Invariant and Type Inference for Matrices*

Thomas A. Henzinger[1], Thibaud Hottelier[2], Laura Kovács[3], and Andrei Voronkov[4]

[1] IST Austria (Institute of Science and Technology Austria)
[2] UC Berkeley
[3] ETH Zürich
[4] University of Manchester

Abstract. We present a loop property generation method for loops iterating over multi-dimensional arrays. When used on matrices, our method is able to infer their shapes (also called types), such as upper-triangular, diagonal, etc. To generate loop properties, we first transform a nested loop iterating over a multi-dimensional array into an equivalent collection of unnested loops. Then, we infer quantified loop invariants for each unnested loop using a generalization of a recurrence-based invariant generation technique. These loop invariants give us conditions on matrices from which we can derive matrix types automatically using theorem provers. Invariant generation is implemented in the software package Aligator and types are derived by theorem provers and SMT solvers, including Vampire and Z3. When run on the Java matrix package JAMA, our tool was able to infer automatically all matrix types describing the matrix shapes guaranteed by JAMA's API.

1 Introduction

Static reasoning about unbounded data structures such as one- or multi-dimensional arrays is both interesting and hard [8, 10, 6, 14, 1, 11, 21, 12, 19, 27]. Loop invariants over arrays can express relationships among array elements and properties involving array and scalar variables of the loop, and thus simplify program analysis and verification.

We present a method for an automatic inference of quantified invariants for loops iterating linearly over all elements of multi-dimensional (mD) arrays, demonstrated here for matrices. It is based on the following steps. First, we rewrite nested loops with conditional updates over matrices into equivalent collections of unnested loops over matrices without conditionals (Section 6). We call this step *loop synthesis*. In order to derive such a collection of loops automatically, we take into account each branch condition and construct a loop encoding this condition. This is done using symbolic summation together with constraint solving. After that, for each loop so derived we compute polynomial invariants using symbolic computation techniques, and then infer *quantified invariants over arrays* in the combined theory of scalars, arrays, and uninterpreted functions, by generalizing the recurrence-based invariant generation technique of [18] to mD arrays (Section 7). The conjunction of the generated quantified invariants can be used to find post-conditions of loops expressing properties of the matrices. From these post-conditions we can derive, using a theorem prover, shape properties

* The results presented here were obtained while the first three authors were at EPFL, Switzerland. The research was supported by the Swiss NSF.

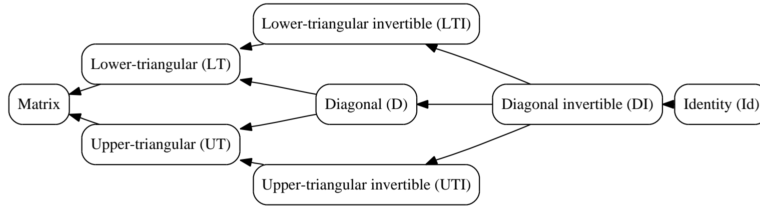

Fig. 1. Matrix type system. The arrows represent the subtyping relation

of matrices, such as upper/lower-triangular matrices, identity matrices etc. (Section 9). We call these properties *matrix types* as they characterize particular types of matrices (Section 3).

Our method for invariant generation and deriving loop properties is sound. It is complete for generating invariants over scalars for a certain class of programs in which all branch conditions in loops are linear. In practice, all matrix loops in the Java matrix package JAMA [13] turned out to have linear branch conditions.

We implemented our approach to invariant generation in the Aligator software package [17] (Section 8). We have shown that the generated proof obligations can be proved automatically by modern theorem provers and SMT solvers. When run on the JAMA package, our technique is able to infer matrix properties which imply all matrix shapes guaranteed by JAMA's API and prove the implication automatically using theorem provers. We successfully ran our system on over 3,000 lines of JAMA code.

We are not aware of any other automated method that can automatically infer quantified properties for programs over mD (or even 2D) arrays without user guidance, such as providing templates, assertions or predicates. The novel features presented in this paper are as follows.

- The basis of our method is a new technique for transforming nested loops iterating over matrices into equivalent collections of unnested loops. This technique uses symbolic summation and constraint solving and improves our previous method [18] of invariant generation over scalars, which could only handle unnested loops over scalars. This technique is general and not specifically intended for programs handling matrices.
- We are able to generate invariants for programs over mD arrays such as matrices. We show that the generated invariants are strong enough to derive matrix types. We do not need a theorem prover as in [19] to generate these invariants.
- We require no user guidance, such as predefined predicates, templates, or annotations to automatically derive quantified loop properties for the class of loops we study.
- We show that the generated matrix properties are strong enough to prove that matrices have corresponding shapes completely automatically, by using SMT solvers or first-order theorem provers with suitably axiomatised subsets of arithmetic.

The long term goal of our work is to verify various properties of domain-specific packages, such as Mathematica [30], Matlab[4], or Mathcad [2], having explicit matrix types.

2 Related Work

Paper [6] addresses the problem of automatically inferring auxiliary annotations such as *invariants and post-conditions* for safety property verification of programs over *mD arrays*. The method relies on using code patterns for describing code constructs that require annotations, and templates to describe the annotations that are required by code patterns. For each program code and safety property some user-guidance is thus needed to identify the relevant code and template patterns. Annotation templates are then embedded in the code, resulting in the automatic generation of program annotations.

Our approach can also be compared to *quantified invariant generation* methods *over 1D arrays*, such as [8, 14, 1, 11, 21, 27]. The methods used in the cited works combine inductive reasoning with predicate abstraction, constraint solving, and interpolation-based techniques, and require user guidance in providing necessary templates, assertions, or predicates. The various approaches differ in the extent of their required user guidance: papers [8, 11, 27] infer invariants by iteratively approximating the strongest boolean combination of a given set of predicates, whereas [14, 1, 21] search for appropriate invariant predicates using a given set of templates that define the boolean structure of the desired invariants.

Compared to the above mentioned work, our approach does not require a priori fixed templates and predicates. We derive quantified invariants directly from the loop description. Our technique allows one to generate properties of mD-arrays programs, which only the authors of [6] have done so far.

Papers [10, 12] do not require user guidance and derive quantified array invariants by using abstract interpretation and partitioning array indexes into symbolic intervals. Our approach handles a richer subset of arithmetic over scalar variables and works for mD arrays.

In our previous paper [19] we derive quantified invariants by combining symbolic computation and first-order theorem proving. The approach requires no user guidance and allows one to infer quantified invariants with alternating quantifiers. In this paper we do not use theorem proving and cannot derive properties requiring quantifier alternations. It would be interesting to integrate the method of [19] into ours, in order to find more complex quantified invariants and properties, such as sortedness and permutation properties of arrays or matrices.

