Proving array properties of programs

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

Laure Gonnord and David Monniaux

University of Lyon/ LIP - CNRS/Verimag

January 25th, 2016

Plan

Goal : Array Bound check

```
int main () {
    int v[10] ;
    v[0]=0;
    return v[20]
}
```

```
int a[N]
for(i=0; i<N; i++) {
    a[i] = 42;
}
assertInitialized(a) ;
```

- Index-based verification.

Goal : More, array initialization

```
int a[N]
for(i=0; i<N; i++) {
    a[i] = 42;
}
#forall i,a[i]==42 ;
```

- Relationships between **indices and content**.

Goal : Even more, sortedness

```
int a[N] ;
int i,j,iMin ;

for (j=0;i<n-1;j++){
    iMin=j ;
    for(i=j+1; i<N; i++) {
        if (a[i] < a[iMin])
            iMin=i ;
    }
    swap(a[j],a[imin]) ;
}
assertSorted(a) ;
```

✔

- Sortedness : $\forall i, a[i] \leq a[i+1]$
- Permutation : later.

Invariants with tunable precision

- (array bound) only on indices
- (init) 1 index + content
- (sort) 2 indices + content

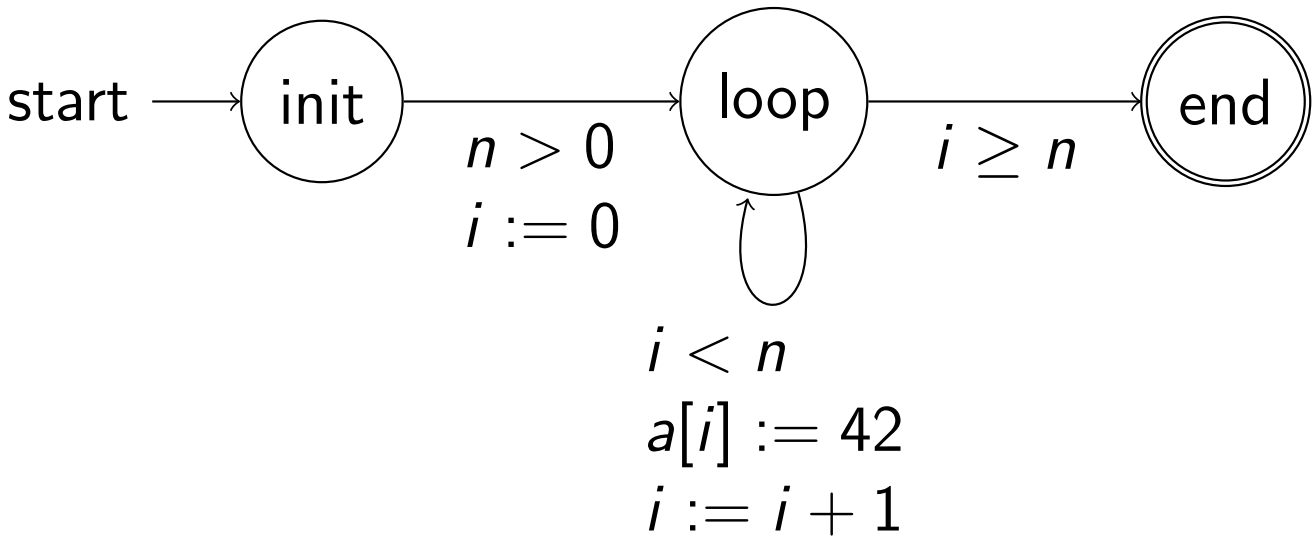
- Need for tunable precision.

