An encoding of array verification problems into array-free Horn clauses

Anonymous
University of Somewhere
Someone@Somewhere.edu

Abstract

Automatically verifying safety properties of programs is hard, and it is even harder if the program acts upon arrays or other forms of maps. Many approaches exist for verifying programs operating upon Boolean and integer values (e.g. abstract interpretation, counterexample-guided abstraction refinement using interpolants), but transposing them to array properties has been fraught with difficulties.

In contrast to most preceding approaches, we do not introduce a new abstract domain or a new interpolation procedure for arrays. Instead, we generate an abstraction as a scalar problem and feed it to a preexisting solver. The intuition is that if there is a proof of safety of the program, it is likely that it can be expressed by elementary steps between properties involving only a small (tunable) number N of cells from the array.

Our transformed problem is expressed using Horn clauses over scalar variables, a common format with clear and unambiguous logical semantics, for which there exist several solvers. In contrast, solvers directly operating over Horn clauses with arrays are still very immature.

An important characteristic of our encoding is that it creates a nonlinear Horn problem, with tree unfoldings, contrary to the linear problems obtained by flatly encoding the control-graph structure. Our encoding thus cannot be expressed by encoding into another control-flow graph problem, and truly leverages the Horn clause format.

Experiments with our prototype VAPHOR show that this approach can prove automatically the functional correctness of several classical examples of the literature, including *selection sort*, *bubble sort*, *insertion sort*, as well as examples from previous articles on array analysis.

1. Introduction

Formal program verification, that is, proving that a given program behaves correctly according to specification in all circumstances, is difficult. Except for very restricted classes of programs and properties, it is an undecidable question. Yet, a variety of approaches have been developed over the last 40 years for automated or semi-automated verification, some of which have had industrial impact.

In this article, we consider programs operating over arrays, or, more generally, *maps* from an index type to a value type (in the following, we shall use "array" and "map" interchangeably). Such programs contain read (e.g. $v := a[i]$) and write ($a[i] := v$) operations over arrays, as well as "scalar" operations.[1]

Universally quantified properties Very often, desirable properties over arrays are universally quantified; e.g. sortedness may be expressed as $\forall k_1, k_2\ k_1 < k_2 \implies a[k_1] \leq a[k_2]$. However, formulas with universal quantification and linear arithmetic over integers and at least one predicate symbol (a predicate being a function to the Booleans) are so expressive that one can define the execution of a Turing machine as a model to such a formula, whence this class is undecidable [16]. Some decidable subclasses have however been identified [8]. There is therefore no general algorithm for checking that such invariants hold, let alone inferring them. Yet, there have been several approaches proposed to infer such invariants (more on this in Section 7).

In this article, we propose a method for inferring such universally quantified invariants, given a specification on the output of the program. Because of the undecidable nature of this problem, this approach may fail to terminate in the general case. Experiments however show that our approach can successfully and automatically verify nontrivial properties (e.g. the output from selection sort is sorted and is a permutation of the input).

[1] In the following, we shall lump as "scalar" operations all operations not involving the array under consideration, e.g. $i := i + 1$. Any data types (integers, strings etc.) are supported if supported by the back-end solver.

Our approach is based on the idea that if there is a proof of safety of an array-manipulating program, it is likely that there exists a proof that can be expressed with simple steps over properties relating only a small number N of (parametric) array cells. For instance, all the sorting algorithms we tried can be proved correct with $N = 2$, and simple array manipulations (copying, reversing...) with $N = 1$.

We convert the verification problem to Horn clauses, a popular format for program verification problems [30] supported by a number of tools. Most conversions to Horn clauses map variables and operations from the program to variables of the same type and the same operations in the Horn clause problem:[2] an integer is mapped to an integer, an array to an array, etc. If some data types are not supported by the back-end analysis, the variables of these types may be discarded, at the expense of precision — thus if the back-end analysis does not support arrays, array reads are abstracted as nondeterministic choices, array writes are discarded. In contrast, our approach abstracts programs much less violently, with tunable precision, even though the result is still a Horn clause problems without arrays. Section 3 explains how many properties (e.g. initialization) can be proved using one "distinguished cell", Section 4 explains how properties such as sortedness can be proved using two cells; completely discarding arrays corresponds to using zero of them.

We illustrate this approach with automated proofs of several examples from the literature: we apply Section 3, 4 or 5.2 to obtain a system of Horn clauses without arrays. This system is then fed to the Z3, ELDARICA or SPACER solver, which produces a model of this system, meaning that the postcondition (e.g. sortedness or multiset of the output equal to that of the input) truly holds.[3]

Previous approaches [25] using "distinguished cells" amounted (even though not described as such) to linear Horn rules; on contrast, our abstract semantics uses non-linear Horn rules, which leads to higher precision (Sec. 7.2).

Multiset of contents It is often necessary to reason not only about individual elements of an array or map, but also about its contents as a whole: e.g. sorting algorithms preserve the contents of the array (even though, locally, when moving elements around, they may break this invariant).

The multiset of the contents of an array of elements of type β is a map from β to $\mathbb{N}$. Using that remark, we can abstract the array both using our "distinguish cell" approach and as the multiset of its elements (Sec. 5.2); we provide suitable program transformations.

We illustrate that approach with an automated proof that the output of selection sort has the same contents as its input (that is, the output is a permutation of the input).

[2] With the exception of pointers and references, which need special handling and may be internally converted to array accesses.

[3] Z3 and ELDARICA can also occasionally directly solve Horn clauses over arrays; we also compare to that.

Contributions Our main contribution is a system of rules for transforming the atomic program statements in a program operating over arrays or maps, as well as (optionally) the universally quantified postcondition to prove, into a system of non-linear Horn clauses over scalar variables only. The precision of this transformation is tunable using a number of "distinguished cells"; e.g. properties such as sortedness need two distinguished cells (Section 4) while simpler properties need only one (Section 3). Statements operating over non-arrays variables are mapped (almost) identically to their concrete semantics. This system over-approximates the behavior of the program. A solution of that system can be mapped to inductive invariants over the original programs, including universally properties over arrays.

A second contribution, based upon the first, is a system of rules of the same kind that also keeps tracks of array/map contents (Section 5.2) as a multiset. This system is suitable for showing e.g. that the output of a sorting algorithm is a permutation of the input, even though the sequence of operations is not directly a sequence of swaps.

We implemented our approach and benchmarked it over several classical examples of array algorithms (Section 6), comparing it favorably to other tools.

2. Program Verification as solving Horn clauses

A classical approach to program analysis is to consider a program as a control-flow graph and to attach to each vertex p_i (control point) an *inductive invariant* I_i: a set of possible values $\mathbf{x}$ of the program variables (and memory stack and heap, as needed) so that i) the set associated to the initial control point p_{i_0} contains the possible initialization values S_{i_0} ii) for each edge $p_i \rightarrow_c p_j$ (c for *concrete*), the set I_j associated to the target control point p_j should include all the states reachable from the states in the set I_i associated to the source control point p_i according to the transition relation $\tau_{i,j}$ of the edge. Inductiveness is thus defined by *Horn clauses*:

$$\forall \mathbf{x},\ \mathbf{x} \in S_{i_0} \implies \mathbf{x} \in I_{i_0} \tag{1}$$

$$\forall \mathbf{x}, \mathbf{x}'\ \mathbf{x} \in I_i \wedge (\mathbf{x}, \mathbf{x}') \in \tau_{i,j} \implies \mathbf{x}' \in I_j \tag{2}$$

For proving safety properties, in addition to inductiveness, one requires that error locations $p_{e_1}, \ldots, p_{e_n}$ are proved to be unreachable (the associated set of states is empty): this amounts to Horn clauses implying false ($\bot$): $\forall \mathbf{x},\ \mathbf{x} \in I_{e_i} \implies \bot$.

Various tools can solve such systems of Horn clauses, that is, can synthesize suitable predicates I_i, which constitute *inductive invariants*. In this article, we tried Z3[4] with the PDR fixed point solver [18], Z3 with the SPACER solver

[4] https://github.com/Z3Prover hash 7f6ef0b6c0813f2e9e8f993d45722c0e5b99e152; due to various problems we preferred not to use results from later versions.

