A simple abstraction of arrays and maps by program translation

David Monniaux, Francesco Alberti

A simple abstraction of arrays and maps by program translation

David Monniaux*
Univ. Grenoble Alpes, VERIMAG, F-38000 Grenoble, France
CNRS, VERIMAG, F-38000 Grenoble, France

Francesco Alberti†
Fondazione Centro San Raffaele, Milan, Italy

June 11, 2015

Abstract

We present an approach for the static analysis of programs handling arrays, with a Galois connection between the semantics of the array program and semantics of purely scalar operations. The simplest way to implement it is by automatic, syntactic transformation of the array program into a scalar program followed analysis of the scalar program with any static analysis technique (abstract interpretation, acceleration, predicate abstraction,...). The scalars invariants thus obtained are translated back onto the original program as universally quantified array invariants. We illustrate our approach on a variety of examples, leading to the "Dutch flag" algorithm.

1 Introduction

Static analysis aims at automatically discovering program properties. Traditionally, it has focused on dataflow properties (e.g. "can this pointer be null?"), then on numerical properties (e.g. "$2x+y \leq 45$ at every iteration of this loop"). When it comes to programs operating over *arrays*, special challenges arise. For instance, the ASTRÉE static analyzer,[1] based on abstract interpretation and commercially used in the avionics, automotive and other industries, supports arrays simplistically: it either "smashes" all cells in a single array into a single

*The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement nr. 306595 "STATOR"

†This work has been carried out while the author was affiliated to the Università della Svizzera Italiana and supported by the Swiss National Science Foundation under grant no. P1TIP2_152261.

[1] [17, 7, 8] http://www.astree.ens.fr http://absint.de/astree/

abstract value, or expands an array of n cells into n variables; in many cases it is necessary to fully unroll loops operating over an array in order to prove the desired property[2].

In general, however, analyzing arrays programs entails exhibiting inductive loop invariants with universal quantification over array indices. Neither smashing nor expansioncan prove, in general, that a simple initialization loop truly does work:

Listing 1: Simple array initialization

```
int t[n]; for(int i=0; i<n; i++) t[i] = 0;
```

To derive the postcondition $\forall k.0 \leq k < n \rightarrow t[k] = 0$, one uses the loop invariant (in the Floyd-Hoare sense) $0 \leq i \leq n \wedge \forall k.0 \leq k < i \rightarrow t[k] = 0$. The $0 \leq i \leq n$ part (or generalizations, e.g., filling the upper triangular part of a matrix) can be automatically inferred by many existing numeric analysis techniques. In contrast, the $\forall k.0 \leq k < i \rightarrow t[k] = 0$ part is trickier and is the focus of this article.

Contribution We propose a generic method for analyzing array programs, which can be implemented i) as a normal abstract domain ii) or by translating the program with arrays into a scalar program (a program without arrays), analyzing this program by any method producing invariants (back-end), and then recovering the array properties. Its precision depends on the back-end analysis. Our method has tunable precision and is formalized by Galois connections [14] and, contrary to most others, is not guided by a target property (here $\forall k.0 \leq k < n \rightarrow t[k] = 0$), though it can take advantage of it. It can therefore be used to supply information to the end-user "what does this program do?" as opposed to be useful only for proving properties. We demonstrate the flexibility of our approach on examples, using the acceleration procedure Flata, the abstract interpreter ConcurInterproc and CPAChecker as back-ends.

We also show a form of *completeness*: for any loop-free program, the precision of the analysis can be chosen so that it is exact with respect to universally quantified array properties (§4.3).

Our approach also applies to general maps *keys* → *values*, though certain optimizations apply only to totally ordered index types.

Contents Section 2 introduces our approach on one example. Section 3 discusses the Galois connections, and section 4 gives the formal definition of our transformation algorithm and associated correctness and partial completeness proofs. Section 5 discusses the use of various backends on more examples. We finish with related work and conclusion.

[2] Possible since Astrée targets safety-critical embedded systems where array sizes are typically fixed at system design and dynamic memory allocation is prohibited.

2 Example: the Sentinel

Our program transformation consists in i) a replacement of reads and writes parameterized by a number of distinguished indices, formalized in section 4 ii) optionally, some "focusing" on a subset of index values iii) for certain backends (ConcurInterproc), the addition of observer variables implementing a form of partitioning.

Listing 2: A "sentinel value" marks the penultimate array cell

```
const int N=1000; int i = 0, t[N];
initialize(N, t); t[N−2] = −1;
while (t[i]>=0) i++;
```

Obviously to us humans, this program cannot crash with an array access out of bounds, and the final value of i is, at most, 998 (its value depends on how the " initialize " procedure works). How can we obtain this result automatically?

Let x be a symbolic constant in $\{0, \dots, N-1\}$. We abstract array t by the single cell t[x], represented by variable tx: reads and writes at position x in t translates to reads and writes to variable tx and reads and writes at other positions are ignored. Program 2 is thus abstracted as:[3]

```
const int N=1000, x = random(); assume(x >= 0 && x < N);
int i = 0, tx = random(); if (N−2 == x) tx=−1;
while(1) { int read = random(); if (i == x) { read = tx; }
           if (read < 0) { break; } i = i+1;                }
```

Flata [9, 28] can compute an exact input/output relation of this program (to demonstrate generality, we left N unfixed and replaced N−2 by a parameter p; we thus use a precondition $0 \leq x < N \land 0 \leq p < N$):

$$\begin{aligned}
&(p = x \land i \leq x-1 \land i \geq 0 \land N \geq x+1) \lor (i = x \land i \geq 0 \land N \geq p+1 \land i \leq p-1) \lor \\
&(x \geq p+1 \land i \leq x-1 \land i \geq 0 \land N \geq x+1 \land p \geq 0) \lor (i = x \land i \leq N-1 \land i \geq p+1 \land p \geq 0) \lor \\
&(i \geq x+1 \land N \geq p+1 \land i \leq N-1 \land x \leq p-1 \land x \geq 0) \lor \\
&(i \leq x-1 \land i \geq 0 \land N \geq p+1 \land x \leq p-1) \lor \\
&(i = x \land i = p \land i \geq 0 \land i \leq N-1) \lor (x \geq p+1 \land i \geq x+1 \land i \leq N-1 \land p \geq 0) \qquad (F)
\end{aligned}$$

Note that our abstraction is valid *whatever the value of* x. This means that (i, p, N) should be a solution of $N > 0 \land \forall x \ (0 \leq x < N \Rightarrow F)$. One can check that this quantified formula entails $i \leq p$.

Arguably, we have done too much work: the only cell in the array whose content matters much is at index p (N−2 in the original program). Running Flata with $x = p$ yields a postcondition implying $i \leq p$. Again, this is sound, because *any* choice of x yields a valid postcondition on (i, p).

3 Galois connections

We shall now see that, for any choice of indices, there is a Galois connection $\underset{\alpha}{\overset{\gamma}{\leftrightarrows}}$ [12] between the concrete (the set of possible values of the vector of variables of

[3] We have left out, for the sake of brevity, tests for array accesses out of bounds.

the original program) and the abstract set of states (the set of possible values of the vector of variables in the transformed program). In general, this Galois connection is not onto: there are abstract elements $x^\natural$ that include "spurious" states, and which may be reduced to a strictly smaller $\alpha \circ \gamma(x^\natural)$.

