# Cell morphing: from array programs to array-free Horn clauses

Laure Gonnord and David Monniaux

University of Lyon/ LIP - CNRS/Verimag

SAS 2016 - September 9th, 2016

# Plan

# Goal: Array Bound check

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

- 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

# Contrib. I: array invariants with tunable precision

A new abstraction for C-like programs with arrays:

- with **tunable** precision, the number of "distinguished cells":
  - $N = 0$: for properties on indices
  - $N = 1$: 1 index + content (init)
  - $N = 2$: 2 indices + content (sort)
- into Horn clauses (a special type of formula) without arrays.
- extensible to other data structures (maps, ...).

# Contrib. II: Implementation

This abstraction is 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

# 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 $\mathcal{I}(i, j)$ over $\mathbb{Z}^2$ such that

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

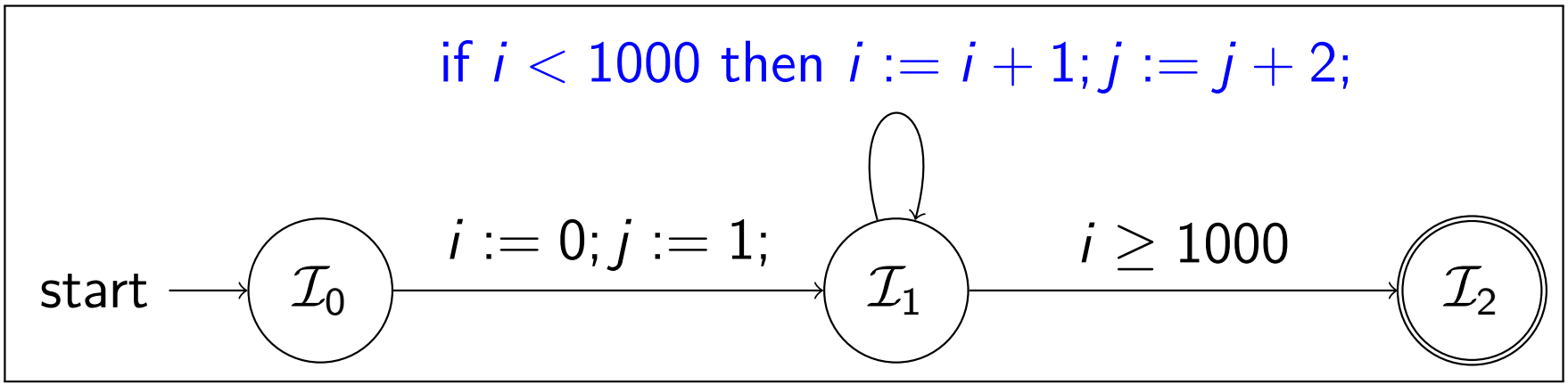
Three **Horn clauses**:

$$\forall \mathit{variables},\ \mathcal{I}_{i_1}(\mathit{arguments}) \wedge \cdots \wedge \mathcal{I}_{i_n}(\mathit{arguments}) \wedge \mathit{arithmetic\ condition} \implies \mathcal{I}_j(\mathit{arguments})$$

(from now we omit $\forall$.)

# Program analysis as solving Horn clauses

Encode the inductiveness condition (+ goal) using Horn clauses, and check sat!




$$true \rightarrow \mathcal{I}_0(i,j) \quad (1)$$

$$\mathcal{I}_0(i,j) \rightarrow \mathcal{I}_1(0,1) \quad (2)$$

$$\mathcal{I}_1(i,j) \wedge i < 1000 \rightarrow \mathcal{I}_1(i+1, j+2) \quad (3)$$

$$\mathcal{I}_1(i,j) \wedge i \geq 1000 \rightarrow \mathcal{I}_2(i,j) \quad (4)$$

$$\mathcal{I}_2(i,j) \wedge j \neq 2001 \rightarrow false \quad (5)$$

# Conclusion : decoupling

Our approach:

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

# Analyzing the Horn clauses

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

- **Without query:**
  - Model-checking
  - Abstract interpretation
- **With query:**
  - Backward / forward abstract interpretation
  - **C**ounter**E**xample-**G**uided **A**bstraction **R**efinement
  - IC3 / PDR
- Tools: Z3 (PDR), Spacer (PDR Variant), Eldarica (CEGAR)

