Proving array-manipulating programs without arrays

David Monniaux    Laure Gonnord

VERIMAG

2016-12-02 / LIPN
Contents

Introduction

Inductive array invariants

Abstraction

Encoding the problem

Experiments

Multisets

Conclusion
My goal

Proving properties on programs manipulating

- arrays
- generalized maps $\text{domain} \rightarrow \text{codomain}$

Many automatic tools only on scalar programs (integers, reals...)

Goal: use tools for scalar programs to analyze array / map programs.
Possible other applications

Parameterized concurrent systems?

An array of processes

\[ \downarrow \]
all local variables interpreted as arrays

Need distributed systems experts to tell me whether my abstraction could be of any use.
Inductive invariants

```c
int i=0, j=1;
while(i < 1000) {
    i=i+1; j=j+2;
}
assert(j == 2001);
```

Prove the postcondition by induction on the iteration count. What property?

\[ j = 2i + 1 \land 0 \leq i \leq 1000 \]

- holds initially
- if it holds, holds at next iteration
- (conjoined with exit condition) implies the postcondition
As Horn clauses

Find predicate \( I(i, j) \) over \( \mathbb{Z}^2 \) such that

1. \( I(0, 1) \) holds
2. \( \forall i, j \in \mathbb{Z} \ I(i, j) \land i < 1000 \implies I(i + 1, j + 2) \)
3. \( \forall i, j \in \mathbb{Z} \ I(i, j) \land i \geq 1000 \implies j = 2001 \)

Three Horn clauses:

\[ \forall \text{variables}, \ I_{i_1}(\text{arguments}) \land \cdots \land I_{i_n}(\text{arguments}) \]
\[ \land \text{arithmetic condition} \implies I_j(\text{arguments}) \]
Program analysis as solving Horn clauses

Encode the inductiveness condition using Horn clauses.

\[ i < 1000; \ i := i + 1; \ j := j + 2; \]

\[
\begin{align*}
\text{true} & \rightarrow l_0(i, j) \quad (1) \\
l_0(i, j) & \rightarrow l_1(0, 1) \quad (2) \\
l_1(i, j) \land i < 1000 & \rightarrow l_1(i + 1, j + 2) \quad (3) \\
l_1(i, j) \land i \geq 1000 & \rightarrow l_2(i, j) \quad (4) \\
l_2(i, j) \land j \neq 2001 & \rightarrow false \quad (5)
\end{align*}
\]
Decoupling

Traditional static analysis tools (e.g. Astrée) do everything at once.

Here two steps

1. Generate system of Horn clauses from program
2. Solve Horn clauses by whatever method

Step 1 in general more complicated than “take control-flow graph and print out rules”.

- Procedures / function cals
- Extract loops / sub-functions into new functions?
- **Pointers**: pre-analysis to segment memory, a memory segment = an array
Analyzing the Horn clauses

Horn clauses = inductiveness constraints (+ query = “this state is unreachable”)

Without query
- Model-checking
- Abstract interpretation
  - Kleene iterations
  - Kleene iterations with widening
  - policy iteration

With query
- Backward / forward abstract interpretation
- CounterExample-Guided Abstraction Refinement
- IC3 / PDR
Introduction

Tools for solving

Z3  PDR
Spacer  a variant on PDR
Eldarica  CEGAR

Maybe a tool from VERIMAG in the future?
Introduction

Inductive array invariants

Abstraction

Encoding the problem

Experiments

Multisets

Conclusion
Array filling

```c
int t[1000];
for(int i=0; i<1000; i++) t[i] = 42;
```

How to prove this program correct?

\[ \forall 0 \leq k < 1000 \ t[k] = 42 \]
Array filling

```c
int t[1000];
for(int i=0; i<1000; i++) t[i] = 42;
```

How to prove this program correct?

\[
\forall 0 \leq k < 1000 \ t[k] = 42
\]

By induction over the loop counter.
Array filling

```c
int t[1000];
for(int i=0; i<1000; i++) t[i] = 42;
```

How to prove this program correct?

$$\forall 0 \leq k < 1000 \ t[k] = 42$$

By induction over the loop counter.

$$0 \leq i \leq 1000 \land \forall 0 \leq k < i \ t[k] = 42$$
Two parts in inductive invariant

\[ 0 \leq i \leq 1000 \land \forall 0 \leq k < i \ t[k] = 42 \]