If A and B are sets, $A \to B$ denotes the set of total functions from A to B, and $\mathcal{P}(A)$ the set of parts of A. If A is finite, $A \to B$ denotes the set of *arrays* indexed by A; specifically, if A is $\{1,\dots,l_1\} \times \dots \times \{1,\dots,l_d\}$ then $A \to B$ denotes the d-dimensional arrays of size $(l_1,\dots,l_d)$. $f[x]$ denotes the application $f(x)$ where f is a program array or map.

Our constructions easily generalize to arbitrary combinations of numbers of arrays and numbers of indices; let us see a few common cases.

3.1 Single index

Applied with a single index, our map abstraction is classical [15, §2.1].

Definition Let $f \in A \to B$, we abstract it by its graph $\alpha_1(f) = \{(a, f[a]) \mid a \in A\}$; e.g., a constant array $\{1,\dots,n\} \to \mathbb{Z}$ with value 42 is abstracted as $\{(i, 42) \mid 1 \leq i \leq n\}$.

We lift α_1 (while keeping the same notation) to a function from $\mathcal{P}(A \to B)$ to $\mathcal{P}(A \times B)$: for $F^\flat \subseteq A \to B$, $\alpha_1(F^\flat) = \bigcup_{f \in F} \alpha_1(f)$, otherwise said

$$\alpha_1(F^\flat) = \Big\{(a, f[a]) \mid a \in A, f \in F^\flat\Big\} \tag{1}$$

Let $F^\natural \subseteq A \times B$. Then we define its concretization $\gamma_1(F^\natural)$:

$$\gamma_1(F^\natural) = \big\{f \in A \to B \mid \forall a \in A\ (a, f[a]) \in F^\natural\big\} \tag{2}$$

It is easy to see that $(\mathcal{P}(A \to B), \subseteq) \mathrel{\underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}}} \mathcal{P}(A \times B)$ is a Galois connection.

Non-surjectivity and reduction Remark that α_1 is not onto (if $|A| > 1$ and $|B| > 0$): there exist multiple $F^\natural$ such that $\gamma_1(F^\natural) = \emptyset$, namely all those such that $\exists a \in A \forall b \in B\ (a, b) \notin F^\natural$. For instance, if considering arrays of two integer elements ($A = \{0, 1\}$, $B = \mathbb{Z}$), then $F^\natural = \{(1, 0)\}$ yields $\gamma_1(F^\natural) = \emptyset$: there is no way to fill the array at index 0.

Let us now see the practical implication. Assume that the program has a single array in $A \to B$ and a vector of scalar variables ranging in S, then the memory state is an element of $X^\flat \triangleq S \times (A \to B)$. The scalar variables are combined into our abstraction as follows:

$$\mathcal{P}(S \times (A \to B)) \cong S \to \mathcal{P}(A \to B) \mathrel{\underset{\alpha_1^S}{\overset{\gamma_1^S}{\leftrightarrows}}} S \to \mathcal{P}(A \times B) \cong \mathcal{P}(S \times A \times B) \triangleq X^\natural, \tag{3}$$

where α_1^S and γ_1^S lift α_1 and γ_1 pointwise. Let $s \in S$. While the absence of any $(s, a, b) \in x^\natural$ ($x^\natural \in X^\natural$) indicates that there is no $(s, f) \in \gamma_1^S(x^\natural)$, that is, scalar state s is unreachable, the converse is *not* true. Consider a single integer scalar variable s and an array a of length 2, and $x^\natural = \{(0,0,1),(1,0,0),(1,1,2)\}$, representing the triples $(s, i, a[i])$. It would seem that $s = 0$ is reachable, but it is not, because there is no way to fill the array at position 1: there is no element in $x^\natural$ of the form $(0, 1, b)$.

A *reduction* is a function $\rho : X^\natural \to X^\natural$ such that $\gamma \circ \rho = \gamma$ and $\rho(x^\natural) \subseteq x^\natural$ for all $x^\natural$. The strongest reduction ρ_{opt} (the minimum for the pointwise ordering induced by $\subseteq$) is $\alpha \circ \gamma$. In the above, $\rho_{\text{opt}}(x^\natural) = \{(1,0,0),(1,1,2)\}$; intuitively, the strongest reduction discards all superfluous elements from the abstract value.

Class of formulas Assume now that the vector of scalar variables $s_1, \dots, s_m$ lies within $S = \mathbb{Z}^m$, the index a lies in $\{1, \dots, l_1\} \times \dots \times \{1, \dots, l_D\}$, and the values $f[a]$ also lie in $\mathbb{Z}$. Consider a formula ψ of the form

$$\forall a_1, \dots, a_d \ \phi(s_1, \dots, s_m, a_1, \dots, a_d, f[a_1, \dots, a_d]) \tag{4}$$

where ϕ is a first-order arithmetic formula (say, Presburger).

Then, $f \models \psi$ if and only if $\alpha_1^S(f) \subseteq \{((s_1, \dots, s_m), (a_1, \dots, a_d), b) \mid \phi(s_1, , \dots, , s_m, a_1, , \dots, , a_d, b)\}$. The sets of program states expressible by formulas of form 4 thus map through the Galois connection to a sub-lattice of $\mathcal{P}\left(\mathbb{Z}^m \times \mathbb{Z}^d \times \mathbb{Z}\right)$. This construction may be generalized to any theory or combination of theories over the sorts used for scalar variables, array indices, and array contents.

Checking that an invariant $\gamma_1^S(G)$ entails ψ, when the set G is defined by a formula Γ, just amounts to checking that $\Gamma \wedge \neg\psi$ is unsatisfiable.

3.2 Several indices, one per array

The above settings can be extended to several arrays. Let $f, g \in A \to B$, we abstract them by the product of their graphs $\alpha_1(f, g) = \{(a, f[a], a', g[a']) \mid a, a' \in A\}$, $\gamma_1(x^\natural) = \{(f, g) \in (A \to B)^2 \mid \forall a, a' \in A \ (a, f[a], a', g[a']) \in x^\natural\}$. This abstraction can express properties of the form

$$\forall a_1, \dots, a_d, a'_1, \dots, a'_d \ \phi(s_1, \dots, s_m, a_1, \dots, a_d, f[a_1, \dots, a_d], a'_1, \dots, a'_d, g[a'_1, \dots, a'_d])$$

As an example, the property that up to index k, monodimensional array f of length n has been copied into array g can be expressed as $\forall a, a' \in \{1, \dots, n\} \ a < k \wedge a = a' \Rightarrow f[a] = g[a']$ within that class.

3.3 Dual indices, same array

Definition Let $f \in A \to B$, pose $\alpha_2(f) = \{(a, f[a], a', f[a']) \mid a, a' \in A\}$ and lift it to a function from $\mathcal{P}(A \to B)$ to $\mathcal{P}\left((A \times B)^2\right)$. Let $F^\natural \subseteq (A \times B)^2$. Then we define its concretization $\gamma_2(F^\natural)$:

$$\gamma_2(F^\natural) = \left\{f \in A \to B \mid \forall a, a' \in A \ (a, f[a], a', f[a']) \in F^\natural\right\} \tag{5}$$

It is easy to see that $(\mathcal{P}(A \to B), \subseteq) \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \mathcal{P}(A \times B)$ is a Galois connection.

If A is totally ordered, it seems a waste to include both $(a, f[a], a', f[a'])$ and $(a', f[a'], a, f[a])$ in the abstraction for $a < a'$. We thus define $\alpha_{2<}(f) =$

$\{(a, f[a], a', f[a']) \mid a < a' \in A\}$ and $\gamma_{2<}(x^\natural) = \{f \in A \to B \mid \forall a, a' \in A\ , a < a' \Rightarrow (a, f[a], a', f[a']) \in x^\natural\}$.