Contributions

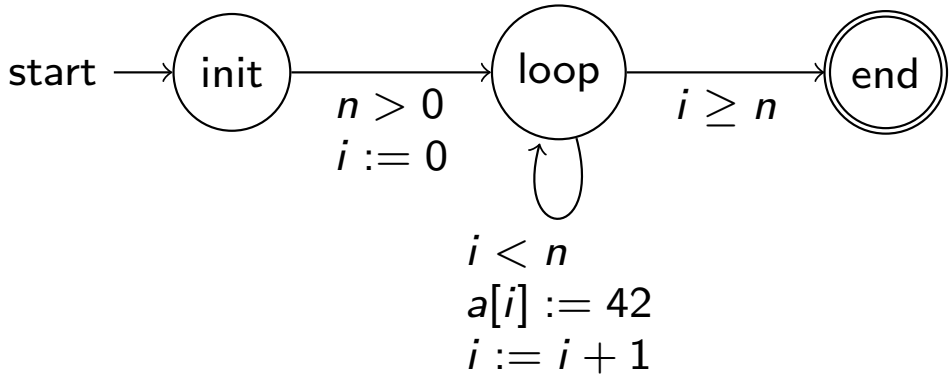
- A new abstraction for C-like programs with arrays:
  - with **tunable** precision.
  - into Horn clauses (a special type of formula) without arrays.
  - extensible to other data structures (maps, . . .).
- Implemented as a standalone tool: VAPHOR
  - Translates a mini language into Horn clauses (SMTLIB FORMAT) ▶ **use your favorite solver as back-end!**
  - Capable of proving standard properties such as initialisation, . . .
  - and . . . sortedness.

Plan

Model of programs - Transition systems



Program verification : 1-abstract semantics



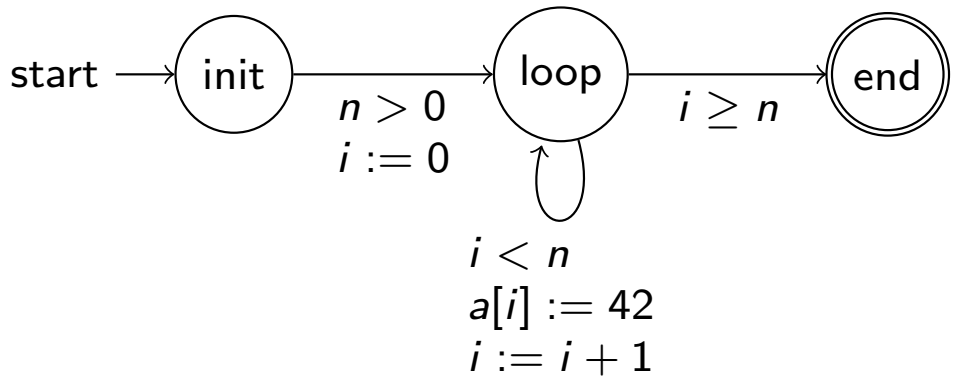
Abstract semantics: ($store(a, i, 42)$ is a where $a[i]$ is 42)

$$\forall n \in \mathbb{Z}\ \forall a \in Array\,(\mathbb{Z}, \mathbb{Z})\ \ n > 0 \implies loop(n, 0, a) \quad (1)$$

$$\forall n, i \in \mathbb{Z}\ \forall a \in Array\,(\mathbb{Z}, \mathbb{Z})\ \ i < n \wedge loop(n, i, a) \implies loop(n, i + 1, \mathbf{store}(a, i, 42)) \quad (2)$$

$$\forall n, i \in \mathbb{Z}\ \forall a \in Array\,(\mathbb{Z}, \mathbb{Z})\ \ i \geq n \wedge loop(n, i, a) \implies end(n, a) \quad (3)$$

Program verification : 2-formula checking



Correctness of the initialisation:

$$\forall n \in \mathbb{Z}\ \forall a \in \mathit{Array}\,(\mathbb{Z}, \mathbb{Z})\ \ 0 \leq k < n \wedge \mathit{end}(n, a) \implies a[k] = 42 \tag{4}$$

Remark : without this goal a trivial solution would be $\mathit{init} = \mathit{loop} = \mathit{end} = \mathit{true}$.

Program Verification as solving Horn Clauses

(Horn clause: $\forall \ldots A \wedge B \wedge C \implies D$)

- Abstract semantics as Horn Clauses :

$$\forall \mathbf{x},\ \mathbf{x} \in \text{initial values} \implies \mathbf{x} \in \text{invar}(i_0)$$
$$\forall \mathbf{x}, \mathbf{x}'\ \mathbf{x} \in \text{invar}(i) \wedge (\mathbf{x}, \mathbf{x}') \in \text{trans}(i, j) \implies \mathbf{x}' \in \text{invar}(j)$$

- Invariants are **unknown**.
- Safety property as Horn Clause.
- SAT $\hookrightarrow$ the property is proven.

