Aligators for Arrays ⋆ (tool paper)

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

[1]IST Austria [2] UC Berkeley [3] TU Vienna [4] TUM

Abstract. This paper presents Aligators, a tool for the generation of universally quantified array invariants. Aligators leverages recurrence solving and algebraic techniques to carry out inductive reasoning over array content. The Aligators' loop extraction module allows treatment of multi-path loops by exploiting their commutativity and serializability properties. Our experience in applying Aligators on a collection of loops from open source software projects indicates the applicability of recurrence and algebraic solving techniques for reasoning about arrays.

1 Introduction

Loop invariants build a basis for reasoning about programs and their automatic discovery is a major challenge. Construction of invariant equalities over numeric scalar variables can be efficiently automated using recurrence solving and computer algebra techniques [15]. A combination of quantifier elimination techniques together with a program instrumentation using an auxiliary loop counter variable generalizes the method of [15] to the construction of invariant inequalities [12]. While the methods of [15, 12] are restricted to reasoning over scalars, recurrence solving and algebraic techniques can provide a basis for computing invariants over vector data types, e.g., arrays. For a restricted class of loops that do not contain any branching statements and under non-deterministic treatment of the loop condition, we can compute universally quantified array invariants by using recurrence solving over the loop body [13].

In this paper we eliminate the restrictions of [15, 12, 13], and present the Aligators tool for generating quantified array invariants for loops containing conditional statements that takes loop conditions into account. Quantified loop invariants are inferred by Aligators based on recurrence solving over array indexes. The obtained invariants are derived without using pre- and post conditions; the specification of the loop can be subsequently used further. The invariant inference engine of Aligators relies on two steps (Section 3.2): (i) it applies full power of inductive reasoning provided by recurrence solving over scalar variables and derives the most precise inductive content over scalars, (ii) it combines recurrence solving and algebraic techniques with the theory of uninterpreted functions to derive invariant properties over arrays. Due to the exact computations of the algebraic techniques, Aligators only supports loops with restricted branching control-flow (Section 3.1).

To make Aligators amenable for practical software verification, we built and interfaced Aligators with a loop extraction module (Section 4.1). This module takes as input

⋆ This research was partly supported by the Swiss NSF. The third author is supported by an FWF Hertha Firnberg Research grant (T425-N23).

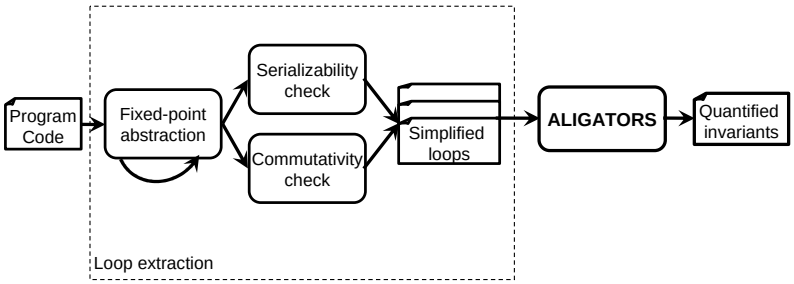

Fig. 1. The overall workflow of Aligators.

a large code, and applies path analysis heuristics to turn loops into the format required by Aligators.

The overall workflow of Aligators is illustrated in Figure 1.

Implementation and Experiments. Aligators is implemented in Mathematica 5.2 [19], whereas the loop extraction module interfaced with Aligators is written in OCaml [5]. Aligators can be downloaded from

`http://mtc.epfl.ch/software-tools/Aligators/`

We have successfully applied Aligators on interesting examples involving arithmetic and array reasoning (Section 4), as well as on a large number of loops extracted from the Netlib repository (`www.netlib.org`). The invariants returned by Aligators were generated in essentially no time, on a machine with a 2.0GHz CPU and 2GB of RAM.

Related Work. Universally quantified invariants are crucial in the verification process of programs with unbounded data structures – see e.g. [7, 14, 9, 18, 10, 16].

In [14, 9, 18] quantified invariants are inferred by deploying predicate abstraction over a set of a priori defined predicates. Alternatively, in [10] quantified invariants are derived by using constraint solving over the unknown coefficients of an a priori fixed invariant template. Unlike these works, Aligators requires no user guidance in providing predicates or templates, but it can be applied to loops with a restricted control-flow.