Non-surjectivity Remark, again, that α_2 is not onto. Consider an array of integers of length 3, that is, a function $f : \{1, 2, 3\} \to \mathbb{Z}$. An analysis computes its abstraction as $x^\natural = \{(1, 0, 2, 0), (1, 0, 3, 0), (2, 0, 3, 0), (1, 0, 3, 1)\}$; recall that each element of that set purports to denote $(a, f[a], a', f[a'])$ for $a < a'$. At first sight, it seems that $f(3) = 1$ is possible, as witnessed by the last element. Yet, there is then no way to fill $a[2]$: there is no x such that $(2, x, 3, 1) \in x^\natural$. This last element is therefore superfluous, and we can conclude that $\forall x\ f[x] = 0$. (See § 5.5 for a real-life example.)

If $x^\natural$ is defined by a first-order formula ($x^\natural = \{(a, b, a', b') \mid \phi(a, b, a', b')\}$), then this reduction (removing all a', b' such that for some $a < a'$ there is no way to fill $f[a]$) is obtained as: $\forall a \exists b\ a < a' \Rightarrow \phi(a, b, a', b')$.

Class of formulas Assume now that the vector of scalar variables $s_1, \ldots, s_m$ lies within $S = \mathbb{Z}^m$, the indices $a < a'$ lie in $\{1, \ldots, n\}$, and the values $f[a], f[a']$ also lie in $\mathbb{Z}$. Consider a formula ψ of the form $\forall a, a'\ a < a' \Rightarrow \phi(s_1, \ldots, s_m, a, f[a], a', f[a'])$ where ϕ is a first-order arithmetic formula (say, Presburger). For instance, one may express *sortedness*: $\forall a, a'\ a < a' \Rightarrow f[a] \leq f[a']$.

Then, $f \models \psi$ if and only if $\alpha^S_{2<}(f) \in \{((s_1, \ldots, s_m), a, b, a', b') \mid \phi(s_1, \ldots, s_m, a, b, a', b')\}$. The sets of program states expressible by formulas of the form $\forall a, a'\ a < a' \Rightarrow \phi(s_1, \ldots, s_m, a, f[a], a', f[a'])$ thus map through the Galois connection to a sub-lattice of $\mathcal{P}\left(\mathbb{Z}^m \times (\mathbb{Z} \times \mathbb{Z})^2\right)$.

4 Abstraction of program semantics

Our analysis may be implemented by a syntactic transformation of array operations into purely scalar operations. In this section, for each operation (read, write) we describe the transformed operation and demonstrate the correctness of the transformation. We then discuss precision.

Without loss of generality, we consider only elementary reads and writes (`r= f[i ];` and `f[i]=r;` with `i` a variable). More complex constructs, e.g. `f[e]=r;` with `e` an expression, can always be decomposed into a sequence of scalar operations and elementary read and writes, using temporary variables.

4.1 Transformation and Correctness

Reading from the array Consider a program state composed of (s, r, i, f) where $r \in B$, $i \in A$ are scalars, $s \in S$ is the rest of the state, and $f \in A \to B$. Consider the instruction `r=f[i ];`, its semantics is:

$$(s, r, i, f) \xrightarrow{\texttt{r=f[i];}} (s, f(i), i, f) \tag{6}$$

We wish to abstract it by the program fragment:

Listing 3: Read from array

```
r = random(); if (i==a) { r=b; }
```

Lemma 1. *The forward and backward semantics of Program 3 abstract the forward and backwards semantics of r=f[i]; by the (α_1^S, γ_1^S) Galois connection.*

More generally, a read with several indexes $a_1, a_2, \ldots$ is abstracted by

```
r=random();if (i==a1) assume(r==b1); if (i==a2) assume(r==b2); ...
```

The same lemma and proof carry to that setting.

Writing to the array Consider the instruction f[i]=r;, its semantics is:

$$(s, r, i, f) \xrightarrow{\texttt{f[i]=r;}} (s, r, i, f[i \mapsto r]) \tag{7}$$

We wish to abstract it by the program fragment:

Listing 4: Write to array

```
if (i==a) { b=r; }
```

Lemma 2. *The forward and backward semantics of Program 4 abstract the forward and backwards semantics of f[i]=r; by the (α_1^S, γ_1^S) Galois connection.*

The same carries over to writing to an array with several indices, abstracted as:

Listing 5: Write to array, multiple indexes

```
if (i==a1) { b1=r; }  if (i==a2) { b2=r; } ...
```

Operations on scalars Consider a program state composed of (s, f) where $f \in A \to B$ is an array and $s \in S$ is the rest of the state. Consider a scalar instruction $s \xrightarrow{P} s'$ and thus $(s, f) \xrightarrow{P^\flat} (s', f)$. We abstract P as: $(s, a, b) \xrightarrow{P^\natural} (s', a, b)$ if $s \to Ps'$. Essentially, operations on scalars are abstracted by themselves. The following result generalizes immediately to (α_2, γ_2) etc.

Lemma 3. *The forward and backward semantics of $\xrightarrow{P^\natural}$ abstract those of $\xrightarrow{P^\flat}$ by the (α_1^S, γ_1^S) Galois connection.*

4.2 Precision loss

"Forgetting" the value of a scalar variable v corresponds to $(s, v, f) \to (s, f)$. This scalar operation may be correctly abstracted, as in Lemma 3, by $(s, v, a, b) \to (s, a, b)$. Surprisingly, applying this operation not only forgets the value of v, it may also enlarge the set of represented f.

Example: $x^\natural = \{(0, v, a, v) \mid a \in A \wedge v \in B\}$ abstracts by (α_1^S, γ_1^S) the set of triples $(0, v, f)$ where f is a constant function of value v. Forgetting v yields the set of pairs $(0, f)$ where f is a constant function. Applying $(s, v, a, b) \to (s, a, b)$ to $x^\natural$ yields $y^\natural = \{(0, a, v) \mid A \in A \wedge v \in B\}$, which concretizes to the set $\{(0, f) \mid f \in A \to B\}$. We have completely lost the "constantness" property.

4.3 Relative completeness

We now consider the problem of *completeness* of this abstraction, assuming that the back-end analysis is perfectly precise (thus *relative completeness*).

Our analysis is incomplete in general. Consider the following program:

Listing 6: Fill with zero, test zero

```
int t[N];  for(int i=0; i<N; i++) t[i]=0;
for(int i=0; i<N; i++) if (t[i]!=0) break;
```

In the second loop, the **break** statement is never reached and thus at the end of the loop, $i = N$. Yet, if we distinguish $n < N$ different indices $i_1, \dots, i_n$, we cannot prove that this statement is never reached: for there will exist $i \in \{0, \dots, N-1\} \setminus \{i_1, \dots, i_n\}$ such that `t[i]` returns, in the abstracted program, an arbitrary value and thus the **break** statement is considered possibly reachable.

