Proving array-manipulating programs without arrays

David Monniaux Laure Gonnord

VERIMAG

2016-12-02 / LIPN

Contents

My goal

Proving properties on programs manipulating

- arrays
- generalized maps *domain* $\rightarrow$ *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

```
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?

$$j = 2i + 1 \wedge 0 \leq i \leq 1000$$

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

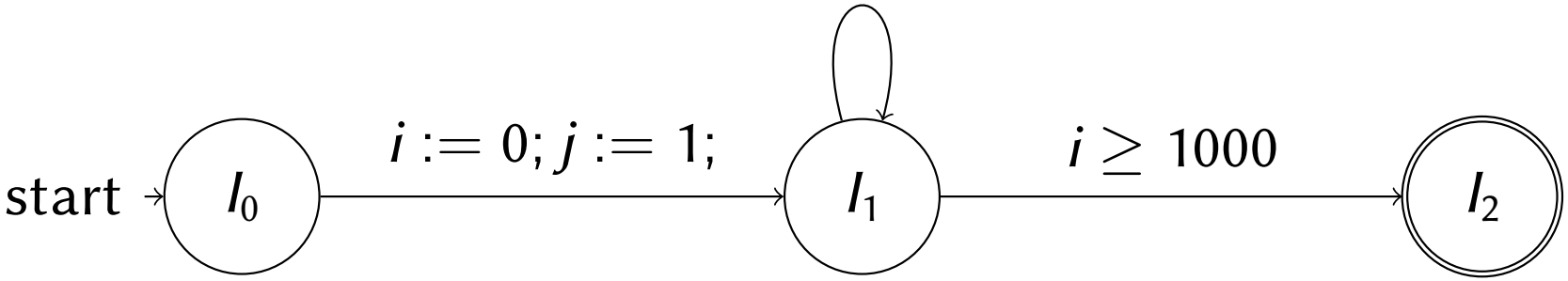
Three **Horn clauses**:

$$\forall variables,\ I_{i_1}(arguments) \wedge \cdots \wedge I_{i_n}(arguments) \\ \wedge\ arithmetic\ condition \implies I_j(arguments)$$

Program analysis as solving Horn clauses

Encode the inductiveness condition using Horn clauses.

$$i < 1000;\ i := i + 1;\ j := j + 2;$$



$$true \rightarrow l_0(i, j) \tag{1}$$

$$l_0(i, j) \rightarrow l_1(0, 1) \tag{2}$$

$$l_1(i, j) \wedge i < 1000 \rightarrow l_1(i + 1, j + 2) \tag{3}$$

$$l_1(i, j) \wedge i \geq 1000 \rightarrow l_2(i, j) \tag{4}$$

$$l_2(i, j) \wedge j \neq 2001 \rightarrow false \tag{5}$$

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
- **C**ounter**E**xample-**G**uided **A**bstraction **R**efinement
- IC3 / PDR

Tools for solving

Z3	PDR
Spacer	a variant on PDR
Eldarica	CEGAR

Maybe a tool from VERIMAG in the future?

Contents

Array filling

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

How to prove this program correct?

$$\forall 0 \leq k < 1000 \; \mathtt{t}[k] = 42$$

Array filling

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

How to prove this program correct?

$$\forall 0 \leq k < 1000 \; \mathtt{t}[k] = 42$$

By induction over the loop counter.

Array filling

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

How to prove this program correct?

$$\forall 0 \leq k < 1000 \; \mathtt{t}[k] = 42$$

By induction over the loop counter.

$$0 \leq \mathtt{i} \leq 1000 \wedge \forall 0 \leq k < \mathtt{i} \; t[k] = 42$$

Two parts in inductive invariant

$$0 \leq \mathrm{i} \leq 1000 \wedge \forall 0 \leq k < \mathrm{i}\ t[k] = 42$$

Numerics $0 \leq \mathrm{i} \leq 1000$
Can be obtained by many methods!

Array $\forall 0 \leq k < \mathrm{i}\ t[k] = 42$
How to get this part?

Find minimum

```
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

```
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) { //find_mini
      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

$$\forall l0 \leq k \leq k' < h,\ a[k] \leq a[k']$$

Inductive invariant for outer loop

$$l0 \leq l < h - 1$$
$$\forall k, k'\ l0 \leq k < l \wedge k \leq k' < h \implies a[k] \leq a[k']$$

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

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

$$
\begin{aligned}
true &\rightarrow I(0, t) \\
I(i, t) \wedge i < 1000 &\rightarrow (i + 1, update(t, i, 42)) \\
I(i, t) \wedge i \geq 1000 &\rightarrow E(t) \\
E(t) \wedge \exists 0 \leq k < 1000\ select(t, k) \neq 42 &\rightarrow false
\end{aligned}
$$

Contents

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:

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

Contents

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 $I^\sharp$ of abstract states $(x_1, \ldots, x_m, k_{1,1}, \ldots, k_{1,n}, a_1^\sharp, \ldots, k_{1,1}, \ldots, k_{1,n}, a_n^\sharp)$
- Scalar instructions give rules as usual.
  Leave $k_{1,1}, \ldots, k_{1,n}, a_1^\sharp, \ldots, k_{1,1}, \ldots, k_{1,n}, a_n^\sharp$ untouched.
- Array reads and array writes are specially encoded.

Read statement

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

$$I_1^\sharp\big((\vec{x}, i, v), (i, a_i)\big) \implies I_2^\sharp\big((\vec{x}, i, a_i), (i, a_i)\big)$$

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

Write statement

$$I_1^\sharp\big((\vec{x}, i, v), (k, a_k)\big) \wedge i \neq k \implies I_2^\sharp\big((\vec{x}, i, v), (k, a_k)\big)$$

$$I_1^\sharp\big((\vec{x}, i, v), (i, a_k)\big) \implies I_2^\sharp\big((\vec{x}, i, v), (i, v)\big)$$

Contents

Speed

Benchmark	N	Z3/PDR		Z3/Spacer		Eldarica	
		Res	Time	Res	Time	Res	Time
bin_search_check	1	sat	0.71	sat	0.34	Crash	
find_mini_check	1	sat	4.22	sat	0.82	sat	110.58
revrefill1D_check_buggy	1	unsat	0.03	unsat	0.07	unsat	9.21
array_init_2D	1	sat	0.46	sat	0.22	sat	12.76
array_sort_2D	1	sat	0.78	sat	0.30	sat	26.68
selection_sort (sortedness)	2	sat*	99.04	timeout(300s)		timeout(300s)	
selection_sort (sortedness)	2	~~unsat~~	83	sat	48	timeout	334
selection_sort (permutation)	1	timeout	600	sat	9.24	timeout	336
bubble_sort_simplified	2	sat	5.98	sat	2.77	sat	158.70
insertion_sort	2	sat(R1)	53.83	timeout(300s)		timeout(300s)	

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

Sorting

Sorting

- outputs a sorted array (done)
- a **permutation** of the input
  $\iff$ multiset of the output = multiset in the input

Multiset of elements of X = map $X \to \mathbb{N}$

Sorting

Sorting

- outputs a sorted array (done)
- a **permutation** of the input
  $\iff$ multiset of the output = multiset in the input

Multiset of elements of X = map $X \to \mathbb{N}$

Add to array a a **ghost variable** $\hat{a}$
Update ghost variable when writing to a:

$$\begin{aligned} \hat{a}[a[i]] &:= \hat{a}[a[i]] - 1; \\ a[i] &:= e; \\ \hat{a}[a[i]] &:= \hat{a}[a[i]] + 1 \end{aligned}$$

Multiset analysis process

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

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