Based on interval-based abstract interpretation, in [7, 11] quantified invariants are also generated with no user guidance. Unlike these approaches, we do not use abstract interpretation, and apply simple path analysis to translate multi-path loops into simple ones. In [16] invariants with quantifier alternations are obtained using a first-order theorem prover. Contrarily to [16], we combine uninterpreted functions with recurrence solving over array indexes, but can only infer universally quantified invariants.

2 Aligators in Action

To invoke Aligators, one uses the following command.

Command 2.1 : `Aligators[Loop, IniVal` $\rightarrow$ `list of assignments]`

Input: Loop[1] and, *optionally*, a list of assignments specifying initial values of scalars
Output: Loop invariant $\phi_{Var} \wedge \phi_{Arrs}$, where ϕ_{Var} is a scalar invariant and ϕ_{Arrs} is a quantified invariant over arrays

[1] Inputs to Aligators are `while`-loops as in (1).

Benchmark	Program	Quantified invariant

Copy [7]	i:=0;
	while $(i < N)$ **do**
	$B[i]$:=$A[i]$; i:=$i+1$
	end do
		$\forall k.\ 0 \leq k<i \implies (B[k] = A[k] \wedge k < N)$

Copy_Prop [14]	i:=0;
	while $(i < N)$ **do**
	$A[i]$:=0; i:=$i+1$
	end do;
	i:=0;
	while $(i < N)$ **do**
	$B[i]$:=$A[i]$; i:=$i+1$
	end do
		$(\forall k.\ 0 \leq k<i \implies (A[k] = 0 \wedge k < N))$
		$\wedge$
		$(\forall k.\ 0 \leq k<N \implies (B[k] = A[k] \wedge k < N))$

Init [14]	i:=0;
	while $(i < N)$ **do**
	$A[i]$:=0; i:=$i+1$
	end do
		$\forall k.\ 0 \leq k<i \implies (A[k] = 0 \wedge k < N)$

Partition [9]	i:=0; j_1:=0; j_2:=0;
	while $(i < N)$ **do**
	if $(A[i] \geq 0)$
	then $C[j_1]$:=$A[i]$; j_1:=j_1+1
	else $B[j_2]$:=$A[i]$; j_2:=j_2+1 **end if**;
	i:=$i+1$
	end do
		$(\forall k.\ 0 \leq k<j_1 \implies (k < N \wedge C[k] \geq 0))$
		$\wedge$
		$(\forall k.\ 0 \leq k<j_2 \implies (k < N \wedge B[k] < 0))$

Part_Init1 [14]	i:=0; j:=0; **while** $(i < N)$ **do** **if** $(A[i] \geq 0)$ **then** $B[j]$:=i; j:=$j+1$ **end if**; i:=$i+1$ **end do**	$\forall k.\ 0 \leq k<j \implies (B[k] < N \wedge A[B[k]] \geq 0)$

Part_Init2 [7]	i:=0; j:=0;
	while $(i < N)$ **do**
	if $(A[i] = B[i])$
	then $C[j]$:=i; j:=$j+1$ **end if**;
	i:=$i+1$
	end do
		$\forall k.\ 0 \leq k<j \implies (C[k] < N \wedge A[C[k]] = B[C[k]])$

Permutation	i:=0;
	while $(i < N)$ **do**
	$B[\sigma(i)]$:=$A[i]$;; i:=$i+1$
	end do
		$\forall k.\ 0 \leq k<i \implies (B[\sigma(k)] = A[k] \wedge k < N)$

Vararg [14]	i:=0;
	while $(A[i] \neq 0)$ **do**
	i:=$i+1$;
	end do
		$\forall k.\ 0 \leq k<i \implies (A[k] < 0 \vee A[k] > 0)$

Table 1. Aligators results on benchmarks from [7, 14, 9].

EXAMPLE 2.1 Consider the Partition program [9] given in Table 1. The loop copies the non-negative (resp. negative) elements of an array A into an array B (resp. into an array C). The invariant returned by Aligators is listed below.