Since one ingredient of our method is numeric invariant generation, we also compare it with other *polynomial invariant generation* techniques [22, 25, 24]. Papers [22, 25] compute polynomial equalities of fixed degree as invariants using the polynomial ideal theory. Unlike [22, 25], our method does not impose bounds on polynomials: we derive polynomial invariants of an arbitrary degree from which any other polynomial invariant can be inferred. Our algorithm thus returns a finite representation of the polynomial invariant ideal, whereas [22, 25] may only iteratively increase the polynomial degree to infer such a basis. Paper [24] derives polynomial invariants for loops with positive rational eigenvalues, by iteratively approximating the polynomial invariant ideal using Gröbner basis computation [3]. In contrast to [24], our approach generates polynomial invariants over scalars for polynomial loops with algebraic, and not just rational, eigenvalues.

```
for (i:=1; i ≤ n; i++) do
  for (j:=1; j ≤ n; j++) do
    if (i > j)
      then L[i,j]:=LU[i,j];
      else if (i = j)
             then L[i,j]:=1;
             else L[i,j]:=0;
  end do
end do
```

Fig. 2. Lower unit triangular part computation [12]

```
% Guard: i > j                | % Guard: i = j                | % Guard: i < j
i := c; j := 0;               | i := 0; j := 0;               | i := 0; j := c;
while (j < n) do              | while (j < n) do              | while (i < n) do
  i := i + 1; j := j + 1;     |   i := i + 1; j := j + 1;     |   i := i + 1; j := j + 1;
  L[i, j] := LU[i, j]         |   L[i, j] := 1                |   L[i, j] := 0
end do                        | end do                        | end do
```

Here c ranges over $\{1, \ldots, n-1\}$

Fig. 3. Loop sequence for Fig.2

3 Matrix Types

Figure 1 describes the matrix properties that we can infer for loops iterating over an $n{\times}n$ square matrix A. These properties are expressed by first-order formulas. We will refer to these properties and the formulas expressing them as *matrix types*. We only give the types for the lower-triangular (LT), lower-triangular invertible (LTI), diagonal (D), diagonal invertible (DI) and identity (Id) matrices, leaving the upper-triangular (UT) and upper-triangular invertible (UTI) types to the reader.

LT: $\forall i,j.\ 1{\leq}i < j{\leq}n \Rightarrow A[i,j] = 0$
LTI: $\forall i,j.\ 1{\leq}i{\leq}j{\leq}n \Rightarrow (i < j \Rightarrow A[i,j] = 0)\ \wedge\ (i = j \Rightarrow A[i,j]{\neq}0)$
D: $\forall i,j.\ 1{\leq}i,j{\leq}n \Rightarrow (i{\neq}j \Rightarrow A[i,j] = 0)$
DI: $\forall i,j.\ 1{\leq}i,j{\leq}n \Rightarrow (i{\neq}j \Rightarrow A[i,j] = 0)\ \wedge\ (i = j{\Rightarrow}A[i,j]{\neq}0)$
Id: $\forall i,j.\ 1{\leq}i,j{\leq}n \Rightarrow (i{\neq}j{\Rightarrow}A[i,j] = 0)\ \wedge\ (i = j{\Rightarrow}A[i,j] = 1)$

For checking invertibility, we use the fact that triangular and diagonal matrices are invertible if and only if every element on the main diagonal is non-zero.

To check whether a matrix is of a given type, we first infer quantified loop properties for a loop iterating over the matrix, as described in Sections 6 and 7, and then prove that the inferred properties imply the matrix type (Section 9).

4 Motivating Example

We give an example illustrating for what kind of loop we would like to infer quantified properties sufficient to derive matrix types.

Consider the program of Figure 2. This program is taken from the JAMA library [13]. We will use this example as our running example throughout the paper. The program computes the lower unit triangular part of an $n \times n$ square matrix LU [9]. This means that the resulting matrix L has only 0s above the main diagonal, only 1s on the main diagonal, and all entries of L below the main diagonal are equal to the corresponding entries of the matrix LU. We need invariants for this program that would help us to prove matrix types of L by using quantifiers over the matrix indexes. The difficulties for automatically finding such loop invariants come from the presence of nested loops, the use of scalar and matrix variables, and the nested conditional used in the loop. We overcome these difficulties as follows.

1. We rewrite the nested loop with conditional updates over L and LU into an equivalent, in some sense, collection of unnested loops without conditionals over matrices L and LU as shown in Figure 3. In this figure the constant c ranges over

$\{1, \ldots, n-1\}$ and appears from the fact that $i > j \iff \exists c(c > 0 \land i = c + j)$ (respectively, $i < j \iff \exists c(c > 0 \land j = c + i)$), see Section 6 for details.

2. We infer scalar invariants and quantified array invariants for each unnested loop using symbolic computation methods. The conjunction of the inferred quantified invariants of the unnested loops expresses matrix loop properties as postconditions of the nested loop with conditionals.

The matrix loop property derived for the loop of Figure 2 is given below.

$$\bigwedge \begin{cases} \forall i, j.\ 1 \leq i, j \leq n \ \Rightarrow \ (\ \forall c.\ c > 0 \ \land \ i = j + c \ \Rightarrow \\ \qquad (\forall k.\ 1 \leq k \leq j \ \Rightarrow \ L[k+c, k] = LU[k+c, k])\ ) \\ \forall i, j.\ 1 \leq i, j \leq n \ \Rightarrow \ (\forall k.\ 1 \leq k \leq j \ \Rightarrow \ L[k, k] = 1) \\ \forall i, j.\ 1 \leq i, j \leq n \ \Rightarrow \ (\ \forall c.\ c > 0 \ \land \ j = i + c \ \Rightarrow \\ \qquad (\forall k.\ 1 \leq k \leq i \ \Rightarrow \ L[k, k+c] = 0)\ ) \end{cases} \tag{1}$$

Here the first quantified conjunct expresses that the lower part of L is updated by elements of LU; the second conjunct describes that the elements of L from its main diagonal are 1s; and the third conjunct expresses that the elements of L above its main diagonal are 0s. The LTI-type of L can be proved from this inferred property.

Note that executing the loops of Figure 3 might access matrix elements which are actually out of the bounds of the $n \times n$ matrices L and LU (for example, when $c = n-1$, $i = n$, $j = n$). However, Figure 3 will never be executed in our work. In our approach, the unnested loops of Figure 3 are "only" used to generate invariant properties. These invariants, together with the property capturing the relation between the constant c and the matrix bounds i and j, are then further used to derive matrix loop properties of Figure 2. Access to matrix elements in the loop properties used for proving matrix types are thus between valid matrix bounds.