**Numerics**  
\[ 0 \leq i \leq 1000 \]  
Can be obtained by many methods!

**Array**  
\[ \forall 0 \leq k < i \ t[k] = 42 \]  
How to get this part?
Find minimum

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

Precondition:

\[ h - l \geq 2 \]

Post condition to prove:

\[ l \leq p < h \]
\[ b = a[p] \]
\[ \forall l \leq k < h, \ b \leq a[k] \]
Find minimum

Auxiliary inductive invariant

\[ l < i < h \]

First two inductive:

\[ l \leq p < h \]
\[ b = a[p] \]

More complicated:

\[ \forall l \leq k < i, \; b \leq a[k] \]
Sorting

```c
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) {
            if (a[i] < b) {
                b = a[i]; p = i;
            }
            i = i+1;
        }
        a[l] = b; a[p] = f;  //swap
        l = l+1;
    }
}
```
Sorting

Postcondition: the output is sorted

\[ \forall l_0 \leq k \leq k' < h, \ a[k] \leq a[k'] \]

Inductive invariant for outer loop

\[ l_0 \leq l < h - 1 \]

\[ \forall k, k' \ l_0 \leq k < l \land k \leq k' < h \ \Rightarrow \ a[k] \leq a[k'] \]
To summarize

To prove properties such as

- initialization
- “all elements are below a bound”

need: a relation between $a[k]$ and program variables, valid $\forall k$

To prove properties such as sortedness

need: a relation between $a[k]$, $a[k']$ and program variables, valid $\forall k, k'$
Arrays in Horn clauses

```plaintext
int t[1000];
for(int i=0; i<1000; i++) t[i] = 42;
```

\[ \text{true} \rightarrow l(0, t) \]
\[ l(i, t) \land i < 1000 \rightarrow (i + 1, \text{update}(t, i, 42)) \]
\[ l(i, t) \land i \geq 1000 \rightarrow E(t) \]
\[ E(t) \land \exists 0 \leq k < 1000 \text{ select}(t, k) \neq 42 \rightarrow \text{false} \]
Contents

Introduction

Inductive array invariants

Abstraction

Encoding the problem

Experiments

Multisets

Conclusion
An abstraction

For variables $\vec{x}$ and array $a$ replace state $(\vec{x}, a)$ by the collection

$$\{(\vec{x}, k, a[k]) \mid k\}$$

Same as replacing a function by its graph!
For a singleton, this abstraction is **faithful.**
Precision loss (for abstraction aficionados)

Two arrays abstracted together.

$a$ constant 33, $b$ constant 42, length 1945, abstraction:

\[
\{(k, 33) \mid 0 \leq k < 1945\} \cup \{(k, 42) \mid 0 \leq k < 1945\}
\]

This abstraction includes e.g.

\[
2k \mapsto 33
\]

\[
2k + 1 \mapsto 42
\]

Cannot specify e.g. “the array is constant”.

[David: DRAW]
For sortedness

For variables $\vec{x}$ and array $a$ replace state $(\vec{x}, a)$ by the collection

$$\{(\vec{x}, k, a[k], k', a[k']) \mid k, k'\}$$

Or, to break symmetries:

$$\{(\vec{x}, k, a[k], k', a[k']) \mid k \leq k'\}$$
Encoding the problem
Method

Scalar variables $x_1, \ldots, x_m$

Arrays $a_1, \ldots, a_n$

- To each program point attach, instead of a set $I$ of concrete states $(x_1, \ldots, x_m, a_1, \ldots, a_n)$, a set $I^\#$ of abstract states $(x_1, \ldots, x_m, k_{1,1}, \ldots, k_{1,n}, a_1^\#, \ldots, k_{1,1}, \ldots, k_{1,n}, a_n^\#)$

- Scalar instructions give rules as usual. Leave $k_{1,1}, \ldots, k_{1,n}, a_1^\#, \ldots, k_{1,1}, \ldots, k_{1,n}, a_n^\#$ untouched.

