zero}(u, v) \wedge u =$

$v = i + 1 = n - 1$ match the degenerate condition when $i = n - 1$, which is obviously false.

### 3.2 Precision and Further Abstract Operators

All usual abstract operators can be defined on formal affine spaces. For reason of space, we only give an enumeration. First, a partial order $\sqsubseteq$, defined in a similar way to the union operator, can be introduced. Then, the concretization $\gamma$ can be shown to be monotonic, and the union $\sqcup$ is the best possible join operator w.r.t. the order $\sqsubseteq$. Furthermore, the definition of guard, constraint satisfiability, and assignment operations closely follows the definition of the same primitives on real affine generator systems [3, 9], thus their design is simple. The main difference is that guards, satisfiability and assignments over predicate parameters involve operations on both the family of generators and the system of constraints representing the sets of valuations. For the latter, only usual operators, such as assignments or extracting a valuation satisfying the set of constraints, are necessary. All the operators on formal affine spaces are proven to be conservative. Moreover, exactness holds for guards, satisfiability, and invertible assignments, when they are applied to a formal affine space whose system of constraints representing the valuations is satisfiable.

## 4 Application to the Analysis of Array Manipulations

Formal affine spaces has been implemented to analyze array manipulation programs. The analysis computes abstract memory states consisting in a finite sequence of predicates, and a formal affine space over the program variables and the predicate parameters. Note that a reduced product of formal affine spaces with convex polyhedra [4] over scalar variables is used to increase precision, since affine generator systems do not precisely handle inequality guards.

Array assignments (*ie* assignments of the form `t[i] := e`) introduce new predicates in the abtract state (intuitively, non-degenerate predicates of the form $\mathbf{p}(u, v)$ with $u = v = i$). Then, some predicates may represent contiguous memory areas of a same array, and thus can be merged in a single predicate. The situations in which two predicates $\mathbf{p}$ and $\mathbf{q}$ can be merged correspond to simple geometric configurations. Two of these configurations for one- and two-dimensional are depicted respectively at the top and the bottom of Fig. 5. All these situations can be expressed as conjunctions of affine equality constraints over the parameters of the two predicates. When these constraints are satisfied, a new predicate $\mathbf{p} \curlyvee \mathbf{q}$ is introduced in the abstract state. The statement $\mathbf{p} \curlyvee \mathbf{q}$ itself over-approximates $\mathbf{p}$ and $\mathbf{q}$: it expresses a property on the values of the array element which is weaker than those expressed by $\mathbf{p}$ and $\mathbf{q}$. And its parameters are initialized to fit the whole area obtained by concatenating the memory areas corresponding to $\mathbf{p}$ and $\mathbf{q}$. Finally, the predicates $\mathbf{p}$ and $\mathbf{q}$ and their parameters are removed from the abstract state.

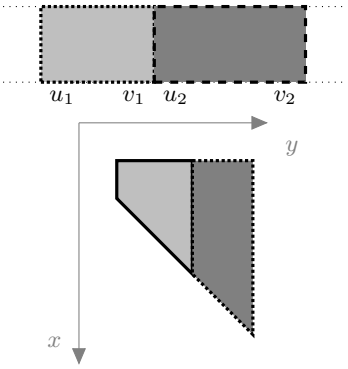


**Fig. 5.** Merging two contiguous predicates

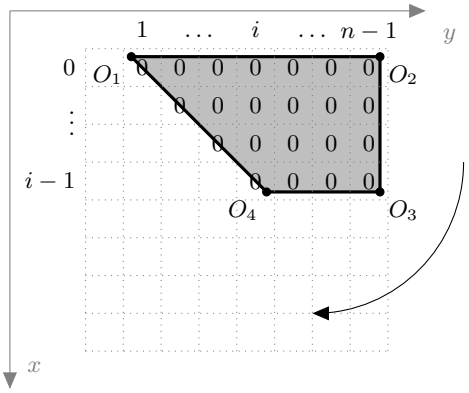


**Fig. 6.** Example of a two-dimensional predicate

*One-dimensional Predicates.* Two kinds of predicates are used to analyze array manipulations, depending on the type of arrays.

For arrays whose elements take their values in a finite set of cardinal $K$ (such as booleans or C enumerations), we consider one predicate $\mathbf{c}$ per possible value. Then $\mathbf{c}(u, v)$ states that the array contains the value $c$ between the indices $u$ and $v$. We allow at most $K$ pairwise distinct predicates $\mathbf{c}_1, \ldots, \mathbf{c}_K$ per array. The merging operations are applied only to predicates representing the same constant. Besides, if two predicates $\mathbf{c}(u_1, v_1)$ and $\mathbf{c}(u_2, v_2)$ ranging over the same array can not be merged, they are simply removed from the abstract state. Although this choice is very strict, it offers a tractable analysis, which is precise enough to handle the examples given in Figs. 1 and 2, as reported in Table 1.