On logics with arrays

Expressivity/Decidability:

- Numerical (affine) constraints $+ \exists + \forall$: OK
- Numerical constraints $+ \exists +$ uninterpreted fun : OK
- Numerical constraints $+ \exists +$ uninterpreted fun $+ \forall$: Undec.
- Uninterpreted + ifthenelse $\rightarrow$ Arrays (store/update)

- **Arrays + $\forall$: Undecidable.**

Experimentally

State-of-the-art solvers (Z3/PDR, Z3/Spacer, Eldarica) are really not performant on array support: (demo!)

```
laure@sorlin:~/.../$ z3 array_fill1.smt2
unsat
```

- Let us translate into classical Horn Formula **without arrays**.

Basic Idea

If there is a proof of a safety property of a given program with arrays, it is likely that there exists a proof that can be expressed with simple steps over properties relating only a small number N of array cells.

- Use **parametric distinguished** variables for cells : a_i instead of $a[i]$.
- Only distintinguish variables when needed.

- array init : N=1, sorting N=2, ...
- The size of the formula we will get will depend on this N.

Plan

Motivation and big picture

Intuition

An algorithm to translate “C” into array-free horn clauses

Experimental results

Conclusion

Our language

W.L.O.G. we consider *int arrays* and *int scalar variables*:

$$
\begin{array}{rcl}
stmt & := & x := expr\_int \\
& | & x := a[i] \\
& | & a[j] := y \\
& | & stmt; smt \\
& | & \text{if}(expr)\text{then } stmt \text{ else } stmt \\
& | & \text{while}(expr) \text{ do } stmt
\end{array}
$$

expr = expression with or without array accesses.

- more array expressions with basic program transformation.

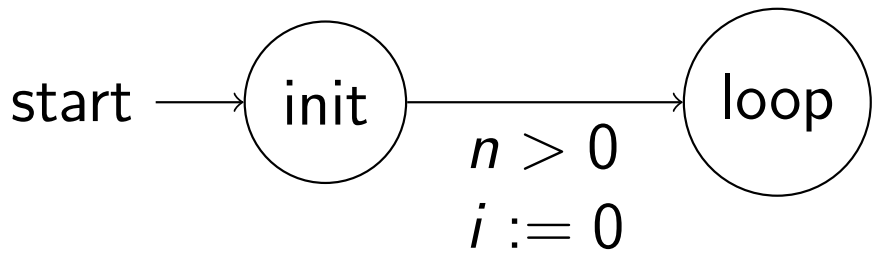
Abstract value

Each node of the CFG is associated to a predicate whose arity is:

- The number of *live* scalar variables.
- $+$ 2N variables (i, ai) to encode parametric cells.

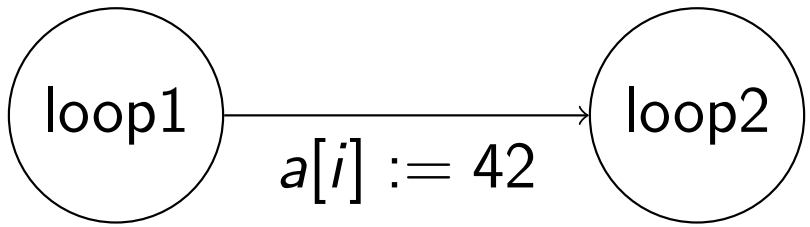
- "we know that at cell i, the value is a_i".

Translation - scalar values : assignments/tests



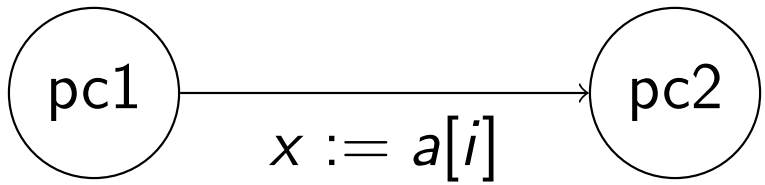
- $\forall ... init(i, n, ...) \wedge n > 0 \Rightarrow loop(0, n, ...)$.