$$\begin{aligned}
\texttt{Input:}\ & \text{Aligators}[\underline{\textbf{while}}\ (i < N)\ \underline{\textbf{do}} \\
& \qquad \underline{\textbf{if}}\ (A[i] \geq 0)\ \underline{\textbf{then}}\ C[j_1] := A[i];\ j_1 := j_1 + 1 \\
& \qquad \underline{\textbf{else}}\ B[j_2] := A[i];\ j_2 := j_2 + 1\ \underline{\textbf{end if}}; \\
& \qquad i := i + 1\ \underline{\textbf{end do}}, \\
& \qquad \texttt{IniVal} \rightarrow \{i := 0;\ j_1 := 0;\ j_2 := 0\}] \\
\texttt{Output:}\ & (i = j_1 + j_2)\ \wedge\ (N > 0\ \implies\ i \leq N)\ \wedge \\
& (\forall k)\ \big(0 \leq k < j_1\ \implies\ (k < N\ \wedge\ C[k] \geq 0)\big)\ \wedge \\
& (\forall k)\ \big(0 \leq k < j_2\ \implies\ (k < N\ \wedge\ B[k] < 0)\big)
\end{aligned}$$

The above invariant is composed of (i) two quantifier-free linear properties over the scalars i, j_1, and j_2, and (ii) two quantified properties over the arrays B and C. Let us note that the scalar invariant generation method of [15, 12] would fail deriving such a complex invariant, as first-order reasoning over the arrays B and C would be required.

Moreover, due to the presence of conditional statements in the loop body, the technique of [13] could not be either applied for quantified invariant generation.

3 Invariant Generation with Aligators

Aligators offers software support for **a**utomated **i**nvariant **g**eneration by **a**lgebraic **t**echniques **o**ver the **r**ationals and *array***s**.

3.1 Programming Model

Notations. In what follows, $\mathbb{K}$ denotes the domain of values of scalar variables (e.g. integers $\mathbb{Z}$). Throughout this paper, the set of scalar and array variables will respectively be denoted by Var and $Arrs$, where $Arrs = RArrs \cup WArrs$ is a disjoint union of the sets $RArrs$ of *read-only* and $WArrs$ of *write-only* array variables.

Expressions. The language of expressions of Aligators' input contains constants from $\mathbb{K}$, variables from $Var \cup Arrs$, 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. For an array variable A and expression e, we will write $A[e]$ to mean the element of A at position e.

Inputs to Aligators. The syntax of Aligators inputs is given below.

$$\begin{array}{l}\underline{\textbf{while}}\ (b_0)\ \underline{\textbf{do}}\\ \quad \underline{\textbf{if}}\ (b_1)\ \underline{\textbf{then}}\ \alpha_{11}; \ldots; \alpha_{1s_1}\\ \quad \underline{\textbf{else}}\ \ldots\ \underline{\textbf{else if}}\ (b_d)\ \alpha_{d1}; \ldots; \alpha_{ds_d}\ \underline{\textbf{end if}}\\ \underline{\textbf{end do}}\end{array} \tag{1}$$

where $b_0, \ldots, b_d$ are boolean expressions, and α_{kl} are assignment statements over $Var \cup Arrs$. For simplicity, we represent (1) by an equivalent collection of *guarded assignments* [3], as given below.

$$\begin{array}{rcl} G_1 & \rightarrow & \alpha_{11}; \ldots; \alpha_{1s_1}\\ & \cdots & \\ G_d & \rightarrow & \alpha_{d1}; \ldots; \alpha_{ds_d}\end{array}, \tag{2}$$

where formulas G_k are called the *guards* of the guarded assignments. The loop (2) is a *multi-path loop* if $d > 1$. If $d = 1$, the loop (2) is called a *simple-loop*.

Similarly to [13], the following conditions on (2) are imposed:

1. For all $k, l \in \{1, \ldots, d\}$, if $k \neq l$ then the formula $G_k \wedge G_l$ is unsatisfiable.
2. If some α_{ku} updates an array variable $A_u \in WArrs$, and some α_{kv} for $u \neq v$ in the *same guarded assignment* updates an array variable $A_v \in WArrs$, then A_u and A_v are different arrays.
3. The assignments α_{ku}'s have one of the following forms:

   (a) Array assignments: $A[e] := f(Var \cup RArrs)$, (3)

   where $A \in WArrs$, e is an expression over Var, and $f(Var \cup RArrs)$ is an arbitrary expression over $Var \cup RArrs$, but contains no write-arrays.

   (b) Scalar assignments: $x := poly(Var)$, (4)

where $x \in Var$, and $poly(Var)$ is a polynomial expression in $\mathbb{K}[Var]$ over Var such that the total degree of any monomial in x from $poly(Var)$ is exactly 1.

4. If some α_{ku} updates a variable $v \in Var \cup Arrs$, and some α_{lv} with $l \neq k$ updates the same variable v, then α_{ku} is syntactically the same as α_{lv}. That is, variable v is modified in the same way in the kth and lth guarded assignments.