In this paper we derive two kinds of loop properties. One kind expresses conditions on scalar and mD-array variables used in the loop. These conditions are loop invariants, and we refer to them as, respectively, *scalar and quantified array invariants*. Another kind of property is a quantified condition on the values of the arrays at the loop exit, cf. formula (1). This condition is a *valid postcondition* of the loop, however, it is not a loop invariant. In the rest of the paper we will make a distinction between invariants and valid postconditions and refer to the latter as (quantified) *loop properties*, (quantified) *matrix loop properties*, or *matrix properties*.

The rest of the paper discusses in detail how we automatically infer scalar invariants, array invariants and matrix properties, and prove matrix types from these matrix properties.

5 Programming Model

This section fixes the relevant notation and introduces our model of programs.

Algebraic notation. Let $\mathbb{N}$ and $\mathbb{Z}$ denote respectively the sets of natural and integer numbers, and $\mathbb{Z}[x]$ denote the ring of polynomial relations in indeterminate x over $\mathbb{Z}$.

Variables. We assume that programs contain *scalar variables* denoted by lower-case letters $a, b, c, \ldots$ and *matrix variables* denoted by capital-case letters $A, B, C, \ldots$. All notations may have indices. W.l.o.g. we assume that matrices are square and reserve the lower-case letter n for their dimension.

Expressions and their semantics. We assume that expressions contain integer constants, variables over scalars and matrices, logical variables, and some function and predicate symbols. We only consider the arithmetical function symbols $+$, $-$, and $\cdot$ as interpreted, all other function symbols are uninterpreted. Similarly, only the arithmetical predicate symbols $=, \neq, \leq, \geq, <$ and $>$ are interpreted, all other predicate symbols are treated as uninterpreted.

Programs and their semantics. We consider programs of the following form, iterating over matrices.

$$\begin{array}{l} \underline{\textbf{for}}\ (i := l_i;\ i \leq n;\ i := i + u_i)\underline{\textbf{do}} \\ \quad \underline{\textbf{for}}\ (j := l_j;\ j \leq n;\ j := j + u_j)\underline{\textbf{do}} \\ \quad\quad \ldots \text{ loop body } \ldots \\ \quad \underline{\textbf{end do}} \\ \underline{\textbf{end do}} \end{array} \tag{2}$$

with $l_i, l_j, u_i, u_j \in \mathbb{Z}$, and the loop body consists of (nested) conditionals, sequencing, and assignments over scalar and matrix variables satisfying some properties formulated below in this section. For the moment, we restrict ourselves to the case when $l_i = l_j = u_i = u_j = 1$. Such programs contain a nested for-loop iterating linearly (row-by-row or column-by-column) over the matrix content by incrementing or decrementing the matrix row and column indices. Let $\mathcal{P}$ be such a program. In the sequel we assume that $\mathcal{P}$ is fixed and present our approach relative to it.

We denote respectively by Var and $Matr$ the sets of scalar and matrix variables of $\mathcal{P}$, where $Matr = RMatr \cup WMatr$ is a disjoint union of the sets $RMatr$ of *read-only* and $WMatr$ of *write-only* matrix variables. Throughout this paper, we assume that $i, j \in Var$ are the loop iteration/index variables of (2). As usual, the expression $A[k, l]$ is used to denote the element of an array A at the row k and column l.

Guarded assignments. Since the loop body of (2) is loop-free, we can equivalently consider it as the collection of all its paths. Every path can be written as a guarded *guarded assignments* [7] of the form

$$G \rightarrow \alpha_1; \ldots; \alpha_s, \tag{3}$$

where G is a formula, called the *guard* of this guarded assignments, and each of the α_k's is an assignment over $Var \cup Matr$. To turn a path into a guarded assignment, we collect all the tests satisfied on the path in the guard and write all assignments on the right of $\rightarrow$ keeping their relative order. This gives us an equivalent representation of the innermost loop body of $\mathcal{P}$ as a collection of guarded assignments of the form given below.

$$\begin{array}{lcl} G_1 & \rightarrow & \alpha_{11}; \ldots; \alpha_{1s_1}, \\ & \cdots & \\ G_d & \rightarrow & \alpha_{d1}; \ldots; \alpha_{ds_d}. \end{array} \tag{4}$$

Since each guard corresponds to a different path, in every state *exactly one guard holds*. That is, the formula $G_k \wedge G_l$ is unsatisfiable for $k \neq l$ and the formula $G_1 \vee \cdots \vee G_d$ is true in all states.

Conditions on loop bodies. After we rewrite loop bodies as collections of guarded assignments as given above, we require the following conditions to hold:

1. Each guard G_k is equivalent to an integer polynomial relation of the form

$$i \mathrel{\mathcal{R}} P(j) \quad \text{or} \quad j \mathrel{\mathcal{R}} Q(i), \tag{5}$$

   where $\mathcal{R} \in \{=, \neq, <, >, \leq, \geq\}$, $P \in \mathbb{Z}[j]$ with $1 \leq \text{degree}(P) \leq 2$, and $Q \in \mathbb{Z}[i]$ with $1 \leq \text{degree}(Q) \leq 2$.
2. If some α_{ku} updates a matrix variable $A_u \in \mathit{WMatr}$, and some α_{kv} for $u \neq v$ in the *same guarded assignment* updates a matrix variable $A_v \in \mathit{WMatr}$, then A_u and A_v are different matrices.
3. The assignments α_{ku}'s have one of the forms given below.
   (a) Matrix assignments:

$$A[i,j] \;\; \mathbf{:=} \;\; f(\mathit{Var} \cup \mathit{RMatr}), \tag{6}$$

   where $A \in \mathit{WMatr}$ and $f(\mathit{Var} \cup \mathit{RMatr})$ is an arbitrary expression over the variables $\mathit{Var} \cup \mathit{RMatr}$. That is, this expression may contain arbitrary interpreted or uninterpreted functions but does not contain write-arrays.
   (b) Scalar assignments over $x_l \in \mathit{Var}$:

$$x_l := c_0 + c_l \cdot x_l + \sum_{\sigma \in M(\mathit{Var} \setminus \{x_l\})} c_\sigma \cdot \sigma, \tag{7}$$

   where
   - $c_0, c_l, c_\sigma \in \mathbb{Z}$ and $c_l \neq 0$;
   - $c_l \neq 1$ or $c_0 \neq 0$ or $c_\sigma \neq 0$ for some σ.
   - $M(\{x_1, \ldots, x_k\}) = \{x_1^{r_1} \cdots x_k^{r_k} \mid 1 \leq r_1 + \cdots + r_k \leq 2, r_1, \ldots, r_k \in \{0,1\}\}$ is a subset of monomials over $\{x_1, \ldots, x_k\} \subseteq \mathit{Var}$.