Translation - arrays : write



- $\begin{cases} \forall \ldots loop1(i, n, k, ak) \wedge i \neq k \Rightarrow loop2(i, n, k, ak) \\ \forall \ldots loop1(i, n, i, ak) \Rightarrow loop2(x, i, i, 42) \end{cases}$

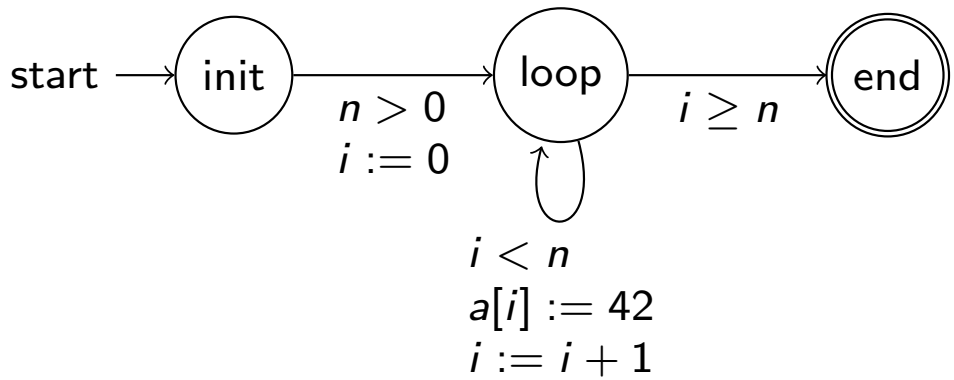
Translation - arrays : read



- (Easy rule) $pc1(i, n, \boldsymbol{x}, i, ai) \Rightarrow pc2(i, n, \boldsymbol{ai}, i, ai)$
- (Less intuitive) $pc1(i, n, \boldsymbol{x}, k, ak) \wedge pc1(i, n, \boldsymbol{x}, i, ai) \wedge k \neq i \Rightarrow pc2(i, n, \boldsymbol{ai}, k, ak)$
(to have $a_i = a[i]$ and $a_k = a[k]$ at pc2 with a given valuation (n, i), then a_i and a_k had to be reachable with the same valuation)

This last rule is **non-linear**.

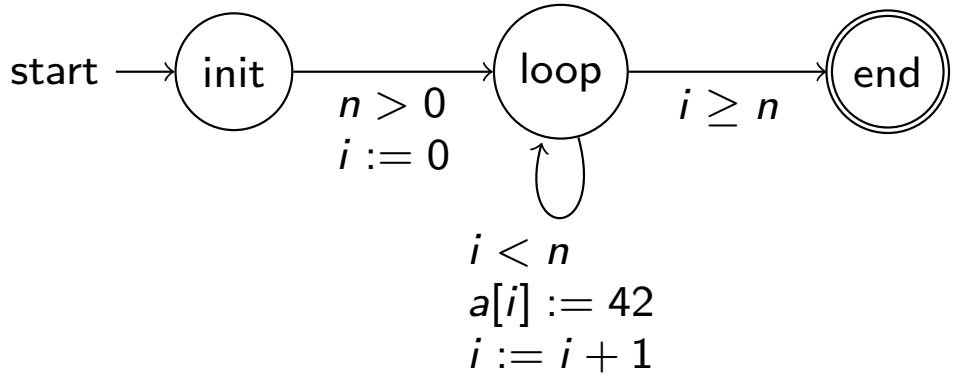
Translation - properties



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

- $0 \leq k < n \wedge \mathit{end}(n, i, k, ak) \Rightarrow a_k = 42$

Translation - on the example



$$0 \leq k < n \implies loop(n, 0, k, a_k)$$

$$0 \leq k < n \wedge i < n \wedge loop(n, i, k, a_k) \implies write(n, i, k, a_k)$$

$$0 \leq k < n \wedge i \neq k \wedge write(n, i, k, a_k) \implies incr(n, i, k, a_k)$$

$$0 \leq k < n \wedge write(n, i, i, a_k) \implies incr(n, i, i, 42)$$

$$0 \leq k < n \wedge incr(n, i, k, a_k) \implies loop(n, i + 1, k, a_k)$$

$$0 \leq k < n \wedge i \geq n \wedge loop(n, i, k, a_k) \implies end(n, i, k, a_k)$$