In what follows, a variable $v \in Var \cup Arrs$ satisfying condition 4 above will be called a *commutable* variable. Note that a *commutable* variable is modified in the same way in all guarded assignments of (2). That is, updates to a commutable variable are described by *only one* polynomial expression as given in (4). Reasoning over commutable variables requires thus no case distinctions between various behaviors on different guarded assignments of (2). The guarded assignments[2] of (2) are called *commutable* if their common variables are commutable.

3.2 Invariant Inference with Aligators

Invariant Generation over Scalars. Invariant properties over scalars variables are inferred as presented in [15, 12]. Namely, (i) assignments over scalars are modeled by recurrence equations over *the loop counter* n; (ii) closed forms of variables as functions of n are derived by recurrence solving; (iii) (all) scalar invariant equalities are inferred by eliminating n from the closed forms of variables; and (iv) scalar invariant inequalities over commutable variable are obtained using quantifier elimination over n.
Invariant Generation over Arrays. In our process of quantified invariant generation, (i) we first rewrite (1) into (2), (ii) generate quantified invariants over non-commutable array variables for each simple-loop given by a guarded assignment of (2), and (iii) take conjunction of the quantified invariants to obtain a quantified invariant of (1).
(i) Input loops (1) to Aligators satisfy[3] the restrictions 1-4 given on page 4. Hence, guards are disjoint, and branches are commutable. Internally, Aligators rewrites an input loop (1) into (2) (as illustrated in Example 3.2), and proceeds with generating invariants for the simple-loops of (2).
(ii) Aligators next infers quantified invariants for the following simple-loop of (2):

$$G \rightarrow \alpha_1; \ldots; \alpha_s. \tag{5}$$

W.l.o.g., we assume that (5) contains only one array update, as below:

$$A[i] := f(Var \cup RArrs), \text{ where } i \in Var \text{ and } A \in WArrs \text{ are non-commutable.} \tag{6}$$

Based on the programming model given on page 4, since variables i and A are non-commutable, changes to i and A can only happen on the guarded assignment (5). Recurrence solving thus can be applied to derive exact closed form representation of i and A.

Let us denote by $n \geq 0$ the loop counter. We write $x^{(n)}$ to mean the value of $x \in Var$ at iteration n. As array updates satisfy the restrictions of Section 3.1, we write $A[x^{(n)}]$ instead of $A^{(n)}[x^{(n)}]$ to speak about the value of the xth element of A at iteration n of the loop.

[2] respectively, conditional branches of (1)

[3] Aligators checks whether an input loop satisfies the restrictions of Section 3.1. If this is not the case, Aligators returns an error messages about the violated restriction.

Based on (5) and (6), at iteration n of (5) the following property holds:

$$G^{(n)} \wedge A[i^{(n+1)}] = f(\,Var^{(n+1)} \cup RArrs), \tag{7}$$

where $Var^{(n+1)} = \{x^{(n+1)} \mid x \in Var\}$, and $G^{(n)}$ is the formula obtained by substituting variables x with $x^{(n)}$ in G. Formula (7) holds at any iteration k upto n. Hence:

$$(\forall k) 0 \leq k \leq n \implies G^{(k)} \wedge A[i^{(k+1)}] = f(\,Var^{(k+1)} \cup RArrs) \tag{8}$$

We further eliminate n from (8), as follows. (i) If the closed forms of loop variables is a linear system in n, linear algebra methods are used to express n as a linear function $p(Var) \in \mathbb{K}[Var]$. (ii) Otherwise, Gröbner basis computation [1] is used to compute n as a polynomial function $p(Var) \in \mathbb{K}[Var]$. We thus obtain the quantified invariant:

$$(\forall k) 0 \leq k \leq p(\,Var) \implies G^{(k)} \wedge A[i^{(k+1)}] = f(\,Var^{(k+1)} \cup RArrs) \tag{9}$$

EXAMPLE 3.1 Consider $i < N \wedge A[i] > 0 \rightarrow C[j_1] := A[i]; j_1 := j_1 + 1; i := i + 1$. Let $n \geq 0$ denote its loop counter. Following (8), we have:

$$i^{(n)} < N \wedge A[i^{(n)}] > 0 \ \wedge \ C[j_1^{(n+1)} - 1] = A[i^{(n+1)} - 1],$$

where $j_1^{(n+1)} = j_1^{(n)} + 1$ and $i^{(n+1)} = i^{(n)} + 1$. Using recurrence solving and replacing the (final) value $j_1^{(n+1)}$ by j_1, we obtain $n = j_1 - 1 - j_1^{(0)}$. It thus follows:

$$(\forall k) 0 \leq k \leq j_1 - 1 - j_1^{(0)} \implies k < N \wedge A[k] > 0 \ \wedge \ C[k] = A[k].$$

(iii) To turn (9) into a quantified invariant of (2), we finally make sure that:
- when eliminating n from (8), n is computed as a polynomial function over only *non-commutable scalar variables*;
- formula (9) is simplified to contain only *non-commutable scalar and array variables*.