The program of Figure 2 trivially satisfies conditions 2 and 3. Its transformation to guarded assignments does not immediately satisfy condition 1, despite that all tests in the program are of the required form $i \mathrel{\mathcal{R}} P(j)$. The problem is that having more than one if-then-else expression results in guards that are *conjunctions* of formulas $i \mathrel{\mathcal{R}} P(j)$, while property 1 requires to have a single formula instead of a conjunction.

Example 1. The loop body of the program of Figure 2 gives rise to the collection of guarded assignments shown below on the left. On the right we give its equivalent representation in which the guards satisfy condition 1.

$i > j \rightarrow L[i,j] := LU[i,j]$	$i > j \rightarrow L[i,j] := LU[i,j]$
$\neg(i > j) \wedge i = j \rightarrow L[i,j] := 1$	$i = j \rightarrow L[i,j] := 1$
$\neg(i > j) \wedge i < j \rightarrow L[i,j] := 0$	$i < j \rightarrow L[i,j] := 0$

The matrix L is conditionally updated at different positions (i,j). Updates over L involve initializations by 0 or 1, and copying from LU.

It is worth mentioning that our experiments over the JAMA library show that (i) in matrix programs nonlinear polynomial expressions over scalars are relatively rare and are of degree at most 2; (ii) polynomial tests on matrix indices are usually linear or otherwise of degree 2; (iii) the operations used for constructing matrices of specific shapes only involve initialization or copying from another matrix. Therefore, we believe

that the restrictions on (5)-(7) cover a significant part of practical applications. It is also worth noting that properties 2 and 3a can be easily generalised so that our method still works: we only need to guarantee that matrices are never updated twice at the same positions.

One can relax and/or modify some of the conditions on the loops formulated here, however, the page limit prevents us from discussing possible modifications.

6 Loop Synthesis

Our aim is to find an explicit representation of the loop scalar variables in terms of the loop counters i and j. Conditions in if-then-else expressions are a main obstacle for doing that. In order to solve the problem, we transform $\mathcal{P}$ into an equivalent, in some sense, collection of unnested while-loops without conditionals, so that each unnested loop encodes the behavior of one conditional branch of $\mathcal{P}$. The unnested loops will be parametrised by new constants, similar to the constant c in Section 4, so that every suitable value of these constants, gives a separate unnested loop.

The transformation is performed separately for each guarded assignment $G \rightarrow \alpha_1; \ldots; \alpha_s$ from (4) and described below.

The general shape of the desired loop is

$$\underline{\textbf{while}}\ \ (index_{i,j} < n)\ \ \underline{\textbf{do}}\ \ \beta_i; \beta_j; \alpha_1; \ldots; \alpha_s\ \ \underline{\textbf{end do}},$$

where β_i and β_j are respectively the assignments to be constructed for i and j, and $index_{i,j}$ is either i or j.

To infer such a loop automatically, a case analysis on the shape of G is performed, as given below. We only present the case when G is $i\,\mathcal{R}\,P(j)$. In what follows, we denote by m the iteration counter of the loop being constructed. For a variable x, we denote by $x^{(m)}$ the value of x at the iteration m, whereas $x^{(0)}$ will stand for the initial value of x (i.e. its value before entering the loop). Note that $1 \leq i^{(m)}, j^{(m)} \leq n$.

Case 1: G **is** $i = P(j)$. **While-loop condition.** The guard G describes the values of i as polynomial expressions of degree at most 2 in j, and leaves j as an "independent" variable. For this reason, we take $index_{i,j} = j$ and construct a while-loop iterating over values i and j such that at each loop iteration G is a valid polynomial relation (i.e. loop invariant) among i and j.

While-loop body. The scalar assignments to i and j of the while-loop being constructed should satisfy the structural constraints of (7). For inferring these assignments, we use symbolic summation and constraint solving as described below.

As the while-loop condition depends on the values of $index_{i,j} = j$, we identify j to be in a linear correspondence with m. The generic assignments for i and j are built as given in (7), and the coefficients c_0, c_l and c_σ are treated as unknowns. As G involves only the variables i and j, the assignments of i and j need to be constructed only over i and j. We thus have

$$i := c_4 \cdot i + c_3 \cdot j + c_2;\ j := c_1 \cdot j + c_0,$$

where $(c_1 \neq 0)\ \wedge\ (c_1 \neq 1 \vee c_0 \neq 0)\ \wedge\ (c_4 \neq 0)\ \wedge\ (c_4 \neq 1 \vee c_3 \neq 0 \vee c_2 \neq 0)$.

Moreover, as G is a polynomial expression in i and j, the multiplicative coefficients c_4 and c_1 of i and j can be considered w.l.o.g. to be 1. We then have

$$i := i + c_3 \cdot j + c_2;\ j := j + c_0,\ \text{with} \tag{8}$$

$$c_0 \neq 0\ \wedge\ (c_3 \neq 0 \vee c_2 \neq 0). \tag{9}$$

From (8), we next derive the *system of recurrences* of i and j over m:

$$\begin{cases} i^{(m+1)} &= i^{(m)} + c_3 \cdot j^{(m)} + c_2 \\ j^{(m+1)} &= j^{(m)} + c_0 \end{cases}$$

Further, we compute the generic closed forms $i^{(m)}$ and $j^{(m)}$ as polynomial functions of m, $i^{(0)}$, and $j^{(0)}$, by symbolic summation and computer algebra techniques as discussed in [18]. We hence obtain:

$$\begin{cases} i^{(m)} = i^{(0)} + (c_2 + c_3 \cdot j^{(0)}) \cdot m + \frac{c_0 \cdot c_3}{2} \cdot m \cdot (m-1) \\ j^{(m)} = j^{(0)} + c_0 \cdot m \end{cases} \tag{10}$$

Next, closed forms $i^{(m)}$ and $j^{(m)}$ from (10) are substituted for variables i and j in G, and a polynomial relation in the indeterminate m is derived, as given below:

$$\sum_{k=0}^{2} q_k \cdot m^k = 0, \tag{11}$$

where the coefficients $q_k \in \mathbb{Z}$ are expressions over c_0, c_2, c_3, $i^{(0)}$, and $j^{(0)}$. Using properties of null-polynomials, we conclude that each q_k must equal to 0, obtaining a system of polynomial equations on c_0, c_2, c_3, $i^{(0)}$, and $j^{(0)}$. Such a system can be algorithmically solved by linear algebra or polynomial ideal theory [3], as discussed below.