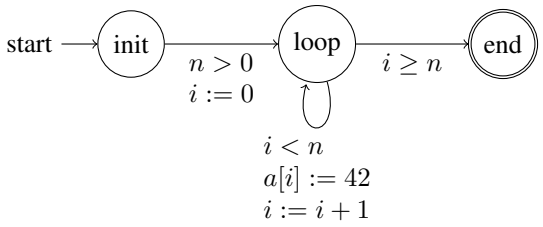

Figure 1. Compact control-flow graph for Program 1

[20, 21],[5] and ELDARICA[29].[6] Since program verification is undecidable, such tools, in general, may fail to terminate, or may return "unknown".

For the sake of simplicity, we shall consider, in this article, that all integer variables in programs are mathematical integers ($\mathbb{Z}$) as opposed to machine integers[7] and that arrays are infinite. Again, it is easy to modify our semantics to include systematic array bound checks, jumps to error conditions, etc.

In examples, instead of writing I_{stmt} for the name of the predicate (inductive invariant) at statement $stmt$, we shall write $stmt$ directly, for readability's sake: thus we write e.g. $loop$ for a predicate at the head of a loop. Furthermore, we shall sometimes coalesce several successive statements into one, for the sake of readability.

Example 1 (Motivating example)**.** *Consider the program:*

Listing 1. 1D array fill

```
void array_fill1(int n, int a[n]) {
  int i = 0;
  while(i < n) {
    a[i] = 42;
    i = i+1;
  }
}
```

We would like to prove that this program truly fills array a[] *with value 42. The flat encoding into Horn clauses assigns a predicate (set of states) to each of the control nodes (Fig. 1), and turns each transition into a Horn rule:*

$$\forall n \in \mathbb{Z}\, \forall a \in Array\,(\mathbb{Z},\mathbb{Z})\ n > 0 \implies loop(n, 0, a) \quad (3)$$

$$\forall n, i \in \mathbb{Z}\, \forall a \in Array\,(\mathbb{Z},\mathbb{Z})\ i < n \wedge loop(n, i, a) \implies loop(n, i+1, store(a, i, 42)) \quad (4)$$

$$\forall n, i \in \mathbb{Z}\, \forall a \in Array\,(\mathbb{Z},\mathbb{Z})\ i \geq n \wedge loop(n, i, a) \implies end(n, a) \quad (5)$$

$$\forall n \in \mathbb{Z}\, \forall a \in Array\,(\mathbb{Z},\mathbb{Z})\ 0 \leq x < n \wedge end(n, a) \implies a[x] = 42 \quad (6)$$

where $store(a, i, v)$ *is array* a *where the value at index* i *has been replaced by* v*.*

None of the tools we have tried (Z3/PDR, Z3/SPACER, ELDARICA) has been able to solve this system, presumably because they cannot infer universally quantified invariants over arrays. Indeed, here the invariant needed in the loop is

$$0 \leq i \leq n \wedge (\forall k\ 0 \leq k < i \implies a[k] = 42) \quad (7)$$

While $0 \leq i \leq n$ is inferred by a variety of approaches, the rest of the formula is a tougher problem.

Most software model checkers attempt constructing invariants from *Craig interpolants* obtained from refutations of the accessibility of error states in partial unfoldings of the problem, but interpolation over array properties is difficult, especially since the goal is not to provide any interpolant, but interpolants that generalize well to invariants [1, 2].

This article instead introduces a way to derive universally quantified invariants from the analysis of a system of Horn clauses on scalar variables (without array variables).

3. Getting rid of the arrays

To use the power of Horn solvers, we soundly abstract problems with arrays to problems without arrays.

In the Horn clauses for example 1, we attached to each program point p_ℓ a predicate I_ℓ over $\mathbb{Z} \times \mathbb{Z} \times Array\,(\mathbb{Z},\mathbb{Z})$ when the program variables are two integers i, n and one integer-value, integer-indexed array a. In any solution of the system of clauses, $\neg I_\ell(i, n, a)$ implies that i, n, a cannot be reached at program point p_ℓ. Instead, we will consider a predicate ("with one distinguished cell") $I^\sharp_\ell$ over $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$ such that $\neg I^\sharp_\ell(i, n, k, a_k)$ [8] implies that there is at p_ℓ no reachable state (i, n, a) such that $a[i] = a_k$. We thus have to provide abstract transformers for each statement.

Without loss of generality, any statement in the program can be assumed to be either:

i) an array read to a fresh variable, v=a[i]; in C syntax, $v := a[i]$ in pseudo-code; the variables of the program are $(\mathbf{x}, i, v)$ where $\mathbf{x}$ is a vector of arbitrarily many variables;
ii) an array write, a[i]=v; (where v and i are variables) in C syntax, $a[i] := v$ in pseudo-code; the variables of the program are $(\mathbf{x}, i, v)$ before and after the statement;
iii) a scalar operation, including assignments and guards over scalar variables.

More complex statements can be transformed to a sequence of such statements, by introducing temporary variables if needed: for instance, $a[i] := a[j]$ is transformed into $temp := a[j];\ a[i] := temp$.

Definition 1 (Read statement)**.** Let v be a variable of type β, i be a variable of type ι, and a be an array of values of type β with an index of type ι. Let $\mathbf{x}$ be the other program

[5] https://bitbucket.org/spacer/code hash 7e1f9af01b796750d9097b331bb66b752ea0ee3c

[6] https://github.com/uuverifiers/eldarica/releases/tag/v1.1-rc

[7] A classical approach is to add overflow checks to the intermediate representation of programs in order to be able to express their semantics with mathematical integers even though they operate over machine integers.

[8] also denoted by $\neg I^\sharp_\ell((i, n), (k, a_k))$ for sake of readability.

variables, taken in χ. The concrete "next state" relation for the read statement v=a[i]; is $(\mathbf{x}, i, v, a) \rightarrow_c (\mathbf{x}, i, a[i], a)$[9].

Its forward abstract semantics is encoded into two Horn clauses, assuming the statement is between p_1 and p_2:

$$\begin{array}{r} \forall \mathbf{x} \in \chi \, \forall i \in \iota \, \forall v, a_i \in \beta \, \forall k \in \iota \, \forall a_k \in \beta \\ k \neq i \wedge I_1^\sharp\big((\mathbf{x}, i, v), (k, a_k)\big) \wedge I_1^\sharp\big((\mathbf{x}, i, v), (i, a_i)\big) \\ \implies I_2^\sharp\big((\mathbf{x}, i, a_i), (k, a_k)\big) \end{array} \tag{8}$$

$$\begin{array}{r} \forall \mathbf{x} \in \chi \, \forall i \in \iota \, \forall v, a_i \in \beta \, \forall k \in \iota \, \forall a_k \in \beta \\ I_1^\sharp\big((\mathbf{x}, i, v), (i, a_i)\big) \implies I_2^\sharp\big((\mathbf{x}, i, a_i), (i, a_i)\big) \end{array} \tag{9}$$

While rule 9 is straightforward (a_i is assigned to the variable v), the nonlinear[10] rule 8 may be more difficult to comprehend. The intuition is that, to have $a_i = a[i]$ and $a_k = a[k]$ at the read instruction with a given valuation $(\mathbf{x}, i)$ of the other variables, both $a_i = a[i]$ and $a_k = a[k]$ had to be reachable with the same valuation.