For integer arrays, conjunctions of interval and bounded difference constraints (*ie* of the form $c_1 \leq x \leq c_2$ or $c_1 \leq x - y \leq c_2$) between the array content and the scalar variables are used. For instance, the predicate $\langle 0 \leq t \leq n-1 \rangle\,(u, v)$ represents the fact that the elements of the array $t$ located between the indices $u$ and $v$ all contain values between 0 and $n-1$ ($n$ being a program scalar variable). Such predicates are implemented under the form of $\mathsf{n}+1$ intervals: one to bound the array values in an interval, $\mathsf{n}$ to bound the differences with the $\mathsf{n}$ scalar variables. Then, the analysis allows at most one predicate per array. If a predicate associated to an array is introduced during the computation while this array already has a predicate, both are merged if possible, or simply removed if not. Moreover, to ensure termination, the statement $\mathbf{p} \curlyvee \mathbf{q}$ is obtained by pointwise widening the intervals contained in $\mathbf{p}$ and $\mathbf{q}$.

*Two-dimensional Predicates.* We use two-dimensional predicates which range over convex quadrilateral areas of two-dimensional arrays. Predicates are of the form $\mathbf{p}(O_1, O_2, O_3, O_4)$, and have now eight parameters, corresponding the $x$- and $y$-coordinates of the associated vertices $O_1$, $O_2$, $O_3$, and $O_4$. Degenerate

**Table 1.** Analysis benchmarks

| Programs | Invariants (by default, at the end of the program) | Time |
|---|---|---|
| Fig. 1 | $\mathbf{zero}(0, p-1) \wedge \mathbf{one}(p, n-1)$ | $\sim 0.6\,$s |
| Fig. 2 | outer loop invariant: $\mathbf{zero}(0, i-1) \wedge \mathbf{one}(i, n-1)$ | $\sim 0.7\,$s |
| `full_init` | i. and d. $\langle 0 \leq t \leq n-1 \rangle\,(0, n-1)$ | $< 0.2\,$s |
| `range_init` | i. and d. $\langle p \leq t \leq q-1 \rangle\,(p, q-1)$ | $< 0.2\,$s |
| `partial_init` | i. $\langle 0 \leq t \leq n-1 \rangle\,(0, j-1)$ and d. $\langle 0 \leq t \leq n-1 \rangle\,(j, n-1)$ | $\sim 0.2\,$s |
| `partition` | i. $\langle ge \geq 0 \rangle\,(0, gelen-1) \wedge \langle lt \leq -1 \rangle\,(0, ltlen-1)$ | $\sim 0.4\,$s |
| | d. $\langle ge \geq 0 \rangle\,(gelen, n-1) \wedge \langle lt \leq -1 \rangle\,(ltlen, n-1)$ | $\sim 0.5\,$s |
| `full_matrix` | r. $\langle m = 0 \rangle\,((0,0), (0, n-1), (n-1, n-1), (n-1, 0))$ | 12.9 s |
| | c.. $\langle m = 0 \rangle\,((n-1, 0), (0, 0), (0, n-1), (n-1, n-1))$ | 13.4 s |
| `lower_triang` | r. $\langle m = 0 \rangle\,((0,1), (0, n-1), (i-1, n-1), (i-1, i))$ | 12.6 s |
| (outer loop | c. $\langle m = 0 \rangle\,((0,1), (0,1), (0, j-1), (j-2, j-1))$ | 14.7 s |
| invariants) | dg. $\langle m = 0 \rangle\,((0,1), (0, k-1), (n-k, n-1), (n-2, n-1))$ | 11.3 s |
| `upper_triang` | r. $\langle m = 0 \rangle\,((1,0), (1,0), (i-1, i-2), (i-1, 0))$ | 14.6 s |
| (outer loop | c. $\langle m = 0 \rangle\,((n-1, 0), (1,0), (j, j-1), (n-1, j-1))$ | 13.1 s |
| invariants) | dg. $\langle m = 0 \rangle\,((n-1, 0), (n-k+1, 0), (n-1, k-2), (n-1, 0))$ | 15.0 s |

and non-degenerate predicates are distinguished by the rotation direction of the points $O_1$, $O_2$, $O_3$, and $O_4$. We use the convention that the interior of the polygon $O_1O_2O_3O_4$ is not empty if and only if $O_1$, $O_2$, $O_3$, and $O_4$ are ordered clockwise, as in Fig. 6. The shape of the polygons $O_1O_2O_3O_4$ is restricted by requirements, not fully detailed here, but implying in particular that the coordinates of the $O_i$ are integer, and the lines $(O_iO_{i+1})$ are either horizontal, vertical, or diagonal. These requirements are weak enough to express the invariants used in the targeted algorithms. Moreover, they allow characterizing degenerate polygons by a condition consisting of several affine inequalities over the predicate parameters.