Linear algebra methods (e.g. Gaussian elimination) offer an algorithmic way to derive integer solutions to a system of linear equations over integers. When G is *linear*, equations (9) and (11) yield a linear [1] constraint system over $c_0, c_2, c_3, i^{(0)}, j^{(0)}$. Hence, a finite representation of the sets of integers solutions for $c_0, c_2, c_3, i^{(0)}, j^{(0)}$ can be always constructed explicitly[2]. The loop assignments over i and j, such that the ideal of all polynomial invariant relations among i and j is generated by G, are thus always derived. Our loop synthesis method is hence complete in transforming nested loops over matrices with linear guards (e.g. JAMA benchmarks) into an equivalent collection of unnested loops.

When G is a *non-linear polynomial* relation (i.e. of degree 2), (11) yields a system of non-linear polynomial equations. Solving this system is done using Gröbner basis computation, which however may yield non-integer (and not even rational) solutions for $c_0, c_2, c_3, i^{(0)}, j^{(0)}$. In such cases, as matrix indices need to be integer valued, our method fails constructing unnested loops over matrix and scalar variables. It is worth to be mentioned though that for all examples we have tried (see Section 8), integer solutions for $c_0, c_2, c_3, i^{(0)}, j^{(0)}$ have successfully been inferred.

Example 2. Consider the condition $i = j$ from Figure 2. The condition of the while-loop being constructed is $j < n$. Substituting generic closed forms (10) into $i = j$, we derive the polynomial relation

$$2 \cdot (i^{(0)} - j^{(0)}) + (2 \cdot c_2 - 2 \cdot c_0 - c_0 \cdot c_3 + 2 \cdot c_3 \cdot j^{(0)}) \cdot m + c_0 \cdot c_3 \cdot m^2 = 0$$

[1] Linearity of G, together with (9) and (11), implies $c_0 \neq 0$, $c_3 = 0$ and $c_2 \neq 0$.

[2] In our work, we take the smallest integer solution for $c_0, c_2, c_3, i^{(0)}, j^{(0)}$.

Program	Branch ♯	$[d, \mathcal{R}]$	Time (s)	Matrix Types
LU decomposition.getL	3	$[1,>]$, $[1,=]$, $[1,<]$	0.52	LT, LTI
LU decomposition.getU	2	$[1,\leq]$, $[1,>]$	0.37	UT
QR decomposition.getR	3	$[1,<]$, $[1,=]$, $[1,>]$	0.57	UT, UTI
QR decomposition.getH	2	$[1,\geq]$, $[1,<]$	0.37	LT
Matrix.identity	2	$[1,=]$, $[1,\neq]$	0.32	LT, UT, LTI, UTI, D, DI, Id

Table 1. Experimental results using Aligator on JAMA programs

The coefficients of m now must equal to 0. This gives us the system

$$\begin{cases} i^{(0)} - j^{(0)} & = 0 \\ 2 \cdot c_2 - 2 \cdot c_0 - c_0 \cdot c_3 + 2 \cdot c_3 \cdot j^{(0)} & = 0 \\ c_0 \cdot c_3 & = 0 \end{cases}$$

Solving this system of equations and considering also constraints (9), we obtain $c_2 = c_0$, $i^{(0)} = j^{(0)}$, and $c_3 = 0$. We conclude that $c_2 = c_0 = 1$, $c_3 = 0$, and $i^{(0)} = j^{(0)} = 0$ are (up to constant multipliers) the desired solutions, yielding the loop assignments $i\!:=\!i+1; j\!:=\!j+1$, with the initial value assignments $i\!:=\!0; j\!:=\!0$.

The while-loop corresponding to the condition $i = j$ is given in Figure 3.

Case 2: G **is** $i\ \mathcal{R}\ P(j)$**, where** $\mathcal{R} \in \{<, >, \leq, \geq, \neq\}$**.** We only present the case when $\mathcal{R}$ is $>$, all other cases are handled in a similar manner.

Since G is $i > P(j)$, G is equivalent to the existentially quantified formula

$$\exists c \in \mathbb{N}.\ (c > 0\ \wedge\ i = P(j) + c).$$

Thus, we apply the approach discussed in Case 1 for deriving a while-loop *parameterized by* c, yielding $i = P(j) + c$ as one of its invariants.

Example 3. Consider the condition $i > j$ from Figure 2. Introducing an integer skolem constant $c > 0$, we first rewrite this condition into $i = j + c$. The condition of the while-loop being constructed is then $j < n$.

Substituting generic closed forms (10) into $i = j + c$, we derive

$$2 \cdot (i^{(0)} - j^{(0)} - c) + (2 \cdot c_2 - 2 \cdot c_0 - c_0 \cdot c_3 + 2 \cdot c_3 \cdot j^{(0)}) \cdot m + c_0 \cdot c_3 \cdot m^2 = 0$$

The coefficients of m must equal to 0. Considering also constraints (9), we obtain a linear constraint system over c_0, c_2, c_3, $i^{(0)}$, and $j^{(0)}$, yielding $c_2 = c_0$, $i^{(0)} = j^{(0)} + c$ and $c_3 = 0$. We conclude that $c_2 = c_0 = 1$, $c_3 = 0$, $i^{(0)} = c$, and $j^{(0)} = 0$, yielding the loop assignments $i\!:=\!i+1; j\!:=\!j+1$, with the initial values given by $i\!:=\!c; j\!:=\!0$.

The while-loop corresponding to the condition $i > j$ (respectively, $i < j$) is given in Figure 3.

Example 4. To illustrate the power of our synthesis method, consider the property $i = j^2$. We want to infer a loop yielding the invariant $i = j^2$. Applying our approach, the condition of the while-loop being constructed is $j < n$. The polynomial equation derived after substituting generic closed forms (10) into $i = j^2$ is

$$2 \cdot (i^{(0)} - {j^{(0)}}^2) + (2 \cdot c_2 - c_0 \cdot c_3 - 4 \cdot c_0 \cdot j^{(0)} + 2 \cdot c_3 \cdot j^{(0)}) \cdot m + (c_0 \cdot c_3 - c_0^2) \cdot m^2 = 0$$

We next solve the system of equations obtained by making the coefficients of m of the above polynomial equal to 0. Together with (9), we get $4 \cdot c_2 = c_3^2$, $i^{(0)} = {j^{(0)}}^2$, and $2 \cdot c_0 = c_3$. We conclude that $c_0 = 1$, $c_2 = 1$, $c_3 = 2$, and $i^{(0)} = j^{(0)} = 0$, yielding the loop assignments $i := i + 2 \cdot j + 1; j := j + 1$, with the initial value assignments $i := 0; j := 0$.

7 Generation of Loop Invariants and Properties

For each while-loop derived in Section 6, loop invariants and properties are inferred in the combined theory of scalars, arrays, and uninterpreted function symbols, by generalizing the technique described in [18] to arrays. To this end, we first compute numeric invariants over scalars and then use them to generate quantified array invariants. The conjunction of these invariants is an invariant of the while-loop. Finally, the conjunction of the inferred quantified array invariants of the while-loops expresses matrix loop properties as postconditions of the nested loop $\mathcal{P}$ with conditionals.