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

Example : demo

```
laure@sorlin:~/.../$ z3 array_fill2.smt2
sat
```

Translation : N=2 case

Idea: distinguish 2 array cells $a[k_1], a[k_2]$ with $k_1 \leq k_2$:

- Now $a[i] := 42$ translates into:

$$pc_1(\mathbf{x}, i, k_1, a_{k_1}, k_2, a_{k_2}) \wedge i \neq k_1 \wedge i \neq k_2 \implies pc_2(\mathbf{x}, i, k_1, a_{k_1}, k_2, a_{k_2})$$

$$pc_1(\mathbf{x}, i, \boldsymbol{i}, a_{k_1}, k_2, a_{k_2}) \wedge i < k_2 \implies pc_2(\mathbf{x}, i, \boldsymbol{i}, \mathbf{42}, k_2, a_{k_2})$$

$$pc_1(\mathbf{x}, i, k_1, a_{k_1}, \boldsymbol{i}, a_{k_2}) \wedge k_1 < i \implies pc_2(\mathbf{x}, i, k_1, a_{k_1}, \boldsymbol{i}, \mathbf{42})$$

- Sorting property is encoded as :

$$0 < k_1 < k_2 < n \wedge exit(\ldots, k_1, a_{k_1}, k_2, a_{k_2}) \implies a_{k_1} \leq a_{k_2}$$

Translation : we have more

- Complete formalisation for $N > 1$ case.
- 2D arrays, maps...
- Multisets: full proof of sorting algorithms.

- **Functional correctness** of selection/insertion sorts.

Plan

A note about implementation

Our tool, VAPHOR:

- 2k lines of OCAML
- Input : mini-java.
- Output : Horn Clauses as a SMTLIB file.

- Not publicly available yet.

Experimental Setup

- Intel 32 i3-3110M cores, 64 GiB RAM, Ubuntu 14.04 LTS.
- But solvers are not parallel !
- Classical benchmarks of the literature
- Wikipedia version of some sorting algorithms

Nice results on literature classical examples

benchmark	N	Z3/PDR		Z3/SPACER		comments
		result	time	result	time	
append	1	sat	2.36	sat	3.34	
append	-1	sat	31.74	sat	0.16	
find	1	sat	0.26	sat	0.26	
find	-1	sat	0.04	sat	0.03	
partialcopy	1	sat	2.57	sat	0.65	
reverse	1	sat	3.79	sat	2.48	
strcpy	1	sat	4.52	sat	0.54	
swapncopy	1	sat	54.10	timeout	299.01	
memcpy	1	sat	4.05	sat	0.61	
initeven	1	sat	1.32	sat	0.71	+ divisibility constraints

Less nice experiments

benchmark	N	Z3/PDR		Z3/SPACER		comments
		result	time	result	time	
selection_sort (sortedness)	2	sat	200	timeout	600	
selection_sort (sortedness)	2	~~unsat~~	83	sat	48	manual
selection_sort (permutation)	1	timeout	600	sat	9.24	manual
bubble_sort	2	timeout	300	sat	80	
bubble_sort	-1	sat	0	sat	0	
insertion_sort	2	sat	1.26	sat	1.36	
insertion_sort	-1	sat	0	sat	0	

▶ Bugs, execution times highly fluctuant, . . . ▶ Horn solvers are in their early ages.

Plan

Summary

- A method to synthetize abstractions of C programs with arrays into array-free Horn Clauses
- Agnostic about the backend solver
- Capable of proving challenging properties in a reasonable amount of time.

- Available on HAL.

A note on related work

- smashing
- exploding
- use slices

- All these approaches are combinations of our two Galois connections.
- Other related in the paper.

Future work

- Consolidate the prototype.
- Implement some simplifications during code generation, and the extensions to other data structures.
- Write our own Horn Solver !