The analysis allows for each matrix at most two predicates: one is one-dimensional, while the other is two-dimensional. Indeed, the matrix algorithms we wish to analyze performs intermediate manipulations on rows, columns, or diagonals. Thus, the former predicate is used to represent the invariant on the current one-dimensional structure, while the latter collects the information on the older structures, which form a two-dimensional shape. The predicates propagate bounded difference constraints relative to the matrix content.

*Benchmarks.* Table 1 reports the invariants discovered by our analyzer, implemented in Objective Caml (5000 lines of code), and the time taken for each analysis on a 1 Gb RAM laptop using one core of a 2 GHz Intel Pentium Core Duo processor. The first six programs involve only one-dimensional arrays. The two first programs are successfully analyzed using constant predicates, and the right array shape is discovered. The third one, `full_init`, initializes each element `t[i]` of the array `t` of size `n` with the value `i`. It results in a fully initialized array with values ranging between 0 and $n-1$. The program `range_init` has a similar behavior, except that it performs the initialization between the indices `p` and `q` only. The programs `partial_init` and `partition` are taken from [7] and [10] respectively. The former copies the value `i` in `t[j]` when the values of two other arrays `a[i]` and `b[i]` are equal, and then increments `j`. The latter partitions the positive or null and strictly negative values of a source array `a` into

the destination arrays `ge` and `lt` respectively. The three last programs involve matrices. The first one, `full_matrix`, fully initializes a matrix `m` of size $n \times n$. The two last ones only fill the upper- and lower-triangular part of the matrix with the value 0. Each program contains two nested loops. As an illustration, the invariant of the outer loop of the column-after-column version of `lower_triang` discovered by the analysis is given in Fig. 6. The reader can verify that the final invariant obtained for $i = n-1$ corresponds to a lower-triangular matrix. Several versions of each program are analyzed: for one-dimensional array manipulation algorithms, incrementing (i.) or decrementing (d.) loops (except for the programs in Figs. 1 and 2 which already use both versions of loops), and for matrix manipulation loops, row-after-row (r.), column-after-column (c.), or diagonal-after-diagonal (dg.) matrix traversal.[5] All the examples involving one-dimensional arrays only are successfully analyzed in less than a second. Analysis time does not exceed 15 s on programs manipulating matrices, which is a good result, considering the complexity of the merge conditions for two-dimensional predicates, and the fact that these programs contain nested loops. These benchmarks show that the analysis is sufficiently robust to discover the excepted invariant for several stategies of array or matrix manipulations programs. In particular, the right representatives for degenerate predicates are automatically found out in various and complex situations. As an example, the degenerate predicates discovered for the programs `lower_triang` (obtained with $i = 0$, $j = 1$, and $k = 1$) and `upper_triang` (obtained with $i = 1$, $j = 0$, and $k = 1$) all represent different configurations of interior-empty quadrilateral shapes. Furthermore, although not reported in Table 1, the analysis handles simple transformations (such as loop unrolling) on the experimented programs, without any loss of precision. Finally, for one-dimensional predicates, we have experimented, with formal affine spaces, the manual substitution of the general degenerate condition $u > v$ by the right degenerate configurations for each program. In that case, operations on formal affine spaces roughly coincide with operations in a usual equality constraint domain. We have found that the additional cost in time due to formal affine spaces is small (between 8% and 30%), which suggests that this numerical abstract domain has good performance, while it automatically discovers the right representatives.

## 5 Related Work

Several static analyses use predicates to represent memory shape properties: among others, [11–15] infer elaborate invariants on dynamic memory structures, such as lists and trees. Most of these works do not involve a precise treatment of arrays. Some abstract interpretation based analyzers [16–18] precisely handle manipulations of arrays whose size is exactly known. Besides, [17] can represent all the array elements by a single abstract element (*array smashing*). Albeit not very precise, it could also represent an unbounded number of array elements.

[5] The source code of each program is available at `http://www.lix.polytechnique.fr/Labo/Xavier.Allamigeon`.