Remark 1. *One weakens the semantics by replacing these two rules by a single Rule 8 without the $i \neq k$ guard. Rule 9 ensures that in the outcome, if $i = k$ then $v = a_k$.*

Definition 2 (Write statement)**.** The concrete "next state" relation for the write statement a[i]=v; is $(\mathbf{x}, i, v, a) \rightarrow_c (\mathbf{x}, i, v, store(a, i, v))$. Its forward abstract semantics is encoded into two Horn clauses, depending whether the distinguished cell is i or not:

$$\begin{array}{r} \forall \mathbf{x} \in \chi \, \forall i \in \iota \, \forall v \in \beta \, \forall k \in \iota \, \forall a_k \in \beta \, i \neq k \wedge \\ I_1^\sharp\big((\mathbf{x}, i, v), (k, a_k)\big) \implies I_2^\sharp\big((\mathbf{x}, i, v), (k, a_k)\big) \end{array} \tag{10}$$

$$\begin{array}{r} \forall \mathbf{x} \in \chi \, \forall i \in \iota \, \forall v \in \beta \, \forall k \in \iota \, \forall a_k \in \beta \\ I_1^\sharp\big((\mathbf{x}, i, v), (i, a_k)\big) \implies I_2^\sharp\big((\mathbf{x}, i, v), (i, v)\big) \end{array} \tag{11}$$

Definition 3 (Initialization)**.** Creating an array variable with nondeterministically chosen initial content is abstracted by

$$\forall \mathbf{x} \in \chi \, \forall k \in \iota \, \forall a_k \in \beta \, I_1^\sharp(\mathbf{x}) \implies I_2^\sharp(\mathbf{x}, k, a_k) \tag{12}$$

Definition 4 (Scalar statements)**.** With the same notations as above, we consider a statement (or sequence thereof) operating only on scalar variables: $\mathbf{x} \rightarrow_s \mathbf{x}'$ if it is possible to obtain scalar values $\mathbf{x}'$ after executing the statement on scalar values $\mathbf{x}$. The concrete "next state" relation for that statement is $(\mathbf{x}, i, v, a) \rightarrow_c (\mathbf{x}', i, v, a)$. Its forward abstract semantics is encoded into one Horn clause:

$$\begin{array}{r} \forall \mathbf{x} \in \chi \, \forall k \in \iota \, \forall a_k \in \beta \\ I_1^\sharp(\mathbf{x}, k, a_k) \wedge \mathbf{x} \rightarrow_s \mathbf{x}' \implies I_2^\sharp(\mathbf{x}', k, a_k) \end{array} \tag{13}$$

Example 2. *A test $x \neq y$ gets abstracted as*

$$\forall x, y, k, a_k \; I_1^\sharp(x, y, k, a_k) \wedge x \neq y \implies I_2^\sharp(x, y, k, a_k) \tag{14}$$

[9] There is a slight abuse in notation as variable v might belong to $\mathbf{x}$

[10] Nonlinear in the sense that it refers to $I_1^\sharp$ twice in its antecedents, and thus unfolding it tends to create a tree structure, as opposed to a "comb".

Definition 5. The scalar operation $kill(v_1, \ldots, v_n)$ removes variables $v_1, \ldots, v_n$: $(\mathbf{x}, v_1, \ldots, v_n) \rightarrow \mathbf{x}$.

We shall apply it to get rid of dead variables, sometimes, for the sake of brevity, without explicit note, by coalescing it with other operations.

We use the same Galois connection [10] as some earlier works [25] [11, Sec. 2.1]:

Definition 6. The *concretization* of $I^\sharp \subseteq \chi \times (\iota \times \beta)$ is

$$\gamma(I^\sharp) = \{(\mathbf{x}, a) \mid \forall i \in \iota \, (\mathbf{x}, i, a[i]) \in I^\sharp\} \tag{15}$$

The *abstraction* of $I \subseteq \chi \times Array\,(\iota, \beta)$ is

$$\alpha(I) = \{(\mathbf{x}, i, a[i]) \mid x \in \chi, i \in \iota\} \tag{16}$$

Theorem 1. *α and γ form a Galois connection*

$$\mathcal{P}\left(\chi \times Array\,(\iota, \beta)\right) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{P}\left(\chi \times (\iota \times \beta)\right).$$

Our Horn rules are of the form $\forall \mathbf{y} \; I_1^\sharp(\mathbf{f_1}(\mathbf{y})) \wedge \cdots \wedge I_1^\sharp(\mathbf{f_m}(\mathbf{y})) \wedge P(\mathbf{y}) \implies I_2^\sharp(\mathbf{g}(\mathbf{y}))$ ($\mathbf{y}$ is a vector of variables, $\mathbf{f_1}, \ldots, \mathbf{f_m}$ vectors of terms depending on $\mathbf{y}$, P an arithmetic predicate over $\mathbf{y}$). In other words, they impose in $I_2^\sharp$ the presence of $\mathbf{g}(\mathbf{y})$ as soon as certain elements $\mathbf{f_1}(\mathbf{y}), \ldots, \mathbf{f_m}(\mathbf{y})$ are found in $I_1^\sharp$. Let $I_{2-}^\sharp$ be the set of such imposed elements. This Horn rule is said to be *sound* if $\gamma(I_{2-}^\sharp)$ includes all states $(\mathbf{x}', a')$ such that there exists $(\mathbf{x}, a)$ in $\gamma(I_1^\sharp)$ and $(\mathbf{x}, a) \rightarrow_c (\mathbf{x}', a')$.

Lemma 2. *The forward abstract semantics of the read statement (Def. 1), of the write statement (Def. 2), of array initialization (Def. 3), of the scalar statements (Def. 4) are sound w.r.t the Galois connection.*

Remark 2. *The scalar statements include "killing" dead variables (Def. 5). Note that, contrary to many other abstractions, in ours, removing some variables may cause irrecoverable loss of precision on other variables [25, Sec. 4.2]: if v is live, then one can represent $\forall k, \; a[k] = v$, which implies $\forall k_1, k_2 \; a[k_1] = a[k_2]$ (constantness), but if v is discarded, the constantness of a is lost.*

Theorem 3. *If $I_1^\sharp, \ldots, I_m^\sharp$ are a solution of a system of Horn clauses sound in the above sense, then $\gamma(I_1^\sharp), \ldots, \gamma(I_m^\sharp)$ are inductive invariants w.r.t the concrete semantics $\rightarrow_c$.*

Proof. From the general properties of fixed points of monotone operators and Galois connections [10]. □

Definition 7 (Property conversion)**.** A property "at program point p_ℓ, for all $\mathbf{x} \in \chi$ and all $k \in \iota$, $\phi(\mathbf{x}, k, a[k])$ holds" (where ϕ is a formula, say over arithmetic) is converted into a Horn query $\forall \mathbf{x} \in \chi \, \forall k \in \iota \, \phi(\mathbf{x}, k, a_k)$.

Our method for converting a scalar program into a system of Horn clauses over scalar variables is thus:

Algorithm 1. 1. Construct the control-flow graph of the program.

2. To each control point p_ℓ, with vector of scalar variables $\mathbf{x}_\ell$, associate a predicate $I^\sharp_\ell(\mathbf{x}_\ell, k, a_k)$ in the Horn clause system (the vector of scalar variables may change from control point to control point).
3. For each transition of the program, generate Horn rules according to Def. 1, 2, 4 as applicable (an initialization node does not need antecedents in its rule).
4. Generate Horn queries from desired properties according to Def. 7.

Example (Ex. 1, continued)**.** *Let us now apply the Horn abstract semantics from Definitions 2 and 4 to Program 1; in this case, $\alpha = \mathbb{Z}$, $\iota = \{0, \ldots, n-1\}$, $\chi = \mathbb{Z}$. After a slight simplification of the Horn clauses, we obtain (formula 4 has been replaced by 19 and 20):*

$$\forall n, k, a_k \in \mathbb{Z}\ 0 \leq k < n \implies loop(n, 0, k, a_k) \quad (17)$$

$$\forall n, i, k, a_k \in \mathbb{Z}\ 0 \leq k < n \wedge i < n \wedge loop(n, i, k, a_k) \implies write(n, i, k, a_k) \quad (18)$$

$$\forall n, i, k, a_k \in \mathbb{Z}\ 0 \leq k < n \wedge i \neq k \wedge write(n, i, k, a_k) \implies incr(n, i, k, a_k) \quad (19)$$

$$\forall n, i, a_k \in \mathbb{Z}\ \wedge write(n, i, i, a_k) \implies incr(n, i, i, 42) \quad (20)$$

$$\forall n, i, k, a_k \in \mathbb{Z}\ 0 \leq k < n \wedge incr(n, i, k, a_k) \implies loop(n, i+1, k, a_k) \quad (21)$$

$$\forall n, i, k, a_k \in \mathbb{Z}\ 0 \leq k < n \wedge i \geq n \wedge loop(n, i, k, a_k) \implies end(n, k, a_k) \quad (22)$$

Finally, we add the postcondition (using Def. 7):

$$\forall n, k, a_k\ 0 \leq k < n \wedge end(n, i, k, a_k) \Rightarrow a_k = 42 \quad (23)$$

A solution to the resulting system of Horn clauses can be found by e.g. Z3/PDR.

Our approach can also be used to establish relationships between several arrays, or between the initial values in an array and the final values.