In contrast, when the program is loop-free, the abstraction is exact with respect to the scalar variables, provided the number of indices used for the abstraction is at least the number of array accesses:

Theorem 1. *Consider a loop-free array program P with arrays $a_1, \dots, a_d$ such that the number of accesses to these arrays are respectively $\alpha_1, \dots, \alpha_d$. By abstracting these arrays with, respectively, $n_1, \dots, n_d$ indices such that $n_i \geq \alpha_i$ for all i, we obtain a Galois connection $\underset{\alpha}{\overset{\gamma}{\leftrightarrows}}$ such that $\pi_S \circ \gamma \circ P^\natural \circ \alpha = \pi_S \circ P^\flat$ where π_S is the projection of the state to the scalar variables.*

This completeness results extends to universally quantified array properties $\forall i_1, \dots\ P(i_1, \dots) \rightarrow Q(a_1[i_1], \dots)$: one appends to the original program (assuming $i_1, \dots, i_n$ are fresh, nondeterministically initialized):

```
assume((P(i1,...)); assert(Q(i1,...));
```

5 More examples

5.1 Matrix initialization

Listing 7: Initialization of $m \times n$ matrix a with value v

```
void array_init_2d(int m, int n, int a[m][n], int v) {
  for(int i = 0; i < m; i++) {
    for(int j = 0; j < n; j++) a[i][j] = v;           } }
```

Again, we consider cell $a[x,y]$, where $0 \leq x < m$ and $0 \leq y < n$, and disregard all other cells. One should not convert this procedure into a single control-flow graph, because the resulting numerical transition system does not have the "flat" structure expected by FLATA [10]. Instead, one must encode the inner loop as a separate procedure:

```
void array_init_2d(int m, int n, int a, int v, int x, int y) {
  assume(x >= 0 && x < m);
  assume(y >= 0 && y < n);
  for(int i=0; i<m; i++)  innerloop(n, a, v, x, y, i);        }
void inner_loop(int n, int a, int v, int x, int y, int i) {
  for(int j=0; j<n; j++)  if (x==i && y==j) a = v;         }
```

Flata then computes the exact input-output relation of inner_loop, and finally the exact input-output relation of array_init_2d:

$$(x=0\wedge m=1\wedge a'=v\wedge y\geq 0\wedge n\geq y{+}1)\vee(a'=v\wedge x\geq 1\wedge y\geq 0\wedge m\geq x{+}1\wedge n\geq y{+}1)\vee$$
$$(n=1\wedge x=0\wedge y=0\wedge a'=v\wedge m\geq 2)\vee(x=0\wedge a'=v\wedge y\geq 0\wedge m\geq 2\wedge n\geq 2\wedge n\geq y{+}1)$$

Each disjunct implies $a' = v$, i.e., the final value of $a[x, y]$ is v. Again, because (x, y) are symbolic constants with no assumption except that they are valid indices for a, this proves that all cells contain v. Assuming $0 \leq x < m \wedge 0 \leq y < n$ this formula may indeed be simplified automatically into $a' = v$.[4]

5.2 Slice initialization

Listing 8: Initialize $a[\mathit{low} \dots \mathit{high} - 1]$ to v

```
void slice_init(int n, int a[n], int low, int high, int v) {
  for(int i=low; i<high; i++) a[i] = v;                      }
```

Again, we transform the program using a single index:

```
  for(int i=low; i<high; i++) if (x == i) a = v;
```

Flata produces as postcondition (assuming $0 \leq x < n \wedge 0 \leq \mathit{low} \leq \mathit{high} \leq n$):

$$\begin{aligned}(\mathit{high} = \mathit{low} \wedge a' = a \wedge \mathit{high} \geq 0 \wedge n \geq \mathit{high} \wedge n \geq x + 1 \wedge x \geq 0)\vee\\ (a' = v \wedge \mathit{low} \leq x \wedge n \geq \mathit{high} \wedge \mathit{high} \geq x + 1 \wedge \mathit{low} \geq 0)\vee\\ (a' = a \wedge n \geq \mathit{high} \wedge \mathit{high} \geq \mathit{low} + 1 \wedge \mathit{low} \geq x + 1 \wedge x \geq 0)\vee\\ (a' = a \wedge \mathit{high} \leq x \wedge n \geq x + 1 \wedge \mathit{high} \geq \mathit{low} + 1 \wedge \mathit{low} \geq 0)\end{aligned} \quad (8)$$

Again, under the assumptions $0 \leq x < n$ and $0 \leq \mathit{low} \leq \mathit{high} \leq n$, this formula is equivalent to: $((\mathit{low} \leq x < \mathit{high}) \to a' = v) \wedge (\neg(\mathit{low} \leq x < \mathit{high}) \to a' = a)$. Thus by quantification, the expected outcome:

$$(\forall x \in [\mathit{low}, \mathit{high})\ a'[x] = v) \wedge (\forall x \notin [\mathit{low}, \mathit{high}) \to a'[x] = a[x]) \quad (9)$$

5.3 Array copy

Listing 9: Copy array a into array b

```
void array_copy(int n, int a[n], int b[n]) {
  for(int i=0; i<n; i++) b[i] = a[i];       }
```

Take a single cell $a[x]$ in a and a single cell $b[y]$ in b; after transformation:

[4] We implemented a simplification algorithm for quantifier-free Presburger arithmetic inspired by [38] so as to understand the output of Flata and ConcurInterproc.

```
  int n, a, b, x, y, tmp;
  assume(0 <= x && x < n && 0 <= y && y < n);
  for(int i=0; i<n; i++) { if (x==i) tmp=a; if (y==i) b=tmp; }
```

Flata Flata yields: $(y \geq x+1 \wedge n \geq y+2 \wedge x \geq 0) \vee (n = y+1 \wedge y \geq x+1 \wedge x \geq 0) \vee (n = x+1 \wedge y \geq 0 \wedge y \leq x-1) \vee (y \geq 0 \wedge y \leq x-1 \wedge n \geq x+2) \vee (y = x \wedge b' = a \wedge n \geq x+2 \wedge x \geq 0) \vee (y = x \wedge b' = a \wedge n = x+1 \wedge x \geq 0)$. Assuming $0 \leq x < n \wedge 0 \leq y < n$, this is equivalent to $x = y \rightarrow a = b$. Thus by quantification, $\forall x, y. x = y \rightarrow a[x] = b[y]$, simplifiable into $\boldsymbol{\forall x. a[x] = b[x]}$.

Software model checking Many software model checkers, including CPAChecker[5], do not handle universally quantified array properties; yet we can use them as back-end analyses! We translate the target property (here $\forall x. 0 \leq x < n \rightarrow a[x] = b[x]$) into a precondition $x = y$ and an assertion on the postcondition $a = b$. CPAChecker then proves the property.[6]

```
int main() {
  int n, a, b, x, y;
  if (0 <= x && x < n && 0 <= y && y < n && x==y) {
    for(int i=0; i<n; i++) {
      int tmp; if (x==i) tmp=a; if (y==i) b=tmp; }
    assert(a==b);                                    } }
```

5.4 In-place array reversal

Listing 10: Array reversal

```
void array_reverse_inplace(int n, contents t[n]) {
  int i=0, j=n-1;
  while(i < j) {
    contents tmp1 = t[i], tmp2 = t[j];
    t[i] = tmp2; t[j] = tmp1;      i++; j--;     } }
```

For this program, we need to distinguish the initial values in the array from the values during the computation (which finally yield the final values). We use three indices $0 \leq x < n$, $0 \leq y \leq z < n$: a is the initial value of $t[x]$, b the current value of $t[y]$, c the current value of $t[z]$.

For each read, we check if the index of the read is equal to y (respectively, z) and return b (respectively, c) if this is the case. If the index is equal to both y and z, it is sound to return either b or c; we chose to return b. For each write, we test if the index is equal to y, in which case we write to b, and equal to z, in which case we write to c. If it is equal to both y and z, we write to both b and c.