The quantified invariant of (1) is given by the conjunction of the quantified invariants without commutable variables of each simple-loop of (2).

EXAMPLE 3.2 The main steps for quantified invariant generations with Aligators are illustrated on the Partition program of Table 1. The initial values of both i and j_1 are 0.

(i) Guarded assignments:	(ii) Quantified invariants	
	with commutable variables:	without commutable variables:
$i < N \wedge A[i] \geq 0$ $\rightarrow C[j_1] := A[i];$ $\quad j_1 := j_1 + 1; i := i + 1$	$(\forall k) 0 \leq k < j_1 \implies$ $k < N \wedge A[k] \geq 0 \wedge A[k] = C[k]$	$(\forall k) 0 \leq k < j_1 \implies$ $k < N \wedge C[k] \geq 0$
$i < N \wedge A[i] < 0$ $\rightarrow B[j_2] := A[i];$ $\quad j_2 := j_2 + 1; i := i + 1$	$(\forall k) 0 \leq k < j_2 \implies$ $k < N \wedge A[k] < 0 \wedge B[k] = A[k]$	$(\forall k) 0 \leq k < j_2 \implies$ $k < N \wedge B[k] < 0$

The final invariant of the Partition program is given in Example 2.1.

4 Experimental Results

We report on our experimental results with Aligators, obtained on a machine with 2.0GHz CPU and 2GB of RAM.

Aligators on benchmark examples. We ran Aligators on a collection of benchmark examples taken from [7, 14, 9]. Our results are summarized in Table 1.

4.1 Loop Extraction for Aligators

Aligators supports modular reasoning by analyzing one loop at a time. To run Aligators on large programs with more than one loop, we built and interfaced Aligators with a loop extraction module written in OCaml [5]. This module extracts and translates loops from large C programs into the shape of (1). For doing so, the loop extraction module takes one complex C program as input, and outputs one or more loops that can be further fed into Aligators. To this end, three main techniques[4] are applied: (i) *fixed point abstraction*, (ii) *serializability check*, and (iii) *commutativity check*.

Fixed Point Abstraction. Given a loop, the goal of this step is to find the largest sequence of loop assignments satisfying the restrictions given in Section 3.1. To this end, we abstract away all loop assignments that either violate the input requirements of Aligators, or depend on variables whose assignments do not satisfy Section 3.1. For abstracting assignment away, we check if the assignment is side-effect free. If yes, the assignment can be soundly approximated by an uninterpreted function. We recursively repeat the previous steps, until a fixed point is reached. As a result, we either obtain an empty loop body which means that the loop does not fit into our programming model, or a sequence of assignments that is a sound approximation of the original loop.

Serializability check. Let us denote by ρ_i the guarded assignment $G_i \rightarrow \alpha_{i1}; \ldots; \alpha_{is_i}$ from (2). The role of the serialization check is to turn, if possible, a multi-path loop (2) into an equivalent collection of simple-loops.

Using regular expression-like notation, we check whether the set of all possible loop executions is included in the set $\mathcal{L} = \{\rho_{i_1}^*; \ldots; \rho_{i_d}^* \mid (\rho_{i_1}, \ldots, \rho_{i_d})$ is a permutation of $(\rho_1, \ldots, \rho_d)\}$. Solving this query involves reasoning in the combined quantifier-free theory of integers, arrays, and uninterpreted functions, which can be efficiently solved using SMT solvers. To this end, we make use of the SMT solver Z3 [2].

Note that the serializability check requires the construction of all permutations of length d over $(\rho_1, \ldots, \rho_d)$. Our experimental results show that the number d of loop paths in practice is small. For the programs we analyzed, more than 80% of the loops have less than 3 branches, and 5% of the loops have been simplified with the serializability check.