Example 3. *Consider the problem of finding the minimum of an array slice $a[l \ldots h-1]$, with value $b = a[p]$:*

Listing 2. Find minimum in an array slice

```
void find_minimum(int n, int a[n], int l,
    int h){
  int p = l, b = a[l], i = l+1;
  while(i < h) {
    int v = a[i];
    if (v < b) {
      b = v;
      p = i;
    }
    i = i+1;
  }
}
```

Again, we encode the abstraction of the statements (Def. 1, 2, 4) as Horn clauses. At the end we have a predicate $end(l, h, p, b, k, a[k])$ on which we impose the properties

$$\forall l, h, p, b, a_p\ end(l, h, p, b, p, a_p) \implies b = a_p \quad (24)$$

$$\forall l, h, p, b, k, a_p, a_k\ l \leq k < h \wedge end(l, h, p, b, k, a_k) \implies b \leq a_k \quad (25)$$

Rule 24 imposes the postcondition $b = a[p]$, Rule 25 imposes the postcondition $\forall k\ l \leq k < h \implies b \leq a[k]$.

No restrictions on domain type and relationships The kind of relationship that can be inferred between loop indices, array indices and array contents is limited only by the capabilities of the Horn solver. For instance, invariants of the form $\forall i\ i \equiv 0 \pmod 2 \implies a[i] = 0$ may be inferred if the Horn solver supports numeric invariants involving divisibility. Similarly, we have made no assumption regarding the nature of the indexing variable: we used integers because arrays indexed by an integer range are a very common kind of data structure, but really it can be any type supported by the Horn clause solver, e.g. rationals or strings.

Matrices Matrices are bidimensional arrays, that is, arrays indexed by two integers x and y: $0 \leq x < m, 0 \leq y < n$ for a $m \times n$ arrays. More generally, arrays can be defined for an arbitrary number d of dimensions. Everything that we have seen so far applies when the type ι of the indexing variable is $\mathbb{Z}^d$. We may therefore apply directly what precedes and generate Horn clauses referring to pairs of indices (x, y). Since not every solver supports these, one may instead use two indices x and y: a comparison $(x_1, y_1) = (x_2, y_2)$ is expressed as $x_1 = x_2 \wedge y_1 = y_2$.

Example 4. *The following program fills a $m \times n$ matrix:*

Listing 3. Fill 2D-matrix

```
void array_fill2(int m, int n, int a[m][n]){
  int i = 0;
  while(i < m) {
    int j = 0;
    while(j < n) {
      a[i][j] = 42;
      j = j+1;
    }
    i = i+1;
  }
}
```

Again, we can prove that $\forall m, n, x, y, a_{xy} \in \mathbb{Z}, exit \implies a_{xy} = 42$; otherwise said, finally, $\forall x, y\ a[x, y] = 42$.

4. Sortedness and other *N*-ary predicates

The Galois connection of Def. 6 expresses relations of the form $\forall k \in \iota\ \phi(\mathbf{x}, k, a[k])$ where $\mathbf{x}$ are variables from the program, a a map and k an index into the map a; in other words, relations between each array element individually and the rest of the variables. It cannot express properties

such as sortedness, which link *two* array elements: $\forall k_1, k_2 \in \iota\ k_1 < k_2 \implies a[k_1] \leq a[k_2]$.

Definition 8. The abstraction with two "distinguished cells" is defined by the concretization and abstraction:

$$\gamma_2(I^\sharp) = \{(\mathbf{x}, a) \mid \forall k_1, k_2 \in \iota\ (\mathbf{x}, k_1, a[k_1], k_2, a[k_2]) \in I^\sharp\} \quad (26)$$

$$\alpha_2(I) = \{(\mathbf{x}, k_1, a[k_1], k_2, a[k_2]) \mid x \in \chi, k_1, k_2 \in \iota\} \quad (27)$$

Theorem 4. *α_2 and γ_2 form a Galois connection.*

$$\mathcal{P}\left(\chi \times Array\left(\iota, \beta\right)\right) \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \mathcal{P}\left(\chi \times \left(\iota \times \beta\right)^2\right).$$

With respect to implementation efficiency, it may be preferable to break this symmetry between indices k_1 and k_2 by imposing $k_1 < k_2$ for some total order.

Definition 9. The abstraction with indices $k_1 < k_2$ is

$$\gamma_{2<}(I^\sharp) = \{(\mathbf{x}, a) \mid \forall k_1 < k_2 \in \iota\ (\mathbf{x}, k_1, a[k_1], k_2, a[k_2]) \in I^\sharp\} \quad (28)$$

$$\alpha_{2<}(I) = \{(\mathbf{x}, k_1, a[k_1], k_2, a[k_2]) \mid x \in \chi, k_1 \leq k_2 \in \iota\} \quad (29)$$

Theorem 5. *$\alpha_{2<}$ and $\gamma_{2<}$ form a Galois connection*

$$\mathcal{P}\left(\chi \times Array\left(\iota, \beta\right)\right) 5 \underset{\alpha_{2\leq}}{\overset{\gamma_{2\leq}}{\leftrightarrows}}$$
$$\mathcal{P}\left(\{(x, k_1, v_1, k_2, v_2) \mid x \in \chi,\ k_1 < k_2 \in \iota,\ v_1, v_2 \in \beta\}\right).$$

These constructions easily generalize to arbitrary N indices $k_1, \ldots, k_N$.

Definition 10 (Read statement, two indices $k_1 < k_2$)**.** The abstraction of $v := a[i]$ is:[11]

$$I_1^\sharp(\mathbf{x}, i, k_1, a_{k_1}, k_2, a_{k_2}) \wedge I_1^\sharp(\mathbf{x}, i, i, a_i, k_2, a_{k_2}) \wedge I_1^\sharp(\mathbf{x}, i, i, a_i, k_1, a_{k_1}) \wedge i < k_1 < k_2 \implies I_2^\sharp(\mathbf{x}, i, a_i, k_1, a_{k_1}, k_2, a_{k_2}) \quad (30)$$

$$I_1^\sharp(\mathbf{x}, i, i, a_i, k_2, a_{k_2}) \wedge I_1^\sharp(\mathbf{x}, i, k_1, a_{k_1}, k_2, a_{k_2}) \wedge I_1^\sharp(\mathbf{x}, i, i, a_i, k_1, a_{k_1}) \wedge k_1 < i < k_2 \implies I_2^\sharp(\mathbf{x}, i, a_i, k_1, a_{k_1}, k_2, a_{k_2}) \quad (31)$$

$$I_1^\sharp(\mathbf{x}, i, k_2, a_{k_2}, i, v) \wedge I_1^\sharp(\mathbf{x}, i, k_1, a_{k_1}, i, v) \wedge I_1^\sharp(\mathbf{x}, i, k_1, a_{k_1}, k_2, a_{k_2}) \wedge k_1 < k_2 < i \implies I_2^\sharp(\mathbf{x}, i, a_i, k_1, a_{k_1}, k_2, a_{k_2}) \quad (32)$$

$$I_1^\sharp(\mathbf{x}, i, i, a_i, k_2, a_{k_2}) \wedge i < k_2 \implies I_2^\sharp(\mathbf{x}, i, i, a_i, k_2, a_{k_2}) \quad (33)$$

$$I_1^\sharp(\mathbf{x}, i, k_1, a_{k_1}, i, v) \wedge k_1 < i \implies I_2^\sharp(\mathbf{x}, i, k_1, a_{k_1}, i, v) \quad (34)$$

This construction generalizes to N-ary abstraction by considering all orderings of i inside $k_1 < \cdots < k_N$, and for each ordering taking all sub-orderings of size N.

[11] We shall now on omit universal quantifiers in front of clauses.