Listing 11: Array reversal, transformed

```
  contents a, b, c;
  int x, y, z, i=0, j=n-1;
```

[5] http://cpachecker.sosy-lab.org/

[6] `scripts/cpa.sh -predicateAnalysis` after preprocessing with `assert.h`

```
if (y == x) b = a;  if (z == x) c = a;
while(i < j) { contents tmp1, tmp2;
  if (i == y) tmp1 = b;  else if (i == z) tmp1 = c;
  if (j == y) tmp2 = b;  else if (j == z) tmp2 = c;
  if (i == y) b = tmp2;  if (i == z) c = tmp2;
  if (j == y) b = tmp1;  if (j == z) c = tmp1;      i++; j--;
    }
```

Flata FLATA takes 480 s[7] to process this program, and outputs an input-output relation ϕ in disjunctive normal form with 292 disjuncts (not reprinted). The output formula is very complicated, with explicit enumeration of many particular cases; the reason for the slowness and the size of the output formula seems to be that FLATA explicitly enumerates many cases up to saturation, with no attempt at intermediate simplifications. We shall now explain what this formula entails.

Let U be $0 \leq x, y, z < n \land y + z = n - 1$. Let $U_<$ be $U \land y < z \land z = x \land y + z = n - 1$, then $\phi \land U_<$ is equivalent to $a = b \land U_<$. This means that under the precondition $U_<$, Prog. 11 has exact postcondition $a = b$. By universal quantification, this means that $\forall x, y, z.U_< \to t[x] = t'[y]$, where t is the input array to Prog. 10 and t' the output. This formula may be simplified into $\forall x.0 \leq x \land 2x \leq n-2 \to t[x] = t'[n-1-x]$; We can obtain similar formulas for the cases $y > z$ and $y = z$. The three cases can can be summarized into

$$\boldsymbol{\forall x.0 \leq x < n \to t[x] = t'[n-1-x]} \tag{10}$$

Flata, focused The above execution time and the complexity of the resulting formula seem excessive, if all that matters is when $(x = y \lor x = z) \land y + z = n - 1$. Indeed, some easy static analysis (by FLATA or another tool) shows that the array accesses within the loop are done at indices i and j that satisfy $0 \leq i \leq j < n$ and $i + j = n - 1$. Such a pre-analysis suggests to target the main analysis to two positions $t[y]$ and $t[z]$ in the current array, satisfying $0 \leq y \leq z < n$ and $y + z = n - 1$. The only positions $a[x]$ that matter in the original array are those that can be read precisely, that is, $x = y$ and $x = z$.

We therefore re-run the analysis with precondition U: $(0 \leq y \leq z < n \land y + z = n - 1 \land x = y)$. FLATA runs for 6 s and outputs a formula with 8 disjuncts, with $a = c$ in all disjuncts. We thus have proved that $\forall x, y, z.U \to t[x] = t'[z]$, which can be simplified into $\boldsymbol{\forall z.2z \geq n - 1 \land z < n \to t'[z] = t[n - 1 - z]}$.

We may also run with the precondition, $(0 \leq y \leq z < n \land y + z = n - 1 \land x = z)$ and get the remainder of the cases to conclude as in Formula 10.

To summarize, when the exact analysis of the transformed program (that is, an exact analysis in the back-end) is too costly, one may choose to *focus* the analysis by restricting the range of the indices $(x, y, z, \dots)$ to some area U considered to be "meaningful", for instance obtained by pre-analysis of the relationships between the indices of the array accesses in the program. This is sound, since the quantification in the resulting formula is over the indices satisfying U. Thus, a bad choice for U may only result in a sound, but uninteresting

[7] All timings using one core of a 2.4 GHz Intel ® Core$^{\text{TM}}$ i3 running 32-bit Linux.

invariant (the worst case is to take an unsatisfiable U: we then obtain a formula talking about an empty set of positions in the arrays, thus a tautology).

ConcurInterproc, focused INTERPROC[8] applies classical abstract interpretation (Kleene iteration accelerated with widenings, with possible narrowing iterations) over a variety of numerical abstract domains provided by the APRON [30] library[9] (intervals, "octagons" [37], convex polyhedra [23, 13]...).

CONCURINTERPROC[10] extends it to concurrency (which we will not use here) and partitioning of the state space according to enumerated types, including Booleans. In a nutshell, while INTERPROC assigns a single abstract element (product of intervals, octagon, polyhedron) to each program location, CONCURINTERPROC attaches 2^n abstract elements, where n is the number of Booleans (or,more generally, one per concrete instantiation of the enumerated variables). In order to achieve this at reasonable cost, the BDDAPRON library uses a compact representation, where identical abstract elements are shared and the associated set of concrete instantiations is represented by a binary decision diagram.

Program 11 contains no Boolean variable (or of any other enumerated type), thus directly applying CONCURINTERPROC over it will yield one convex polyhedron at the end; yet we need to express a disjunction of such polyhedra (e.g. there is the case where $x = y$, and the case where $x \neq y$, which may be subdivided into $x < y$ and $y < z$). Furthermore, inside the loop one would have to distinguish $i < y$, $i = y$, $i > y$. This is where, in other analysis of array properties by abstract interpretation [22, 24, 39, 40, 16] one introduces "slices" or "segments" of programs, often according to syntactic criteria. In our case, we wish to distinguish certain locations in the array (or combinations of several locations, as here with three indices x, y, z) according to more semantic criteria.

Our solution is to introduce *observer* variables, which are written to but never read and whose final value is discarded, but which will guide the analysis and the partitioning performed. Here, we choose to have one flag variable per access, initially set to "false", and set to "true" when the access has taken place. As previously, we use a precondition $y + z = n - 1 \wedge x = z$.

Listing 12: Array reversal, transformed and instrumented

```
contents a, b, c;
int x, y, z;
bool y0,z0,y1,z1,y2,z2,y3,z3,y4,z4;
x0=y0=y1=z1=y2=z2=y3=z3=y4=z4=false;
int i=0, j=n-1;
assume(y+z == n-1); assume(x==z);
if (y == x) { b = a; y0 = true; } if (z == x) { c = a; z0 = true; }
while(i < j) {
  contents tmp1, tmp2;
  if (i == y) {tmp1 = b; y1 = true;} else if (i == z) {tmp1 = c; z1 = true;}
  if (j == y) {tmp2 = b; y2 = true;} else if (j == z) {tmp2 = c; z2 = true;}
  if (i == y) {b = tmp2; y3 = true;} if (i == z) {c = tmp2; z3 = true;}
  if (j == y) {b = tmp1; y4 = true;} if (j == z) {c = tmp1; z4 = true;}
```

[8] http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/

[9] http://apron.cri.ensmp.fr/library/

[10] http://pop-art.inrialpes.fr/interproc/concurinterprocweb.cgi

```
  i++; j--;                                                    }
```

ConcurInterproc, within 0.16 s, concludes that $a = b$.

5.5 Dutch national flag

Quicksort is a divide-and-conquer sorting algorithm: pick a *pivot*, swap array cells until the array is divided into two areas: elements less than the pivot, and elements greater than or equal to it; then recurse in both areas. An improvement, in case many elements may be identical, is to swap the array into three areas: elements less than the pivot, equal to it, and greater than it, and recurse in the "less" and "greater" areas. This three-way partition is equivalent to the "Dutch national flag problem" [19, ch. 14], of swapping pebbles of colors red, white and blue (corresponding to "less", "equal" and "greater") into three segments.

Listing 13: Dutch flag[11]

```
void threeWayPartition(int data[], int size, int low, int high)
   {
 int p = -1, q = size;
 for (int i = 0; i < q;) {
  if(data[i] < low) {swap(&data[i], &data[++p]); ++i;}
  else if(data[i]>=high) {swap(&data[i], &data[--q]);} else ++i
      ;
}}
```

We transform this program with two indices $0 \leq x < y < n$ (remark that this is valid only if $n \geq 2$) with associated values datax and datay, and instrument it with Boolean observer variables: for each read or write access to an index i, we keep a Boolean recording the value of predicate $x \leq i$ and one for $x \geq i$ (respectively for y). The values in the array are encoded as pebble colors LOW, MIDDLE, HIGH.