- Array reads and array writes are specially encoded.
Read statement

\[ k \neq i \land I_1^\#(\vec{x}, i, v, (k, a_k)) \land I_1^\#(\vec{x}, i, v, (i, a_i)) \implies I_2^\#(\vec{x}, i, a_i, (k, a_k)) \]

\[ I_1^\#(\vec{x}, i, v, (i, a_i)) \implies I_2^\#(\vec{x}, i, a_i, (i, a_i)) \]

Note **nonlinear rule**: refers to TWO abstract states in the antecedent.
Encoding the problem

Write statement

\[
\begin{align*}
I_1^#((\vec{x}, i, v), (k, a_k)) \land i \neq k & \implies I_2^#((\vec{x}, i, v), (k, a_k)) \\
I_1^#((\vec{x}, i, v), (i, a_k)) & \implies I_2^#((\vec{x}, i, v), (i, v))
\end{align*}
\]
Contents

Introduction

Inductive array invariants

Abstraction

Encoding the problem

Experiments

Multisets

Conclusion
## Speed

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>N</th>
<th>Z3/PDR</th>
<th>Z3/Spacer</th>
<th>Eldarica</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin_search_check</td>
<td>1</td>
<td>sat</td>
<td>0.71</td>
<td>sat</td>
</tr>
<tr>
<td>find_mini_check</td>
<td>1</td>
<td>sat</td>
<td>4.22</td>
<td>sat</td>
</tr>
<tr>
<td>revrefill1D_check_buggy</td>
<td>1</td>
<td>unsat</td>
<td>0.03</td>
<td>unsat</td>
</tr>
<tr>
<td>array_init_2D</td>
<td>1</td>
<td>sat</td>
<td>0.46</td>
<td>sat</td>
</tr>
<tr>
<td>array_sort_2D</td>
<td>1</td>
<td>sat</td>
<td>0.78</td>
<td>sat</td>
</tr>
<tr>
<td>selection_sort (sortedness)</td>
<td>2</td>
<td>sat*</td>
<td>99.04</td>
<td>timeout(300s)</td>
</tr>
<tr>
<td>selection_sort (sortedness)</td>
<td>2</td>
<td>unsat</td>
<td>83</td>
<td>sat</td>
</tr>
<tr>
<td>selection_sort (permutation)</td>
<td>1</td>
<td>timeout</td>
<td>600</td>
<td>sat</td>
</tr>
<tr>
<td>bubble_sort_simplified</td>
<td>2</td>
<td>sat</td>
<td>5.98</td>
<td>sat</td>
</tr>
<tr>
<td>insertion_sort</td>
<td>2</td>
<td>sat(R1)</td>
<td>53.83</td>
<td>timeout(300s)</td>
</tr>
</tbody>
</table>
Caveats

- Sensitivity to random seed / transformation vagaries
- Bugs
  - crashes
  - different versions of Z3 answer sat / unsat / unknown

Do not pay too much attention to speeds!
Contents

Introduction
Inductive array invariants
Abstraction
Encoding the problem
Experiments
Multisets
Conclusion
Multisets

Sorting

- outputs a sorted array (done)
- a permutation of the input
  \[\iff\text{multiset of the output} = \text{multiset in the input}\]

Multiset of elements of \(X = \text{map } X \rightarrow \mathbb{N}\)
Multisets

Multisets

Sorting

 outputs a sorted array (done)

 a permutation of the input

 \[ \iff \text{multiset of the output} = \text{multiset in the input} \]

Multiset of elements of \( X = \text{map } X \to \mathbb{N} \)

Add to array \( a \) a **ghost variable** \( \hat{a} \)

Update ghost variable when writing to \( a \):

\[
\hat{a}[a[i]] := \hat{a}[a[i]] - 1; \\
\hat{a}[a[i]] := \hat{a}[a[i]] + 1 \\
\]

\[
a[i] := e; \\
\hat{a}[a[i]] := \hat{a}[a[i]] + 1
\]
Multiset analysis process

1. Add for each array (including initial array) \( a : Y \rightarrow X \) a ghost 
   \( \hat{a} : X \rightarrow \mathbb{N} \)

2. Add updates to \( \hat{a} \) to each write to \( a \)

3. Abstract both \( a \) and \( \hat{a} \)

4. Solve with Horn solver backend

Proves e.g. that selection sort permutes the array!

Note: the program needs not be a sequence of explicit swaps, allows optimized sequence of swaps with temporaries.
Contents

Introduction
Inductive array invariants
Abstraction
Encoding the problem
Experiments
Multisets
Conclusion
Conclusion

Arrays

- Success on finding minimum, selection sort, insertion sort etc.
- Back-end scalar Horn solver could not deal with heap sort

http://laure.gonnord.org/pro/demopage/vaphor/
Perspective

(Joint work with L. Gonnord and J. Braine)

- translation from Horn clauses to Horn clauses
- lists and other container classes