Definition 11 (Write statement, two indices $k_1 < k_2$)**.** The abstraction of $a[i] := v$ is:

$$I_1^\sharp(\mathbf{x}, i, v, k_1, a_{k_1}, k_2, a_{k_2}) \wedge i \neq k_1 \wedge i \neq k_2 \implies I_2^\sharp(\mathbf{x}, i, v, k_1, a_{k_1}, k_2, a_{k_2}) \quad (35)$$

$$I_1^\sharp(\mathbf{x}, i, v, i, a_{k_1}, k_2, a_{k_2}) \wedge i < k_2 \implies I_2^\sharp(\mathbf{x}, i, v, i, v, k_2, a_{k_2}) \quad (36)$$

$$I_1^\sharp(\mathbf{x}, i, v, k_1, a_{k_1}, i, a_{k_2}) \wedge k_1 < i \implies I_2^\sharp(\mathbf{x}, i, v, k_1, a_{k_1}, i, v) \quad (37)$$

Lemma 6. *The abstract forward semantics of the read statement (Def. 10) and of the write statement(Def. 11) are sound w.r.t the Galois connection.*

Example 5 (Selection sort)**.** *Selection sort finds the least element in $a[l \ldots h-1]$ (using Prog. 2 as its inner loop) and swaps it with $a[l]$, then sorts $a[l+1, h-1]$. At the end, $a[l_0 \ldots h-1]$ is sorted, where l_0 is the initial value of l.*

Listing 4. Selection sort

```
void selection_sort(int l0, int h, int a[])
  {
  int l = l0;
  while (l < h-1) {
    int p = l, b = a[l], f = b, i = l+1;
    while(i < h) {
      int v = a[i];
      if (v < b) {
        b = v;
        p = i;
      }
      i = i+1;
    }
    a[l] = b;
    a[p] = f;
    l = l+1;
  }
}
```

Using the rules for the read (Def. 10) and write (Def. 11) statements, we write the abstract forward semantics of this program as a system of Horn clauses.

We wish to prove that, at the end, $a[l_0, h-1]$ is sorted: at the exit node,

$$\forall l_0 \leq k_1 < k_2 < h\ \ a[k_1] \leq a[k_2] \quad (38)$$

This is expressed as the final condition

$$\forall l_0, h, k_1, a_{k_1}, k_2, a_{k_2}\ l_0 \leq k < k_2 < h \wedge exit(l_0, h, k_1, a_{k_1}, k_2, a_{k_2}) \implies a_{k_1} \leq a_{k_2} \quad (39)$$

By running a solver on these clauses, we show that the output of selection sort is truly sorted[12] Let us note that this

[12] In Example 6 we shall see how to prove that the multiset of elements in the output is the same as in the input.

proof relies on nontrivial invariants:[13]

$$\forall k_1, k_2\ l_0 \leq k_1 < l \wedge k_1 \leq k_2 < h \implies a[k_1] \leq a[k_2] \tag{40}$$

This invariant can be expressed in our Horn clauses as:

$$\begin{array}{l}\forall l_0, l, h, k_1, a_{k_1}, k_2, a_{k_2} \in \mathbb{Z}\ l_0 \leq k_1 < l \wedge k_1 < k_2 < h \\ \quad \wedge \mathit{outerloop}(l_0, l, h, k_1, a_{k_1}, k_2, a_{k_2}) \implies a_{k_1} \leq a_{k_2}\end{array} \tag{41}$$

If this invariant is added to the problem as an additional query to prove, solving time is reduced from 6 min to 1 s. It may seem counter-intuitive that a solver takes less time to solve a problem with an additional constraint; but this constraint expresses an invariant necessary to prove the solution, and thus nudges the solver towards the solution.

Our approach is therefore flexible: if a solver fails to prove the desired property on its own, it is possible to help it by providing partial invariants. This is a less tedious approach than having to provide full invariants at every loop header, as common in assisted Floyd-Hoare proofs.

5. Sets and multisets

Our abstraction for maps may be used to abstract (multi)sets.

5.1 Simple sets and multisets

Many programming languages provide libraries for computing over sets or multisets of elements. One should reason on programs using these libraries by using the set-theoretic, high-level specification of their interface, as opposed to internal implementation details.

Remark, again, that we have made no assumption on the set of indices ι (except, occasionally, that is endowed with a total order, but that assumption may be dispensed from). A subset of ι is just a map from ι to the Booleans, a multiset a map from ι to the natural numbers. Testing the membership of one item $k \in \iota$ therefore just amounts to an array read $a[k]$, forcing membership or non-membership just amounts to a write.

A single (multi)set a is abstracted as a set of pairs $(k, a[k])$. If one has several (multi)sets a, b, c, one may either abstract them with separate indices $(i, a[i], j, a[j], k, a[k])$, or with a common index $(k, a[k], b[k], c[k])$. This last option is less expressive, but simpler, and is often sufficient.

Definition 12 ((Multi)set union)**.** Let a, b, c be three multisets. The operation $a := \mathit{union}(b, c)$ is abstracted as:

$$\begin{array}{l}\forall \mathbf{x} \in \chi\ \forall k \in \iota\ I_1^\sharp(\mathbf{x}, k, a_k, b_k, c_k) \\ \quad \implies I_2^\sharp(\mathbf{x}, k, b_k \vee c_k, b_k, c_k)\end{array} \tag{42}$$

(For multiset, replace $\vee$ by $+$.)

[13] Nontrivial in the sense that a human user operating a Floyd-Hoare proof assistant typically does not come up with them so easily.

Definition 13 (Set intersection)**.** Let a, b, c be three multisets. The operation $a := \mathit{intersection}(b, c)$ is abstracted as:

$$\begin{array}{l}\forall \mathbf{x} \in \chi\ \forall k \in \iota\ I_1^\sharp(\mathbf{x}, k, a_k, b_k, c_k) \\ \quad \implies I_2^\sharp(\mathbf{x}, k, b_k \wedge c_k, b_k, c_k)\end{array} \tag{43}$$

If operations such as "get the (min/max)imal element" are to be abstracted precisely, then one can enrich the abstraction by adding tracking variables l and h for the minimal and maximal elements, and updating them accordingly. In the case of sets of integers, such tracking variables may be used to implement the "for each" iterator: iterate i from l to h and test whether i is in the set.

5.2 Multiset of elements in an array

In Example 5, we showed how to prove that the output of selection sort is sorted. This is not enough for functional correctness: we also have to prove that the output is a permutation of the input, or, equivalently, that the multiset of elements in the output array is the same as that in the input array.

Let us remark that it is easy to keep track, in an auxiliary map, of the number $\#a(x)$ of elements of value x in the array $a[]$. Only write accesses to $a[]$ have an influence on $\#a$: a write $a[i] := v$ is replaced by a sequence:

$$\#a(a[i]) := \#a(a[i]) - 1;\ a[i] := v;\ \#a(v) := \#a(v) + 1 \tag{44}$$

(that is, in addition to the array write, the count of elements for the value that gets overwritten is decremented, and the count of elements for the new value is incremented).

This auxiliary map $\#a$ can itself be abstracted using our approach! Let us now see how to implement this in our abstract forward semantics expressed using Horn clauses. We enrich our Galois connection (Def. 6) as follows:

Definition 14. The *concretization* of $I^\sharp \subseteq \chi \times (\iota \times \beta) \times (\beta \times \mathbb{N})$ is

$$\begin{array}{l}\gamma_\#(I^\sharp) = \Big\{(\mathbf{x}, a) \mid \forall i \in \iota\ \forall v \in \beta \\ \quad \big(\mathbf{x}, (i, a[i]), (v, \mathrm{card}\{j \in \iota \mid a[j] = v\})\big) \in I^\sharp\Big\}\end{array} \tag{45}$$

where $\mathrm{card}\, X$ denotes the number of elements in the set X.

The *abstraction* of $I \subseteq \chi \times \mathit{Array}\,(\iota, \beta)$ is

$$\begin{array}{l}\alpha_\#(I) = \Big\{\big(\mathbf{x}, (i, a[i]), (v, \mathrm{card}\{j \in \iota \mid a[j] = v\})\big) \\ \qquad\qquad \Big| \ x \in \chi, i \in \iota\Big\}\end{array} \tag{46}$$

Theorem 7. *$\alpha_\#$ and $\gamma_\#$ form a Galois connection*

$$\mathcal{P}\left(\chi \times \mathit{Array}\,(\iota, \beta)\right) \underset{\alpha_\#}{\overset{\gamma_\#}{\leftrightarrows}} \mathcal{P}\left(\chi \times (\iota \times \beta) \times (\beta \times \mathbb{N})\right).$$

The Horn rules for array reads and for scalar operations are the same as those for our first abstraction, except that we carry over the extra two components identically.