ConcurInterproc computes a postcondition within 1 min. The resulting formula ϕ has 52 cases; we will not print it here. We check that $\phi \wedge x \leq p \rightarrow \mathit{datax} = \text{BLUE}$, meaning that finally, $\boldsymbol{\forall x. 0 \leq x \leq p \rightarrow t[x] = \mathbf{BLUE}}$ Similarly, $\phi \wedge y \geq q \rightarrow \mathit{datay} = \text{RED}$, thus $\boldsymbol{\forall y. q \leq y < n \rightarrow t[y] = \mathbf{RED}}$. We would expect as well that $\forall x. p < x < q \rightarrow t[x] = \text{WHITE}$. Yet, this does not immediately follow from ϕ: $\phi \wedge p < y < q \wedge \mathit{datay} = \text{RED}$ is satisfiable! Could there be red cells in the supposedly white area?

Note that ϕ, for fixed values of n, p, q, encodes quadruples $(x, \mathit{datax}, y, \mathit{datay})$, which encompass all possible values of $(x, t[x], y, t[y])$ for $x < y$. In particular, for $t[y] = \text{RED}$ to be possible for given n, p, q, one must have suitable $t[x]$ for all $x < y$, such that $(x, t[x], y, \text{RED})$ satisfies ϕ for the same n, p, q. In other words, to have a cell $t[y] = \text{RED}$ one must be able to find values $t[x]$ for all cells to the left of it. We check that, indeed, $p < y < q \wedge \mathit{datay} \neq \text{WHITE} \wedge (\forall x. 0 \leq x < y \rightarrow \phi)$ is unsatisfiable,[12] meaning that $\forall y. (p < y < q \wedge y > 0) \rightarrow t[y] = \text{WHITE}$.

[11] Courtesy of Wikipedia

[12] From Presburger arithmetic, a decidable theory.

Furthermore, $\phi \land x = 0 \land x < q \land datax \neq \text{WHITE}$ has no solution. We can thus conclude $\boldsymbol{\forall y. p < y < q \rightarrow t[y] = \textbf{WHITE}}$.

Thus, we encountered a case of "spurious" solutions in the abstract element, due to the fact that the abstraction is not onto and that certain abstract elements can be reduced to a smaller element with the same concretization; which was achieved through quantification (see subsection 3.3). This reduction can thus be performed through some form of *quantifier elimination.*

6 Related work

Acceleration For certain classes of loops, it is possible to compute exactly the transitive closure τ^+ of the relation τ encoding the semantics of the loop, within a decidable class. Acceleration for arrays has been studied by Bozga et al. [11], who obtain the transitive closure in the form of a *counter automaton.* The translation from counter automaton to array properties expressed in first-order logic then requires an abstraction step, resulting in a loss of precision. Alberti et al. [4, 1] proposed a template-based solution. Certain classes of τ's admit a definable acceleration in Presburger arithmetic augmented with free function symbols, at the price of nested quantifiers. The $\exists^*\forall^*$ fragment of this theory is undecidable [25]; thus again abstraction is needed to apply this technique in practice. Yet, there are cases where exact acceleration is possible [3]. Contrary to these approaches, i) ours does not put restrictions on the shape of the loop (and the program in general) ii) we perform the tunable abstraction first, with the rest of the analysis being delegated to a back-end (which can possibly use exact acceleration on scalar programs [9]).

Abstract interpretation Various array abstractions [22, 24, 39, 40, 16] distinguish *slices* or *segments*, whose contents is then abstracted by another abstract domain. Depending on the approach, relationships between several slices may or may not be expressed, and the partitioning may be syntactic or based on some pre-analysis. To our best knowledge, none of these approaches work on multidimensional arrays or on maps, contrary to ours. One major difference between these approaches and ours is that ours separates the analysis, both in theory and implementation, into an abstraction that maps array programs to scalar programs and an analysis for the scalar programs, while theirs are more "monolithic". Even though they are parametric in abstract domains for values and possibly indexes, they must be used inside an abstract interpreter based on Kleene iterations with widening. In contrast, ours can use any back-end analysis for scalar programs, including exact acceleration, abstract interpretation with Kleene iterations, policy iteration, and even, if a target property is supplied, predicate abstraction (see CEGAR below).

Cox et al. [18] do not target array programs per se, but programs in highly dynamic object-oriented languages such as Javascript, where an object is a map from fields to values and the set of possible field names is not fixed. Dillig et al. [20] overcome the dichotomy of strong vs weak updates with *liquid updates*. Their approach is monolithic and cannot express properties such as sortedness.

Predicate abstraction and CEGAR *Predicate abstraction* starts from the control structure of a program and incrementally refines it by splitting control states according to predicates chosen by the user [21] or, commonly, obtained by counterexample-guided abstraction refinement (CEGAR). From an abstract counterexample trace not corresponding to a concrete counterexample, they refine the model using local predicates constituting a step-by-step proof that this abstract trace does not match any concrete trace. The hope is that this proof generalizes to more counterexample traces and that the predicates eventually converge to define an inductive invariant. The predicates are obtained from *Craig interpolants* [33, 36, 35] extracted from the proof of unsatisfiability produced by a *satisfiability modulo theory* (SMT) solver. The difficulty here is to generate Craig interpolants that tend to generalize to inductive invariants, on *quantified* formulas involving arrays [34]. We are interested in predicates such as $\forall 0 \leq k < i,\ t[k] = 0$, which generalizes to an inductive invariant on Program 1, as opposed to, say, $t[0] = 0 \wedge t[1] = 0$, which is equivalent for $i = 2$ but does not generalize to arbitrary i. In order to achieve practical scalability, some work restrict themselves to the inference of array predicates to certain forms, e.g. *range predicates* [31]. Others tune the interpolating procedure towards the generation of better interpolants [2, 5]. A major difference between our approach and those based on CEGAR is that we do not require a “target” property to prove, which is necessary for having counterexamples, though we can use one if needed. If such a property is provided, our approach can use as a back-end a CEGAR system limited to scalar variables.

Theorem proving and SMT-based approaches The generation of invariants for programs with arrays has been also studied using automated theorem proving [26, 27]; this approach is generally limited by the fact that theory reasoning (e.g. arithmetic) and superposition-based deductive reasoning (on which the Vampire first-order theorem prover is based [32]) are not yet efficiently integrated. As opposed to [6], we do not rely on quantifier-instantiation procedures.

Quantification Flanagan et al. [21] also use Skolem constants that they quantify universally after analysis steps. As opposed to us, they require the user to specify the predicates on which the program will be abstracted.

Abstraction of sets of maps Our approach generalizes a classical abstraction of sets of maps [15, §2.1]. Jeannet et al. [29] considered the problem of abstracting sets of functions of signature $D_1 \rightarrow D_2$, assuming a *finite* abstract domain A_1 of cardinality n abstracting subsets of D_1 and an abstract domain A_2 abstracting subsets of D_2^n. In contrast, we do not make any cardinality assumption.

Partitioning Rival et al. [41] introduced partitioning according to an abstraction of the history of the computation. Our approach using observer variables for using ConcurInterproc (subsection 5.4) is akin to considering a finite abstraction of the trace of read/writes into a given array.