Invariant generation over scalars. We infer scalar (numeric) invariants by combining symbolic summation and computer algebra, as described in [18]. Namely, (i) we build recurrence equations for scalars over the loop iteration counter m, (2) compute closed forms of scalars as functions of m, and (3) eliminate variables in m from the system of closed forms. The generators of the polynomial invariant ideal of the loop are thus inferred.

Example 5. The closed form system of the second inner loop from Figure 3 is

$$\begin{cases} i^{(m)} = i^{(0)} + m \\ j^{(m)} = j^{(0)} + m \end{cases}$$

After eliminating m, and substituting the initial values $i^{(0)} = j^{(0)} = 0$, the derived polynomial invariant is $i = j$. Proceeding in a similar manner, we obtain the scalar invariant $i = j + c$ for the first loop of Figure 3, whereas the third loop of Figure 3 yields the scalar invariant $j = i + c$.

Invariant generation over mD arrays. We generalize the method described in [18] to infer quantified array invariants. Recall that we only handle array assignments of the form (6) with $A \in WMatr$. For inferring universally quantified array invariants over the content of A, we make use of the already computed closed forms of scalars. The closed forms of the matrix indices i and j describe the positions at which A is updated as functions of m. We note that array updates are performed by iterating over the array positions, where the update expressions involve only scalars, read-only array variables, and interpreted and uninterpreted function symbols. Thus the closed form of an array element is given by substituting the closed form solutions for each scalar variable in (6), and is expressed as a function of m as follows:

$$A[i^{(m)}, j^{(m)}] = f(Var^{(m)} \cup RMatr), \tag{12}$$

where $Var^{(m)} = \{x^{(m)} | x \in Var\}$.

Further, we rely on the following fact. For all loop iterations up to the current one given by m, the array update positions and array update expressions can be expressed as functions of m. As m is a new variable not appearing elsewhere in the loop, we treat it symbolically, noting that every possible value of m corresponds to a single loop iteration. Therefore we can strengthen (12) to the formula universally quantified over loop iterations up to m as follows:

$$\forall k.\ 1 \leq k \leq m \Rightarrow A[i^{(k)}, j^{(k)}] = f(Var^{(k)} \cup RMatr). \tag{13}$$

Program	Time (s)
Matrix.copy	< 0.1
Matrix.getMatrix	< 0.1
Matrix.constructWithCopy	< 0.1
Matrix.arrayLeftDivide	< 0.1
Matrix.arrayRightDivide	< 0.1
Matrix.times	< 0.1
Matrix.arrayTimes	< 0.1
Matrix.plus	< 0.1
Matrix.minus	< 0.1

Program	Time (s)
Matrix.getArrayCopy	< 0.1
Matrix.setMatrix	< 0.1
Matrix.uminus	< 0.1
Matrix.arrayLeftDivideEquals	< 0.1
Matrix.arrayRightDivideEquals	< 0.1
Matrix.timesEquals	< 0.1
Matrix.arrayTimesEquals	< 0.1
Matrix.plusEquals	< 0.1
Matrix.minusEquals	< 0.1

Table 2. Other loop properties inferred by Aligator on JAMA programs

We finally rewrite (13) as a quantified formula over $Var \cup Arr$, by eliminating m. For doing so, we rely once more on the closed forms of scalar variables, and express m as a linear function $g(Var) \in \mathbb{Z}[Var]$. This formula is given below:

$$\forall k.\ 1 \le k \le g(Var) \Rightarrow A[i^{(k)}, j^{(k)}] = f(Var^{(k)} \cup RMatr). \tag{14}$$

Formula (14) is a quantified array invariant over the content of A.

Example 6. Using the closed forms $i^{(m)} = i^{(0)} + m$ and $j^{(m)} = j^{(0)} + m$, the array assignment corresponding to the second loop of Figure 3 can be expressed as a function of the iteration counter m, as follows: $L[i^{(0)} + m, j^{(0)} + m] = 1$.

From Example 5, we have $m = j$, $i^{(0)} = 0$, and $j^{(0)} = 0$. The corresponding quantified array invariant of the second loop of Figure 3 is: $\forall k.\ (1 \le k \le j) \Rightarrow L[k, k] = 1$.

Similarly, we derive the following quantified array invariant of the first loop of Figure 3 as: $\forall k.\ (1 \le k \le j) \Rightarrow L[k + c, k] = LU[k + c, k]$.

Finally, the third loop of Figure 3 yields the array invariant: $\forall k.\ (1 \le k \le i) \Rightarrow L[k, k + c] = 0$.

Matrix loop properties of $\mathcal{P}$. We can now derive the following matrix loop property of $\mathcal{P}$:

$$\forall i, j.\ (1 \le i, j \le n)\ \Rightarrow\ \bigwedge_{k=1}^{d} \phi_k, \tag{15}$$

where ϕ_k satisfies one of the following conditions.

1. ϕ_k is a quantified array invariant (14) inferred for the while-loop corresponding to the guarded assignment from (4) with the guard $G_k \equiv i = P(j)$, or respectively with the guard $G_k \equiv j = Q(i)$.
2. ϕ_k is

   $\forall c.\ (c > 0 \land i = P(j) \pm c) \Rightarrow \phi_k^c$ or $\forall c.\ (c > 0 \land j = Q(i) \pm c) \Rightarrow \phi_k^c$

   where ϕ_k^c is a quantified array invariant (14) inferred for the while-loop corresponding to the guarded assignment from (4) with the guard $i = P(j) \pm c$, or respectively with the guard $j = Q(i) \pm c$. The formula ϕ_k is thus a quantified loop property of the while-loop corresponding to the guarded assignment from (4) with the guard $G_k \equiv i\, \mathcal{R}\, P(j)$, or respectively with the guard $G_k \equiv j\, \mathcal{R}\, Q(j)$, where $\mathcal{R} \in \{<, >, \le, \ge, \neq\}$.

The appropriate type of A can be proved from (15), as discussed in Section 9.

Example 7. The quantified array invariant ϕ_1 of the first loop of Figure 3 is:

$$\forall c.\ c > 0\ \land\ i = j + c\ \Rightarrow\ (\forall k.\ 1 \le k \le j\ \Rightarrow\ L[k + c, k] = LU[k + c, k])$$

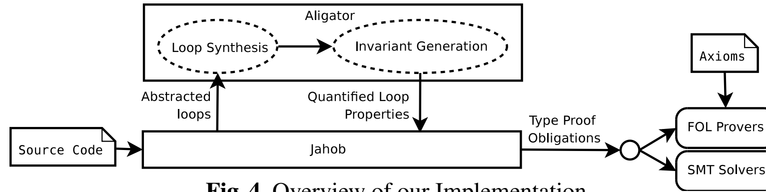

Fig. 4. Overview of our Implementation

The corresponding matrix loop property of the nested loop from Figure 2 is shown in (1). The LTI-type of L is a logical consequence of the formula given above and the lower-triangular invertible shape property of L can be thus inferred, as presented in Section 9.