Definition 15 (Read statement)**.** Same notations as Def. 1:

$$k \neq i \wedge I_1^\sharp\big((\mathbf{x}, i, v), (k, a_k), (z, a_{\#z})\big) \wedge$$
$$I_1^\sharp\big((\mathbf{x}, i, v), (i, a_i), (z, a_{\#z})\big) \implies I_2^\sharp\big((\mathbf{x}, i, a_i), (k, a_k), (z, a_{\#z})\big)$$
$$I_1^\sharp\big((\mathbf{x}, i, v), (i, a_i), (z, a_{\#z})\big) \implies I_2^\sharp\big((\mathbf{x}, i, a_i), (i, a_i), (z, a_{\#z})\big)$$

Lemma 8. *The abstract forward semantics of the read statement (Def. 15) is a sound abstraction of the concrete semantics given in Def. 1.*

The abstraction of the write statement is more complicated (see the sequence of instructions in Formula 44). To move by a write operation $a[i] := v$ from a control point p_1 to a control point p_2, we need two intermediate control points p_a and p_b.

Definition 16 (Write statement)**.** With the same notations in Def. 2:

$$a_i \neq z \wedge I_1^\sharp\big((\mathbf{x}, i, v), (k, a_k), (z, a_{\#z})\big)$$
$$\wedge I_1^\sharp\big((\mathbf{x}, i, v), (i, a_i), (z, a_{\#z})\big) \implies I_a^\sharp\big((\mathbf{x}, i, v), (k, a_k), (z, a_{\#z})\big) \tag{47}$$

$$I_1^\sharp\big((\mathbf{x}, i, v), (k, a_k), (a_i, a_{\#z})\big) \wedge I_1^\sharp\big((\mathbf{x}, i, v), (i, a_i), (a_i, a_{\#z})\big)$$
$$\implies I_a^\sharp\big((\mathbf{x}, i, v), (k, a_k), (a_i, a_{\#z} - 1)\big) \tag{48}$$

$$v \neq z \wedge I_a^\sharp\big((\mathbf{x}, i, v), (k, a_k), (z, a_{\#z})\big)$$
$$\wedge I_a^\sharp\big((\mathbf{x}, i, v), (i, a_i), (z, a_{\#z})\big) \implies I_b^\sharp\big((\mathbf{x}, i, v), (k, a_k), (z, a_{\#z})\big) \tag{49}$$

$$I_a^\sharp\big((\mathbf{x}, i, v), (k, a_k), (v, a_{\#z})\big) \wedge I_a^\sharp\big((\mathbf{x}, i, v), (i, a_i), (v, a_{\#z})\big)$$
$$\implies I_b^\sharp\big((\mathbf{x}, i, v), (k, a_k), (v, a_{\#z} + 1)\big) \tag{50}$$

$$i \neq k \wedge I_1^\sharp\big((\mathbf{x}, i, v), (k, a_k), (z, a_{\#z})\big)$$
$$\implies I_2^\sharp\big((\mathbf{x}, i, v), (k, a_k), (z, a_{\#z})\big) \tag{51}$$

$$I_1^\sharp\big((\mathbf{x}, i, v), (i, a_k), (z, a_{\#z})\big) \implies I_2^\sharp\big((\mathbf{x}, i, v), (i, v), (z, a_{\#z})\big) \tag{52}$$

Lemma 9. *The abstract forward semantics of the write statement (Def. 16) is a sound abstraction of the concrete semantics given in Def. 2.*

If we want to compare the multiset of the contents of an array a at the end of a procedure to its contents at the beginning of the procedure, one needs to keep a copy of the old multiset. It is common that the property sought is a relation between the number of occurrences $\#a(z)$ of an element z in the output array a and its number of occurrences $\#a_0(z)$ in the input array a^0. In the above formulas, one may therefore replace the pair $(z, a_{\#z})$ by $(z, a_{\#z}, a^0_{\#z})$, with $a^0_{\#z}$ always propagated identically.

Example 6. *Consider again selection sort (Program 4). We use the abstract semantics for read (Def. 15) and write (Def. 16), with an additional component $a^0_{\#z}$ for tracking the original number of values z in the array a.*

We specify the final property as the query

$$exit(l_0, h, k, a_k, z, a_{\#z}, a^0_{\#z}) \implies a_{\#z} = a^0_{\#z} \tag{53}$$

6. Experiments

Implementation We implemented our prototype VAPHOR in 2k lines of OCAML. VAPHOR takes as input a mini-Java program (a variation of WHILE with array accesses, and assertions) and produces a SMTLIB2 file[14]. The core analyzer implements the translation for one-dimensional arrays described in Section 3 and Section 4, and also the direct translation toward a formula with array variables.

Experiments We have tested our analyser on several examples from the literature, including the array benchmark proposed in [13] also used in [3] (Table 1); and other classical array algorithms including *selection sort*, *bubble sort* and *insertion sort* (Table 2). We compared our approach to existing Horn clause solvers capable of dealing with arrays. All these files are provided as supplementary material.

Limitations Our tool does not currently implement multi-dimensional arrays (matrices) or reasoning over array contents (multiset of values). Experiments for these were thus conducted by manually applying the transformations described in this article in order to obtain a system of Horn clauses. For this reason, because applying rules manually is tedious and error-prone, the only sorting algorithm for which we have checked that the multiset of the output is equal to the multiset of the inputs is selection sort. We are however confident that the two other algorithms would go through, given that they use similar or simpler swapping structures.

Some examples from Dillig et al. [13] involve invariants with even/odd constraints. The Horn solvers we tried do not seem to be able to infer invariants involving divisibility predicates unless these predicates were given by the user. For these cases we added these even/odd properties as additional invariants to prove.

Efficiency caveats Our tool does not currently simplify the system of Horn clauses that it produces. We have observed that, in some cases, manually simplifying the clauses (removing useless variables, inlining single antecedents by substitution…) dramatically reduces solving times. Also, precomputing some simple scalar invariants on the Horn clauses (e.g. $0 \leq k < i$ for a loop from k to $i-1$) and asserting them as assertions to prove in the Horn system sometimes reduces solving time.

We have observed that the execution time of a Horn solver may dramatically change depending on minor changes in the input, pseudorandom number generator seed, or version of the solver. For instance, the same version of Z3 solves the same system of Horn clauses (proving the correctness of selection sort) in 3m 40s or 3h 52m depending on whether the random seed is 1 or 0.[15]

[14] http://smtlib.cs.uiowa.edu/

[15] We suspect that different choices in SAT lead to different proofs of unsatisfiability, thus different interpolants and different refinements in the PDR algorithm.

Table 1. Comparison on the array benchmarks of [13]. Timing are in seconds, CPU time. $N > 0$ indicates our abstraction with N indices, $N = -1$ indicates a call to a Horn solver supporting arrays, for reference, when at least one solver succeeded. "sat" means the property was proved, "unsat" that it could not be proved ($N > 0$) or that it was disproved by a counterexample ($N = -1$). Timeout was 5 mn unless otherwise noted. The machine has 32 i3-3110M cores, 64 GiB RAM, C/C++ solvers were compiled with gcc 4.8.4, the JVM is OpenJDK 1.7.0-85. When $N = -1$, Z3 version d7c3e77b66414d1d10f1df73b6b1a792496710e6 was used instead, due to soundness issues with respect to arrays in previous versions.

benchmark	N	Z3/PDR		Z3/SPACER		ELDARICA		comments
		result	time	result	time	result	time	
Correct problems, "sat" expected								
append	1	sat	2.36	sat	3.34	sat	26.90	
append	-1	sat	31.74	sat	0.16	unknown	6.63	
copy	1	sat	7.57	sat	0.78	timeout	331.30	
copyodd	1	sat	0.01	sat	0.00	sat	8.70	
copyodd	-1	sat	0.00	sat	0.00	sat	8.28	
find	1	sat	0.26	sat	0.26	sat	12.45	
find	-1	sat	0.04	sat	0.03	unknown	7.29	
findnonnull	1	sat	0.26	sat	0.15	sat	29.50	
findnonnull	-1	sat	0.04	sat	0.03	unknown	7.14	
init2i	1	sat	0.22	sat	0.17	timeout	325.38	
initcte	1	sat	0.22	sat	0.09	timeout	324.81	
partialcopy	1	sat	2.57	sat	0.65	sat	195.17	
reverse	1	sat	3.79	sat	2.48	sat	95.94	
strcpy	1	sat	4.52	sat	0.54	sat	26.62	
strlen	1	sat	0.27	sat	0.16	sat	125.21	
swapncopy	1	sat	54.10	timeout	299.01	timeout	326.28	
memcpy	1	sat	4.05	sat	0.61	timeout	331.34	
initeven	1	sat	1.32	sat	0.71	timeout	321.19	divisibility constraints added to help the solver
mergeinterleave	1	sat	39.49	sat	4.61	timeout	322.39	invariants "hints" added to help the solver
Incorrect problems, "unsat" expected (recent Z3 can find counterexamples with $N = -1$ on all examples)								
copyodd_buggy	1	unsat	0.06	unsat	0.06	unsat	6.30	
initeven_buggy	1	unsat	0.07	unsat	0.06	unsat	5.23	
reverse_buggy	1	unsat	2.46	unsat	2.02	unsat	57.99	
swapncopy_buggy	1	unsat	1.01	unsat	0.90	unsat	29.23	
mergeinterleave_buggy	1	unsat	2.68	unsat	0.74	unsat	28.44	

Table 2. Other array-manipulating programs, including various sorting algorithms.The ~~striked out~~ result is likely a bug in Z3; the alternative is a bug in Spacer, since the same system cannot be satisfiable and unsatisfiable at the same time.