7 Conclusion and Future Work

We have shown that a number of properties of array programs can be proved by abstracting the array a using a few symbolic cells $a[x], a[y], \dots$ by automatically translating the program into a scalar program, running a static analyzer over the scalar program and translating back the invariant for the original program. In some cases, a form of quantifier elimination is used over the resulting formulas.

Our approach is not specific to arrays, and can be applied to any map structure $X \to Y$ (e.g. hash tables and other container classes). A possible future extension is multiset properties, a multiset being map $X \to \mathbb{N}$.

The main weakness of our approach is the need for a rather precise back-end analysis (for the scalar program obtained by translation). Our experiments highlighted some inefficiencies in e.g. Flata and ConcurInterproc: in the former, many paths can be enumerated and complicated formulas generated even though a much simpler equivalent form exists; in the latter, polyhedra that are only slightly different (say, one constraint is different) are handled wholly separately. This gives immediate directions for research for improving exact acceleration, as in Flata, or disjunctions of polyhedra, as in ConcurInterproc. Another difficulty, if using ConcurInterproc or other tools focusing on convex sets of integer vectors, is the need to use observer variables and/or an auxiliary pre-analysis to "focus" the main analysis.

We stress again that we obtained our results using unmodified versions of very different back-end analyzers (ConcurInterproc, Flata, CPAChecker), which testifies to the flexibility of our approach. Performance and precision improvements can be expected by modifying the back-end analyzers (e.g. precision could be improved by performing reduction steps during the analysis, rather than after the computation of the invariants).

References

[1] F. Alberti, S. Ghilardi, and N. Sharygina. "Decision Procedures for Flat Array Properties". In: *TACAS*. 2014, pp. 15–30.

[2] F. Alberti et al. "An extension of lazy abstraction with interpolation for programs with arrays". In: *Formal Methods in Systems Design* 45.1 (2014), pp. 63–109.

[3] Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina. "Decision Procedures for Flat Array Properties". In: *J. Autom. Reasoning* 54.4 (2015), pp. 327–352.

[4] Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina. "Definability of Accelerated Relations in a Theory of Arrays and Its Applications". In: *FroCoS*. 2013, pp. 23–39. DOI: 10.1007/978-3-642-40885-4_3.

[5] Francesco Alberti and David Monniaux. "Polyhedra to the rescue of array interpolants". In: *Symposium on applied computing (Software Verification & Testing)*. ACM, 2015.

[6] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. "On Solving Universally Quantified Horn Clauses". In: *SAS*. 2013, pp. 105–125.

[7] Blanchet et al. "A Static Analyzer for Large Safety-Critical Software". In: *PLDI*. ACM. 2003, pp. 196–207. DOI: 10.1145/781131.781153. arXiv: cs/0701193.

[8] Bruno Blanchet et al. "Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software". In: *The Essence of Computation: Complexity, Analysis, Transformation*. LNCS 2566. Springer, 2002, pp. 85–108. DOI: 10.1007/3-540-36377-7_5.

[9] M. Bozga, R. Iosif, and F. Konecný. "Fast Acceleration of Ultimately Periodic Relations". In: *CAV*. 2010, pp. 227–242.

[10] M. Bozga, R. Iosif, and Y. Lakhnech. "Flat parametric counter automata". In: *Fundamenta Informaticae* 91 (2009), pp. 275–303.

[11] M. Bozga et al. "Automatic Verification of Integer Array Programs". In: *CAV*. 2009, pp. 157–172.

[12] P. Cousot and R. Cousot. "Abstract Interpretation Frameworks". In: *J. Log. Comput.* 2.4 (1992), pp. 511–547.

[13] P. Cousot and N. Halbwachs. "Automatic Discovery of Linear Restraints Among Variables of a Program". In: *POPL*. 1978, pp. 84–96.

[14] Patrick Cousot and Radhia Cousot. "Abstract Interpretation Frameworks". In: *J. Log. Comput.* 2.4 (1992), pp. 511–547. DOI: 10.1093/logcom/2.4.511.

[15] Patrick Cousot and Radhia Cousot. "Invited Talk: Higher Order Abstract Interpretation. Application to Comportment Analysis Generalizing Strictness, Termination, Projection, and PER Analysis". In: *IEEE International Conference on Computer Languages*. IEEE, 1994, pp. 95–112.

[16] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. "A parametric segmentation functor for fully automatic and scalable array content analysis". In: *POPL*. ACM, 2011, pp. 105–118. DOI: 10.1145/1926385.1926399.

[17] Patrick Cousot et al. "Why does Astrée scale up?" In: *Formal Methods in System Design* 35.3 (2009), pp. 229–264. DOI: 10.1007/s10703-009-0089-6.

[18] Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. "Automatic Analysis of Open Objects in Dynamic Language Programs". In: *SAS*. Vol. 8723. LNCS. Springer, 2014, pp. 134–150. ISBN: 978-3-319-10935-0. DOI: 10.1007/978-3-319-10936-7_9.

[19] Edsger Wybe Dijkstra. *A discipline of programming.* Prentice-Hall, 1976. ISBN: 0-13-215871-X.

[20] Işıl Dillig, Thomas Dillig, and Alex Aiken. "Fluid Updates: Beyond Strong vs. Weak Updates". In: *ESOP*. 2010, pp. 246–266.

[21] C. Flanagan and S. Qadeer. "Predicate abstraction for software verification". In: *POPL*. 2002, pp. 191–202.

[22] D. Gopan, T.W. Reps, and S. Sagiv. "A framework for numeric analysis of array operations". In: *POPL*. 2005, pp. 338–350.

[23] Nicolas Halbwachs. "Détermination automatique de relations linéaires vérifiées par les variables d'un programme". PhD thesis. Univ. Grenoble, Mar. 1979. URL: https://tel.archives-ouvertes.fr/tel-00288805.

[24] Nicolas Halbwachs and Mathias Péron. "Discovering properties about arrays in simple programs". In: *PLDI*. ACM, 2008, pp. 339–348. DOI: 10.1145/1375581.1375623.

[25] J.Y. Halpern. "Presburger arithmetic with unary predicates is Π_1^1 complete". In: *The Journal of Symbolic Logic* 56.2 (1991), pp. 637–642.

[26] K. Hoder, L. Kovács, and A. Voronkov. "Invariant Generation in Vampire". In: *TACAS*. 2011, pp. 60–64.

[27] Krystof Hoder, Laura Kovács, and Andrei Voronkov. "Interpolation and Symbol Elimination in Vampire". In: *IJCAR*. Vol. 6173. LNCS. Springer, 2010, pp. 188–195. ISBN: 978-3-642-14202-4. DOI: 10.1007/978-3-642-14203-1_16.

[28] Hossein Hojjat et al. "A Verification Toolkit for Numerical Transition Systems". In: *FM*. 2012, pp. 247–251. DOI: 10.1007/978-3-642-32759-9_21.

[29] Bertrand Jeannet, Denis Gopan, and Thomas W. Reps. "A Relational Abstraction for Functions". In: *SAS*. Vol. 3672. LNCS. Springer, 2005, pp. 186–202. ISBN: 3-540-28584-9. DOI: 10.1007/11547662_14.

[30] Bertrand Jeannet and Antoine Miné. "Apron: A Library of Numerical Abstract Domains for Static Analysis". In: *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*. Vol. 5643. LNCS. Springer, 2009, pp. 661–667. ISBN: 978-3-642-02657-7. DOI: 10.1007/978-3-642-02658-4_52.

[31] R. Jhala and K.L. McMillan. "Array Abstractions from Proofs". In: *CAV*. 2007, pp. 193–206.