8 Implementation and Experiments

Implementation. We implemented a tool that infers matrix loop properties as described in Sections 6 and 7. Our tool is implemented in the Jahob verification system [20]. It takes a Java program as its input, and returns a quantified matrix loop property for each loop from its input. In more detail, the main features of our tool are as follows.

(i) It extends the Jahob framework [20] by handling mD arrays and floating point numbers;
(ii) It performs all the preprocessing steps needed for translating loops in the format described in Section 5, for instance, finding read- and write-only arrays and checking that the guards are pairwise disjoint using the SMT solver Z3 [5].
(iii) Most importantly, it integrates the software package Aligator [17] for synthesizing loops and generating quantified array invariants and loop properties. To this end, we extended Aligator with constraint solving over integers and loop synthesis, and generalized the recurrence-based invariant generation algorithm of [17] over scalars to mD arrays.

Finally, using the derived matrix properties returned by our tool, matrix types for loops are inferred by running theorem provers on the resulting proof obligations induced by type checking, as described in Section 9.

The overall workflow of our implementation is illustrated in Figure 4.

Experiments. We ran our tool on the JAMA linear algebra package [13], which provides user-level classes for constructing and manipulating matrices in Java. All matrix types guaranteed by JAMA's API, which fall into our type system, have successfully been derived from the matrix loop properties generated by our tool. We summarize some of our results, obtained on a machine with a 2.0GHz CPU and 2GB of RAM, in Table 1. The first column of the table contains the name of the JAMA program, the second specifies the number of conditional branches in the innermost loop, whereas the third columns gives the degree d and relation $\mathcal{R}$ (equality, inequality or disequality) of the polynomial guard for each branch. The fourth column shows timing (in seconds) needed by Aligator to infer quantified matrix loop invariants and properties. The fifth column specifies which types we could automatically prove from the matrix loop properties using theorem provers. Theorem proving experiments are described in more detail in Section 9.

Proof obligations	Vampire (s)	E (s)	iProver (s)	Z3 (s)
LU decomposition.getL ⇒ LT	48	105	98	0.1
LU decomposition.getL ⇒ LTI	49	107	101	0.1
QR decomposition.getR ⇒ UT	53	109	410	0.1
QR decomposition.getH ⇒ LT	49	0.2	22	Unknown
LU decomposition.getU ⇒ UT	49	0.2	23	Unknown
Matrix.identity ⇒ LT	48	102	84	0.1
Matrix.identity ⇒ UT	49	112	6	0.1
Matrix.identity ⇒ LTI	48	103	86	0.1
Matrix.identity ⇒ UTI	49	112	8	0.1
Matrix.identity ⇒ D	97	214	90	0.1
Matrix.identity ⇒ DI	98	215	94	0.1
Matrix.identity ⇒ I	97	215	91	0.1
Average time	58.7	116.2	92.8	0.1

Table 3. Theorem proving results on JAMA programs

It is worth mentioning that our tool automatically inferred quantified array invariants and loop properties also for those JAMA programs which perform simple operations or provide access to submatrices or copies of given matrices. Such programs are e.g. Matrix.copy, Matrix.getMatrix, etc; the timings are shown by Table 2. The quantified properties of these loops do not explicitly describe matrix types, but they are strong enough so that a theorem prover can prove type-related properties, for instance, that a shape is preserved through a matrix copy, see Section 9.

We have also run our tool successfully on the JAMPACK library [28]. Results and timings are nearly identical to the ones in Table 1 and Table 2.

Aligator cannot yet handle programs with more complex matrix arithmetic. For example, JAMA loops implementing the Gaussian elimination algorithm involve various column and row switching and multiplying operations. We cannot generate loop properties implying that the resulting matrix is triangular. Handling such programs is beyond the scope of our technique but is an interesting subject for further research.

9 Type Checking Matrices

For automatic derivation of matrix loop properties one should be able to prove automatically formulas expressing that the derived loop properties imply corresponding matrix types. In this section we present experimental results showing that such formulas can be proved automatically by modern theorem provers.

Note that both the loop properties and matrix types are complex formulas with quantifiers and integer linear arithmetic. Combining first-order reasoning and linear arithmetic is very hard, for example, some simple fragments of this combination are Π_1^1-complete [16]. A calculus that integrates linear arithmetic reasoning into the superposition calculus is described in [16] but it is not yet implemented. There two kinds of tools that can be used for proving such formulas automatically.

First-order theorem provers. Such provers are very good in performing first-order reasoning with quantifiers but have no support for arithmetic. Partial and incomplete axiomatisations of fragments of arithmetic can be added to it. For example, this approach was used in generating loop invariants for programs over arrays in [19], and this is the approach we used in our experiments. Namely, we added the following formulas as axioms:

$$\forall i,j.\,(i \leq j \iff i < j \lor i = j); \qquad \forall i,j,k.\,(i < j \land j < k \Rightarrow i < k);$$
$$\forall i,j.\,(i < j \Rightarrow i \neq j); \qquad \forall i,j.\,(i < j \lor j \leq i);$$
$$0 < 1; \qquad \forall i.\,(0 < i \iff 1 \leq i);$$
$$\forall i,j.\,(i + j = j + i); \qquad \forall i.\,(i + 0 = i);$$
$$\forall i_1,j_1,i_2,j_2.\,(i_1 \leq j_1 \land i_2 \leq j_2 \Rightarrow i_1 + i_2 \leq j_1 + j_2);$$
$$\forall i,j,k.\,(i < j \iff \exists k(i + k = j \land 0 < k)).$$

These formulas axiomatise inequalities and addition. We used the following first-order theorem provers: Vampire [23], E [26] and iProver [15], the three fastest first-order provers at the last CASC competitions [29]. Vampire and E are based on the superposition calculus, iProver is an instantiation-based prover.

SMT solvers. Contrary to first-order theorem provers, SMT solvers are good in (quantifier-free) theory reasoning, including reasoning with linear arithmetic. To work with quantifiers, they instantiate universally quantified variables by ground terms using various heuristics. If a problem requires few such instances to be proved (which is the case for the proof obligations generated), SMT solvers can be very good in solving this problem. Among SMT solvers, we used Z3 [5] that has a good support for quantifiers.