# benchmark	N	Z3/PDR		Z3/SPACER		ELDARICA		comments
		result	time	result	time	result	time	
fill1D_check_full	1	sat	0.20	sat	0.11	timeout	307.77	
fill1D_check_full	1	timeout	299.14	timeout	299.01	unknown	6.37	
bin_search_check	1	sat	1.49	sat	0.29	crash	1.41	
bin_search_check	-1	sat	0.09	sat	0.06		1.43	
find_mini_check	1	sat	2.66	sat	0.39	sat	91.64	
find_mini_check	-1	sat	1.20	timeout	299.23	unknown	10.49	
revrefill1D_check_buggy	1	unsat	0.09	unsat	0.07	unsat	11.57	
revrefill1D_check_buggy	-1	unsat	0.04	unsat	0.16	unknown	7.17	
array_fill2D	1	sat	0.5	sat	0.5	sat	8.5	manual translation
selection_sort (sortedness)	2	sat	200	timeout	600	timeout	335	
selection_sort (sortedness)	2	~~unsat~~	83	sat	48	timeout	334	manual translation
selection_sort (permutation)	1	timeout	600	sat	9.24	timeout	336	manual translation
bubble_sort	2	timeout	300	sat	80	timeout	339	
bubble_sort	-1	sat	0	sat	0	sat	6.72	
insertion_sort	2	sat	1.26	sat	1.36	sat	112	
insertion_sort	-1	sat	0	sat	0	sat	5	

Furthermore, we have run into numerous problems with solvers, including one example that, on successive versions of the same solver, produced "sat" then "unknown" and finally "unsat", as well as crashes.

For all these reasons, we believe that solving times should not be regarded too closely. The purpose of our experimental evaluation is not to benchmark solvers relative to each other, but to show that our abstraction, even though it is incomplete, is powerful enough to lead to fully automated proofs of functional correctness of nontrivial array manipulations, including sorting algorithms. Tools for solving Horn clauses are still in their infancy and we thus expect performance and reliability to increase dramatically.

7. Related work

7.1 Cell-based abstractions

Smashing The simplest abstraction for an array is to "smash" all cells into a single one — this amounts to removing the k component from our first Galois connection (Def. 6). The weakness of that approach is that all writes are treated as "may writes" or *weak updates*: $a[i] := x$ adds the value x to the set of values admissible for the array a,

but there is no way to remove any value from that set. Such an approach thus cannot treat initialization loops (e.g. Program 1) precisely.

Exploding At the other extreme, for an array of statically known finite length N (which is common in embedded safety-critical software), one can distinguish all cells $a[0], \ldots, a[N-1]$ and treat them as separate variables $a_0, \ldots, a_{N-1}$. This is a good solution when N is small, but a terrible one when N is large: i) many analysis approaches scale poorly with the number of active variables ii) an initialization loop will have to be unrolled N times to show it initializes all cells. Both smashing and exploding have been used with success in the Astrée static analyzer [5, 6].

Slices More sophisticated analyses [12, 14, 15, 26, 27] distinguish *slices* or *segments* in the array, their boundaries depending on the index variables. For instance, in array initialization (Program 1), one slice is the part already initialized (indices $< i$), the other the part yet to be initialized (indices $\geq i$). In the simplest case, each slice is "smashed" into a single value, but more refined analyses express relationships between slices. Since the slices are segments $[a, b]$ of indices, these analyses generalize poorly to multidimensional arrays. Also, there is often a combinatorial explosion in analyzing how array slices may or may not overlap.

Cornish et al. [9] implement a similar approach by a program-to-program translation over the LLVM intermediate representation, followed by a scalar analysis.

To our best knowledge, all these approaches factor through our Galois connections $\underset{\alpha}{\overset{\gamma}{\leftrightarrows}}$, $\underset{\alpha_{2<}}{\overset{\gamma_{2<}}{\leftrightarrows}}$ or combinations thereof: that is, their abstraction can be expressed as a composition of our abstraction and further abstraction — even though our implementation of the abstract transfer functions is completely different from theirs. Our approach, however, separates the concerns of i) abstracting array problems to array-less problems ii) abstracting the relationships between different cells and indices.

Fluid updates Dillig et al. [13] extend the slice approach by introducing "fluid updates" to overcome the dichotomy between strong and weak updates. They specifically exclude sortedness from the kind of properties they can study.

7.2 Array removal by program transformation

Monniaux and Alberti [25] analyze array programs by transforming them into array-free programs. They use the same Galois connections ($\underset{\alpha}{\overset{\gamma}{\leftrightarrows}}$, $\underset{\alpha_{2\leq}}{\overset{\gamma_{2\leq}}{\leftrightarrows}}$) as us, but i) they implement their analysis by a program-to-program transformation ii) their abstraction is weaker than ours

The Horn clauses corresponding to the encoding of the "read" operation in their approach are the same as ours but with some antecedents dropped, thus over-approximate ours:

$$i \neq k \wedge I_1^\sharp(\mathbf{x}, i, k, a_k) \implies I_2^\sharp(\mathbf{x}, v, i, k, a_k) \tag{54}$$

$$I_1^\sharp(\mathbf{x}, i, i, v) \implies I_2^\sharp(\mathbf{x}, v, i, i, v) \tag{55}$$

Their approach is strictly less precise than ours; for instance it cannot directly prove that an array initialization followed by a loop that checks every element of the array and sets a flag if the element is incorrect never sets its flag. They (Sec. 5.5) are sometimes able to recover the loss of precision induced by their abstractions by applying a form of *quantifier elimination*; we do not need that.

Another difficulty they obviously faced was the limitations of the back-end solvers that they could use. The integer acceleration engine FLATA severely limits the kind of transition relations that can be considered and scales poorly. The abstract interpreter CONCURINTERPROC can infer disjunctive properties (necessary to distinguish two slices in an array) only if given case splits using observer Boolean variables; but the cost increases greatly (exponentially, in the worst case) with the number of such variables.

7.3 Instantiation in Horn clauses

Bjørner et al. [3] propose an approach for solving universally quantified Horn clauses: a Horn clause $(\forall x\ P(x, y)) \rightarrow Q(y)$, not handled by current solvers, is abstracted by $P(x_1(y)) \wedge \cdots \wedge P(x_n(y)) \rightarrow Q(y)$ where the x_i are heuristically chosen instantiations. Our approach can be construed as an application of their approach to the axioms of arrays, with specific instantiation heuristics.

We improve on their interesting contribution in several ways. i) Instead of presenting our approach as a heuristic instantiation scheme, we show that it corresponds to specific Galois connections, which clarifies what abstraction is done and what kind of properties can or cannot be represented.

ii) We handle sortedness properties. None of their examples deal with sortedness and it is unclear how their instantiation heuristics would behave on them.

iii) We handle multisets (and thus permutation properties) by reduction to arrays. It is possible that our approach in this respect can be described as an instantiation scheme over the axioms for arrays (including the multiset of array contents), but, again, it is unclear how their instantiation heuristics would behave in this respect.

Their approach has not been implemented except in private research prototypes; we could not run a comparison.[16]

7.4 Predicate abstraction, CEGAR and array interpolants

There exist a variety of approaches based on counterexample-guided abstraction refinement using *Craig interpolants* [22–24]. In a nutshell, Craig interpolants are predicates suitable

[16] In particular, their approach is *not* implemented in Z3 (personal communication from N. Bjørner).

for proving, using Hoare triples, that some unfolding of the execution cannot lead to an error state. They are typically obtained by reprocessing the proof of unsatisfiability of the unfolding produced by an SMT solver.