EXAMPLE 4.1 Consider the loop given below.

i := 1;
for(i := 1; $i \leq n$; i++) **do**
 if ($i \leq a$) **then** $A[i]$:= $B[i]$ **else** $A[i]$:= 0 **end if**
end do

The result of serializability check on the above loop is a collection of two simple-loops, as follows.

$i := 1$
while($i \leq a$) **do**
 $A[i]$:= $B[i]$; $i := i + 1$
end do

i := $a + 1$
while($i \leq n$) **do**
$A[i]$:= 0; $i := i + 1$
end do

[4] The loop extraction module performs other small steps as well – e.g. for-loops are rewritten in their while-loop format.

Program	LoC	Loops	Analyzable Loops	Invariants	Array Invariants
Gzip	8607	201	100	62	39
Bzip2	6698	260	106	75	35
OggEnc	58413	680	464	185	11

Table 2. The first column is the name of the program analyzed. The second and third columns give respectively the lines of code and the total number of loops. The fourth column contains the number of loops that fall into our programming model. The fifth column presents the number of loops for which invariants were inferred by Aligators, whereas the last column gives the number of those invariants which describe one or more array content.

Commutativity check. The goal of this step is to collapse multi-path loops (2) with only commutable variables into a simple-loop. To this end, we check whether the assignments of variables are syntactically the same in all branches.

Aligator on large programs. We ran Aligators on two file-compression tools, Gzip [6] and Bzip2 [17], and on the MP3 encoder OggEnc [4]. The results are presented in Table 2. The array invariants inferred by Aligators express copy/permutation/shift/initialization relations between two arrays. Our results suggests that Aligators generated invariants for over 25% of the extracted loops. The second column of Table 2 shows that roughly half of the loops fits into our programming model. The obtained invariants were generated by Aligators in essentially no time; Aligators can analyze and reason about 50 loops per second.

5 Conclusion

We describe Aligators, an automated tool for quantified invariant generation for programs over arrays. Our tool requires no user guidance, it applies recurrence solving to arrays, and has been successfully applied to generate invariants for loops extracted from large, non-trivial programs. Further work includes integrating control-flow refinement techniques, such as [8], into Aligators, and using our tool in conjunction with other approaches, such as [10, 16], to invariant generation.

References

1. 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.
2. L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proc. of TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008.
3. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
4. M. Smith et al. The OggEnc Home Page. http://www.xiph.org/, 1994.
5. X. Leroy et al. *The Objective Caml system - release 3.11*. INRIA, 2008.
6. J. Gailly and M. Adler. The Gzip Home Page. http://www.gzip.org/, 1991.
7. D. Gopan, T. W. Reps, and S. Sagiv. A Framework for Numeric Analysis of Array Operations. In *Proc. of POPL*, pages 338–350, 2005.
8. S. Gulwani, S. Jain, and E. Koskinen. Control-flow Refinement and Progress Invariants for Bound Analysis. In *Proc. of PLDI*, pages 375–385, 2009.
9. S. Gulwani and A. Tiwari. Combining Abstract Interpreters. In *Proc. of PLDI*, pages 376–386, 2006.
10. A. Gupta and A. Rybalchenko. InvGen: An Efficient Invariant Generator. In *Proc. of CAV*, pages 634–640, 2009.

11. N. Halbwachs and M. Péron. Discovering Properties about Arrays in Simple Programs. In *Proc. of PLDI*, pages 339–348, 2008.
12. T. A. Henzinger, T. Hottelier, and L. Kovács. Valigator: A Verification Tool with Bound and Invariant Generation. In *Proc. of LPAR*, pages 333–342, 2008.
13. T. A. Henzinger, T. Hottelier, L. Kovács, and A. Voronkov. Invariant and Type Inference for Matrices. In *Proc. of VMCAI*, pages 163–179, 2010.
14. R. Jhala and K. L. McMillan. Array Abstractions from Proofs. In *Proc. of CAV*, pages 193–206, 2007.
15. L. Kovács. Reasoning Algebraically About P-Solvable Loops. In *Proc. of TACAS*, volume 4963 of *LNCS*, pages 249–264, 2008.
16. L. Kovács and A. Voronkov. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *Proc. of FASE*, pages 470–485, 2009.
17. J. Seward. The Bzip2 Home Page. http://www.bzip.org/, 1996.
18. S. Srivastava and S. Gulwani. Program Verification using Templates over Predicate Abstraction. In *Proc. of PLDI*, pages 223–234, 2009.
19. S. Wolfram. *The Mathematica Book. Version 5.0*. Wolfram Media, 2003.