The results of running the four systems on the hardest generated problems are summarised in Table 3. An example of a hard problem is given in Example 7: it is not immediately obvious how one should instantiate quantifiers in the generated loop property to prove that it implies the lower-triangular-invertible type. The results of Table 3 were obtained on a machine with eight 2.8GHz CPU and 16GB of RAM. For each run, the provers were limited to a single CPU and 2GB of RAM. It turned out that the three first-order theorem provers were able to prove all the proof obligations, while Z3 was unable to solve two of them. On the solved problems Z3 spent essentially no time while the first-order provers spent between 58.7s and 116.2s on the average.

We also ran Vampire on simpler problems. The simplest problems of this kind are that Matrix.copy preserves all types. Other simple properties involve loops applying the same operation to all element of a matrix, for example, that Matrix.uminus preserves all types apart from Identity. All these problems were proved by Vampire in essentially no time. It also turned out that many problems involving element-wise operations on more than one matrix are easy and proved in no time as well. One example is that Matrix.plus preserves the LTI property.

One conclusion of our experiments is that our method can be fully automated. On the other hand, some of the generated problems turned out to be highly non-trivial. This suggests that this and similar experiments may also help to improve theorem proving with quantifiers and theories, and therefore improve theorem proving support for program analysis and program verification. The generated problems have been added to the TPTP library [29].

10 Conclusions and Future Work

We address the problem of automatically inferring quantified invariants for programs iterating over mD arrays, such as matrices. For doing so, we combine symbolic summation with constraint solving to derive unnested loops iterating over mD arrays, and

use symbolic summation to generate loop invariants and properties in the combined theory of scalars, arrays, and uninterpreted functions. The inferred quantified loop invariants give us conditions on matrices from which we can derive matrix types using a first-order theorem prover. We implemented our approach to invariant generation in the Aligator package [17], successfully derived many matrix properties for all examples taken from the JAMA library [13], and used theorem provers and SMT solvers to prove automatically that these matrix properties imply matrix shapes guaranteed by the library.

We believe that the technique of generating invariants for loops with linear conditions introduced in Sections 6 and 7 has an independent value and can be used in other programs as well. Future work includes integrating our approach to loop property generation with techniques using predicate abstraction [11] and first order theorem proving [19], and extending our method to handle programs with more complex matrix arithmetic [13, 2, 30, 4].

References

1. D. Beyer, T. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant Synthesis for Combined Theories. In *Proc. of VMCAI*, pages 346–362, 2007.
2. B. Birkeland. *Calculus and Algebra with MathCad 2000*. Haeftad. Studentlitteratur, 2000.
3. B. Buchberger. An Algorithm for Finding the Basis Elements of the Residue Class Ring of a Zero Dimensional Polynomial Ideal. *J. of Symbolic Computation*, 41(3-4):475–511, 2006.
4. I. Danaila, P. Joly, S. M. Kaber, and M. Postel. *An Introduction to Scientific Computing: Twelve Computational Projects Solved with MATLAB*. Springer, 2007.
5. L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. In *Proc. of TACAS*, pages 337–340, 2008.
6. E. Denney and B. Fischer. A Generic Annotation Inference Algorithm for the Safety Certification of Automatically Generated Code. In *GPCE*, pages 121–130, 2006.
7. E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, 1975.
8. C. Flanagan and S. Qadeer. Predicate Abstraction for Software Verification. In *Proc. of POPL*, pages 191–202, 2002.
9. G. H. Golub and C. F. van Loan. *Matrix Computations*. Johns Hopkins Univ. Press, 1996.
10. D. Gopan, T. W. Reps, and M. Sagiv. A Framework for Numeric Analysis of Array Operations . In *Proc. of POPL*, pages 338–350, 2005.
11. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting Abstract Interpreters to Quantified Logical Domains. In *Proc. of POPL*, pages 235–246, 2008.
12. N. Halbwachs and M. Peron. Discovering Properties about Arrays in Simple Programs. In *Proc. of PLDI*, pages 339–348, 2008.
13. J. Hicklin, C. Moler, P. Webb, R. F. Boisvert, B. Miller, R. Pozo, and K. Remington. JAMA: A Java Matrix Package. http://math.nist.gov/javanumerics/jama/, 2005.
14. R. Jhala and K. L. McMillan. Array Abstractions from Proofs. In *Proc. of CAV*, pages 193–206, 2007.
15. K. Korovin. iProver - An Instantiation-based Theorem Prover for First-order Logic. In *Proc. of IJCAR*, pages 292–298, 2009.
16. K. Korovin and A. Voronkov. Integrating Linear Arithmetic into Superposition Calculus. In *Proc. of CSL*, volume 4646 of *LNCS*, pages 223–237, 2007.
17. L. Kovacs. Aligator: A Mathematica Package for Invariant Generation. In *Proc. of IJCAR*, pages 275–282, 2008.

18. L. Kovacs. Reasoning Algebraically About P-Solvable Loops. In *Proc. of TACAS*, pages 249–264, 2008.
19. L. Kovacs and A. Voronkov. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *Proc. of FASE*, pages 470–485, 2009.
20. V. Kuncak and M. Rinard. An overview of the Jahob analysis system: Project goals and current status. In *NSF Next Generation Software Workshop*, 2006.
21. K. L. McMillan. Quantified Invariant Generation Using an Interpolating Saturation Prover. In *Proc. of TACAS*, pages 413–427, 2008.
22. M. Müller-Olm and H. Seidl. Computing Polynomial Program Invariants. *Indormation Processing Letters*, 91(5):233–244, 2004.
23. A. Riazanov and A. Voronkov. The Design and Implementation of Vampire. *AI Communications*, 15(2-3):91–110, 2002.
24. E. Rodriguez-Carbonell and D. Kapur. Generating All Polynomial Invariants in Simple Loops. *J. of Symbolic Computation*, 42(4):443–476, 2007.
25. S. Sankaranaryanan, H. B. Sipma, and Z. Manna. Non-Linear Loop Invariant Generation using Gröbner Bases. In *Proc. of POPL*, pages 318–329, 2004.
26. S. Schulz. E — a brainiac theorem prover. *AI Communications*, 15(2-3):111–126, 2002.
27. S. Srivastava and S. Gulwani. Program Verification using Templates over Predicate Abstraction. In *Proc. of PLDI*, pages 223–234, 2009.
28. G. W. Stewart. JAMPACK: A Java Package For Matrix Computations. http://www.mathematik.hu-berlin.de/ lamour/software/JAVA/Jampack/.
29. G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. The FOF and CNF Parts, v3.5.0. *J. of Automated Reasoning*, To appear, 2009.
30. S. Wolfram. *The Mathematica Book. Version 5.0*. Wolfram Media, 2003.