[32] Laura Kovács and Andrei Voronkov. "First-Order Theorem Proving and Vampire". In: *CAV*. 2013, pp. 1–35. DOI: 10.1007/978-3-642-39799-8_1.

[33] Kenneth L. McMillan. "Applications of Craig Interpolation to Model Checking". In: *ICATPN*. Vol. 3536. LNCS. Springer, 2005, pp. 15–16. ISBN: 3-540-26301-2. DOI: 10.1007/11494744_2.

[34] Kenneth L. McMillan. "Quantified Invariant Generation Using an Interpolating Saturation Prover". In: *TACAS*. Vol. 4963. LNCS. Springer, 2008, pp. 413–427. ISBN: 978-3-540-78799-0. DOI: 10.1007/978-3-540-78800-3_31.

[35] K.L. McMillan. “Interpolants from Z3 proofs.” In: *FMCAD*. 2011, pp. 19–27.

[36] K.L. McMillan. “Lazy Abstraction with Interpolants”. In: *CAV*. 2006, pp. 123–136.

[37] Antoine Miné. “The octagon abstract domain”. In: *Higher-Order and Symbolic Computation* 19.1 (2006), pp. 31–100. DOI: 10.1007/s10990-006-8609-1.

[38] David Monniaux. “A Quantifier Elimination Algorithm for Linear Real Arithmetic”. In: *LPAR*. LNCS 5330. Springer, 2008, pp. 243–257. ISBN: 978-3-540-89439-1. DOI: 10.1007/978-3-540-89439-1_18. arXiv: 0803.1575.

[39] Mathias Péron. “Contributions to the Static Analysis of Programs Handling Arrays”. Theses. Université de Grenoble, Sept. 2010. URL: https://tel.archives-ouvertes.fr/tel-0062369

[40] Valentin Perrelle. “Analyse statique de programmes manipulant des tableaux”. Theses. Université de Grenoble, Feb. 2013. URL: https://tel.archives-ouvertes.fr/tel-00973892.

[41] Xavier Rival and Laurent Mauborgne. “The trace partitioning abstract domain”. In: *ACM Trans. Program. Lang. Syst.* 29.5 (2007). DOI: 10.1145/1275497.1275501.

A Proofs

Lemma 1. *The forward and backward semantics of Program 3 abstract the forward and backwards semantics of* `r=f[i];` *by the* (α_1^S, γ_1^S) *Galois connection.*

Proof. Consider an abstraction $x^\natural \subseteq S \times B \times A \times (A \times B)$ of (s, r, i, f): $\forall a \in A\ (s, r, i, a, f[a]) \in x^\natural$. The image of the set $x^\natural$ by that program is $y^\natural = \{(s, r', i, a, b) \mid r' \in B \wedge i \neq a \wedge (s, r, i, a, b) \in x^\natural\} \cup \{(s, b, i, i, b) \mid (s, r, i, i, b) \in x^\natural\}$. It is clear that $(s, f(i), i, f) \in \gamma(y^\natural)$, otherwise said $\forall a \in A\ (s, f(i), i, a, f[a]) \in y^\natural$.

The pre-image of the set $x^\natural$ by that program is $z^\natural = \{(s, r, i, a, b) \mid r \in B \wedge i \neq a \wedge (s, r', i, a, b) \in x^\natural\} \cup \{(s, r, i, i, b) \mid r \in B \wedge (s, b, i, i, b) \in x^\natural\}$. Assume $(s, r', i, f) \in \gamma(x^\natural)$ and $(s, r, i, f) \xrightarrow{\texttt{r:=f[i]}} (s, r', i, f)$; then • either $r' \neq f(i)$: then there is no such (s, r, i, f), thus any such $(s, r, i, f) \in \gamma(z^\natural)$Th • either $r' = f(i)$, then any (s, r, i, f) fits; let us now prove $(s, r, i, f) \in \gamma(z^\natural)$: let $a \in A$, then either $i = a$ and $(s, r, i, a, f[a]) \in z^\natural$ (second disjunct), or $i \neq a$ and $(s, r, i, a, f[a]) \in z^\natural$ (first disjunct). □

Lemma 2. *The forward and backward semantics of Program 4 abstract the forward and backwards semantics of* `f[i]=r;` *by the* (α_1^S, γ_1^S) *Galois connection.*

Proof. Consider an abstraction $x^\natural \subseteq S \times B \times A \times (A \times B)$ of (s, r, i, f): $\forall a \in A\ (s, r, i, a, f[a]) \in x^\natural$. The image of the set $x^\natural$ by that program is $y^\natural = \{(s, r, i, a, b) \mid i \neq a \wedge (s, r, i, a, b) \in x^\natural\} \cup \{(s, r, i, i, r) \mid (s, r, i, a, b) \in x^\natural\}$. Let us prove that $(s, r, i, f[i \mapsto r]) \in \gamma(y^\natural)$. Let $a \in A$. If $a \neq i$, then $(s, r, i, a, f[i \mapsto r](a)) = (s, r, i, a, f(a)) \in y^\natural$ (first disjunct); if $a = i$, then $(s, r, i, a, f[i \mapsto r](a)) = (s, r, i, i, r) \in y^\natural$ (second disjunct).

The pre-image of the set $x^\natural$ by that program is $z^\natural = \{(s,r,i,i,b' \mid b' \in B \wedge (s,r,i,i,b) \in x^\natural\} \cup \{(s,r,i,a,b) \mid i \neq a \wedge (s,r,i,a,b) \in x^\natural\}$. Assume $(s,r,i,f') \in \gamma(x^\natural)$ and $(s,r,i,f) \xrightarrow{\texttt{r:=f[i]}} (s,r,i,f')$; let us prove $(s,r,i,f) \in \gamma(z^\natural)$. Let $a \in A$. If $a = i$, then $(s,r,i,i,f(i)) \in z^\natural$ (first disjunct) If $a \neq i$, then $(s,r,i,a,f(a)) = (s,r,i,a,f'(a)) \in z^\natural$ (second disjunct). □

Theorem 1. *Consider a loop-free array program P with arrays $a_1,\ldots,a_d$ such that the number of accesses to these arrays are respectively $\alpha_1,\ldots,\alpha_d$. By abstracting these arrays with, respectively, $n_1,\ldots,n_d$ indices such that $n_i \geq \alpha_i$ for all i, we obtain a Galois connection $\underset{\alpha}{\overset{\gamma}{\leftrightarrows}}$ such that $\pi_S \circ \gamma \circ P^\natural \circ \alpha = \pi_S \circ P^\flat$ where π_S is the projection of the state to the scalar variables.*

Proof. Consider an execution trace T in P, and record the indices $\xi_{i,j}$ of the j-th (numbered syntactically) access to the i-th array. Consider now the program P' obtained by abstracting P according to α_i indices for each array a_i, i.e. each read $r := a_i[e]$ is transformed into

```
r = random();
if (e==x_{i,1}) { assume(r==b_{i,1}); } if (e==x_{i,2}) { assume(r==b_{i,2}); }
...
```

and each write $a_i[e] := w$ as

```
if (e==x_{i,1}) { b_{i,1} = w; } if (e==x_{i,2}) { b_{i,2} = w; } ...
```

Now replay T in P', with the same initial values, the same external and nondeterministic choices, and $x_{i,j} = \xi_{i,j}$. Then, for any array access in the execution of P', at least one of the tests is taken (the program does not fall into the case where none of the selected indices match the index for the read/write instruction). In the case of a read $r := a_i[e]$, the value read in P' is then the same as the one read in P. Then, the execution of P' faithfully mimics that of P. The final values for the execution of T in P' are thus the same as those in P, which proves the statement. □