To our knowledge, only [19, 6, 7, 20, 10, 21] handle precise properties ranging over an unbounded number of one-dimensional array elements. Most of them involve the predicates presented in this paper, and some other expressing more properties on the values of the array elements, such as equality, sorting or pointer aliasing properties. The approach of [19, 20, 10] differs ours in the use of a theorem prover in order to abstract reachable states in [19], and of counterexample-guided abstraction refinement in [20, 10]. They share with our analysis common benchmarks: for example, [20, 10] analyzes the program `full_init` in respectively 1.190 s and 0.27 s, and `partition` in 7.960 s and 3.6 s.[6] The returned invariants are the same as those given in Table 1. The other works [6, 7, 21] use the abstract interpretation framework. The analysis developed in [21] involves predicates on arrays and lists, and allows expressing invariants of the form $E \wedge \bigwedge_j \forall U_j(F_j \Rightarrow e_j)$, where $E$, $F_j$ and $e_j$ are quantifier-free facts. This approach is more general than ours, since it automatically discovers universally quantified predicates, while we explicitely define the family of predicates (uni- or two-dimensional) in our analysis. The drawback is that it requires under-approximation abstract domains and associated operators because of the universal quantification. In constrast, our concretization operator (defined in (2)) involves a universal quantifier over valuations $\nu \in V$, which can be shown to commute with the existential quantifier $\exists R \in \mathsf{Span}(E + \Omega)$. Then, *exact* operations on the inequality constraint systems representing the valuations, such as intersections or assignments, yield sound and precise results (see Sect. 3.2). In [21], `full_init` and `partition` are respectively analyzed in 3.2 s and 73.0 s on a 3 GHz machine, yielding the same invariants than with our analysis. In [6], semantic loop unrolling and introduction by heuristics of well-chosen degenerate predicates (called *tautologies*) are combined. It handles array initialization algorithm (the exact nature of the algorithm, partial, incrementing, decrementing, *etc*, is not mentionned), and bubble sort and QuickSort algorithms. In [7], array elements are distinguished according to their position w.r.t. to the current loop index (strictly before, equal to, or strictly after). This yields a partition of the memory configurations into distinct categories, which are characterized by the presence or the absence of array elements having a certain position w.r.t. to a loop index. The program `partial_init` is analyzed in 40 s on a 2.4 GHz machine, and yields a partition of four memory configurations corresponding to the invariant given in Table 1. Finally, as far as we know, no existing work reports any experiments on two-dimensional array manipulation programs.

## 6 Conclusion

We have introduced a numerical abstract domain which allows to represent sets of equivalence classes of predicates, by inferring affine invariants on some representatives of each class, without any heuristics. Combined with array predicates, it has been experimented in a sound static analysis of array and matrix manipulation programs. Experimental results are very good, and the approach is

[6] A 1.7 GHz machine was used in both works.

sufficiently robust to handle several array traversing stategies. Future work will focus on the extension of the abstraction to other systems of generators, such as convex polyhedra, in order to incorporate the reduced product implemented in the analysis into the abstraction of equivalence classes.

## References


1. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL'77, Los Angeles, California, ACM Press, New York, NY (1977)
2. Miné, A.: The octagon abstract domain. In: AST 2001 in WCRE 2001. IEEE, IEEE CS Press (2001) 310–319
3. Karr, M.: Affine relationships among variables of a program. Acta Inf. **6** (1976)
4. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL'78, Tucson, Arizona, USA, ACM Press (1978)
5. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. Journal of Logic Programming **13**(2–3) (1992) 103–179
6. Cousot, P.: Verification by abstract interpretation. In: Proc. Int. Symp. on Verification – Theory & Practice, Springer Verlag (2003)
7. Gopan, D., Reps, T., Sagiv, M.: A framework for numeric analysis of array operations. SIGPLAN Not. **40**(1) (2005)
8. Mauborgne, L., Rival, X.: Trace Partitioning in Abstract Interpretation Based Static Analyzers. In: ESOP'05. (2005)
9. Müller-Olm, M., Seidl, H.: A Note on Karr's Algorithm. In: ICALP. Volume 3142 of LNCS., Springer (2004)
10. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: PLDI'07, New York, NY, USA, ACM Press (2007)
11. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3–valued logic. In: POPL'99. (1999)
12. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. SIGPLAN Not. **36**(5) (2001)
13. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: TACAS 2006. LNCS, Springer (2006)
14. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P., Wies, T., Yang, H.: Shape analysis for composite data structures. In: CAV'07. (2007) To appear.
15. Beyer, D., Henzinger, T.A., Théoduloz, G.: Lazy shape analysis. In: CAV'06. Volume 4114 of LNCS., Springer-Verlag, Berlin (2006)
16. Venet, A., Brat, G.: Precise and efficient static array bound checking for large embedded C programs. In: PLDI '04, New York, NY, USA, ACM Press (2004)
17. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE Analyser. In: ESOP'05. LNCS 3444, Springer (2005)
18. Allamigeon, X., Godard, W., Hymans, C.: Static Analysis of String Manipulations in Critical Embedded C Programs. In: SAS'06. LNCS 4134, Springer (2006)
19. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL'02, New York, NY, USA, ACM Press (2002)
20. Jhala, R., McMillan, K.L.: Array abstractions from proofs. In: CAV'07. Volume 4590 of LNCS., Springer (2007)
21. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL'08 (to appear). (2008)