Generating good interpolants from purely arithmetic problems is already a difficult problem, and generating good universally quantified interpolants on array properties has proved even more challenging [1, 2, 19].

7.5 Acceleration

It is possible to compute exactly the transitive closure of some transition relations, and thus to summarize some loop exactly. The class of transition relations supported is however restricted.

Bozga et al. [7] have proposed a method for accelerating certain transition relations involving actions over arrays, which outputs the transitive closure in the form of a *counter automaton*. Translating the counter automaton into a first-order formula expressing the array properties however results in a loss of precision.

8. Conclusion and perspectives

We have proposed a generic approach to abstract programs and universal properties over arrays (and, more generally, arbitrary maps) by syntactic transformation into a system of Horn clauses without arrays, which is then sent to a solver. This transformation is powerful enough to prove, fully automatically and within minutes, that the output of selection sort is sorted and is a permutation of the input.

While some solvers have difficulties with the kind of Horn systems that we generate, some (e.g. SPACER) are capable of solving them quite well. We have used the stock version of the solvers, without help from their designers or special tuning, thus higher performance is to be expected in the future. Indeed, we feel the kind of systems we generate would make good benchmarks for Horn solvers. If the solver cannot find the invariants on its own, it can be helped by partial invariants from the user.

As shown by experiments, our approach significantly improves on the procedures currently built in array-capable SMT-solvers, as well as earlier approaches for inferring quantified invariants over arrays, which typically cannot prove sorting algorithms.

Refinement We are investigating an approach for counter-example-based refinement of the analysis. We however have so far not needed it for proving programs correct.

Existentials Our approach cannot infer witnesses for existentials. Future work could include quantifier instantiation heuristics for existentials.

Backward analysis Our rules are for "forward analysis": they express that if configuration is possible at one step during one execution, then some configuration may be possible at the next step during that execution. We thus define a super-set of all states reachable from program initialization, and the desired property is proved if this set is included in the property.

An alternative approach is "backward analysis": find a super-set of the set of all states reachable from a property violation, not intersecting the initial states. A possible research direction would be to derive backward rules and compare their efficiency to that of forward rules.

High-level maps and sets Many programming languages provide libraries for finite maps and (multi)sets. In this article, we have explained how to abstract some, but not all of their features (Sec. 5.1) — for instance we do not provide an iterator for non-integer set element types. Future work should include reviewing their features and common usage in order to design suitable abstractions.

Query-less analysis One advantage of some of earlier approaches (the abstract interpretation ones from Sec. 7.1 and the program transformation from Monniaux and Alberti [25]) is that they are capable of inferring what a program does, or at least a meaningful abstraction of it (e.g. "at the end of this program all cells in the array a contains 42") as opposed to merely proving a property supplied by the user. Our approach can achieve this as well, *provided it is used with a Horn clause solver that does not require queries* and still provides some interesting solution (a query-less Horn problem has a trivial solution: "true" to all predicates).

This Horn clause solver should however be capable of generating disjunctive properties (e.g. $(k < i \wedge a_k = 0) \vee (k \geq i \wedge a_k = 42)$); thus a simple approach by abstract interpretation of the Horn clauses in, say, a sub-class of the convex polyhedra, will not do. We know of no such Horn solver; designing one is a research challenge. Maybe certain partitioning approaches used in sequential program verification [17, 28] may be transposed to Horn clauses.

Objects We have considered simple programs operating over arrays or maps, as opposed to a real-life programming language with objects, references or, horror, pointer arithmetic. Yet, our approach can be adapted to such languages. One can indeed see each object field name in a language such as Java (e.g. `String x;`) as a map from object references to values (here, of type `String`). The reference may be an index (perhaps i if the object is the i-th object allocated) or a more complex record of the site of allocation.

Pointers Languages with pointers, pointer arithmetic and, worse, access to an object of a type through a pointer of an incompatible type (not uncommon in traditional C programming), can be handled by seeing the memory as an array of bytes, but this leads to impractically inefficient analysis. It is however often possible to segment the memory into independent variables (never accessed through pointers, or at least accessed only through pointers at known locations) and a number of disjoint arrays. Our analysis can then be used over these arrays.

References

[1] F. Alberti et al. An extension of lazy abstraction with interpolation for programs with arrays. *Formal Methods in Systems Design*, 45(1):63–109, 2014.

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

[3] N. Bjørner, K. McMillan, and A. Rybalchenko. On solving universally quantified Horn clauses. In *SAS*, pages 105–125, 2013.

[4] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. On solving universally quantified Horn clauses. In *SAS*, pages 105–125, 2013.

[5] Blanchet et al. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207. ACM, 2003. doi: 10.1145/781131.781153.

[6] 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*, number 2566 in LNCS, pages 85–108. Springer, 2002. doi: 10.1007/3-540-36377-7_5.

[7] M. Bozga, P. Habermehl, R. Iosif, F. Konecný, and T. Vojnar. Automatic verification of integer array programs. In *CAV*, pages 157–172, 2009.

[8] A.R. Bradley, Z. Manna, and H.B. Sipma. What's decidable about arrays? In *VMCAI*, pages 427–442, 2006.

[9] J. Robert M. Cornish, Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Analyzing array manipulating programs by program transformation. In *LOPSTR*, 2014.

[10] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992. doi: 10.1093/logcom/2.4.511.

[11] Patrick Cousot and Radhia Cousot. Invited talk: Higher order abstract interpretation. In *IEEE International Conference on Computer Languages*, pages 95–112. IEEE, 1994.

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

[13] Işıl Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In *ESOP*, pages 246–266, 2010.

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

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

[16] J.Y. Halpern. Presburger arithmetic with unary predicates is Π^1_1 complete. *J. Symbolic Logic*, 56(2):637–642, 1991. ISSN 0022-4812. doi: 10.2307/2274706. URL http://dx.doi.org/10.2307/2274706.

[17] J. Henry, D. Monniaux, and M. Moy. Succinct representations for abstract interpretation. In *SAS*, pages 283–299, 2012.

[18] Kryštof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2012. ISBN 978-3-642-31611-1. doi: 10.1007/978-3-642-31612-8_13.

[19] R. Jhala and K.L. McMillan. Array abstractions from proofs. In *CAV*, pages 193–206, 2007.

[20] Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M. Clarke. Automatic abstraction in smt-based unbounded software model checking. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 846–862. Springer, 2013. ISBN 978-3-642-39798-1. doi: 10.1007/978-3-642-39799-8_59.

[21] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 17–34. Springer, 2014. ISBN 978-3-319-08866-2. doi: 10.1007/978-3-319-08867-9_2.

[22] Kenneth L. McMillan. Applications of Craig interpolation to model checking. In *ICATPN*, volume 3536 of *LNCS*, pages 15–16. Springer, 2005. ISBN 3-540-26301-2. doi: 10.1007/11494744_2.

[23] K.L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136, 2006.

[24] K.L. McMillan. Interpolants from Z3 proofs. In *FMCAD*, pages 19–27, 2011.

[25] David Monniaux and Francesco Alberti. A simple abstraction of arrays and maps by program translation. In *Static analysis symposium (SAS)*, Lecture Notes in Computer Science. Springer, 2015. To appear, available at http://arxiv.org/abs/1506.04161.

[26] Mathias Péron. *Contributions to the Static Analysis of Programs Handling Arrays*. Theses, Université de Grenoble, September 2010. URL https://tel.archives-ouvertes.fr/tel-00623697.

[27] Valentin Perrelle. *Analyse statique de programmes manipulant des tableaux*. Theses, Université de Grenoble, February 2013. URL https://tel.archives-ouvertes.fr/tel-00973892.

[28] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007. doi: 10.1145/1275497.1275501.

[29] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for Horn-clause verification. In Natasha Sharygina and Helmut Veith, editors, *Computer-aided verification (CAV)*, volume 8044 of *Lecture Notes in Computer Science*, pages 347–363. Springer, 2013. ISBN 978-3-642-39798-1. doi: 10.1007/978-3-642-39799-8_24.

[30] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Classifying and solving Horn clauses for verification. In Ernie Cohen and Andrey Rybalchenko, editors, *VSTTE 2013, revised selected papers*, volume 8164 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2014. doi: 10.1007/978-3-642-54108-7_1.