# Plan

# Array filling

```
int t[1000]
for(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.

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

# Two parts in inductive invariant

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

**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? Current version of the solvers (we tried) are unable to synthetise such an invariant. ▶ **abstract arrays with numeric variables**

# Plan

# One-cell abstraction

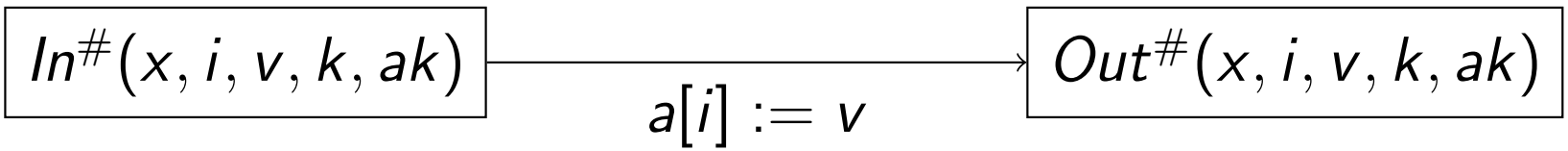
## Main idea

Abstract $a$ of *array* type into a couple $(k, ak = a[k])$

## Algorithm

- To each program point attach, instead of a set $\mathcal{I}$ of concrete states $(x_1, \ldots, x_m, a)$, a set $\mathcal{I}^\#$ of abstract states $(x_1, \ldots, x_m, \mathbf{k}, \mathbf{ak})$
- Scalar instructions give rules as usual. Leave new variables untouched.
- Array store and array select are specially encoded.
- Proof goals are encoded accordingly.

# Cell Abstraction, Store $(a[i] := v)$ 1/2




"The value at array index $i$ is now v, the rest is unchanged"

$$In^{\#}(x, i, v, k, ak) \wedge i \neq k \implies Out^{\#}(x, i, v, k, ak)$$

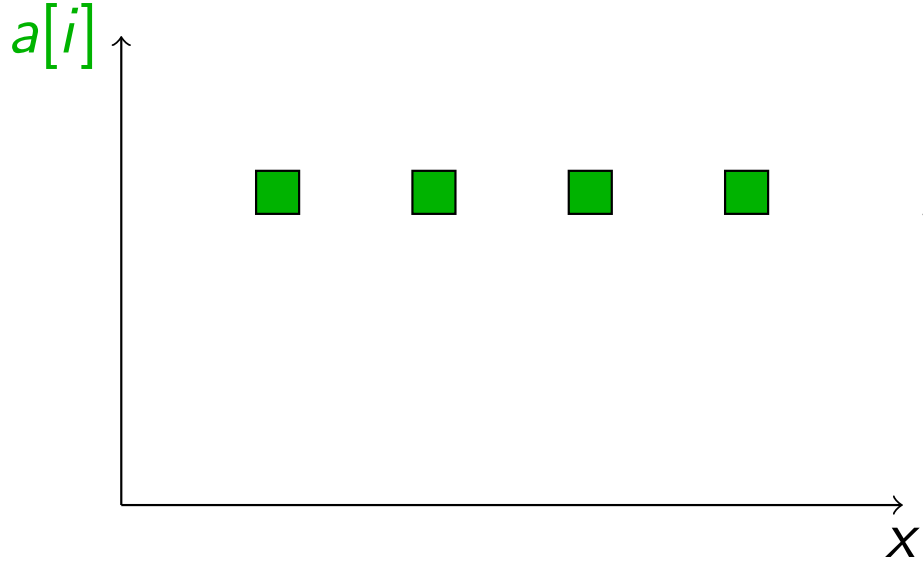
$$In^{\#}(x, \boldsymbol{i}, v, \boldsymbol{i}, ak) \implies Out^{\#}(x, i, v, \boldsymbol{i}, \boldsymbol{v})$$

# Cell Abstraction, Store $(a[i] := v)$ 2/2

$$In^{\#}(x, i, v, k, ak) \wedge i \neq k \implies Out^{\#}(x, i, v, k, ak)$$

$$In^{\#}(x, \boldsymbol{i}, v, \boldsymbol{i}, ak) \implies Out^{\#}(x, i, v, \boldsymbol{i}, \boldsymbol{v})$$

Example



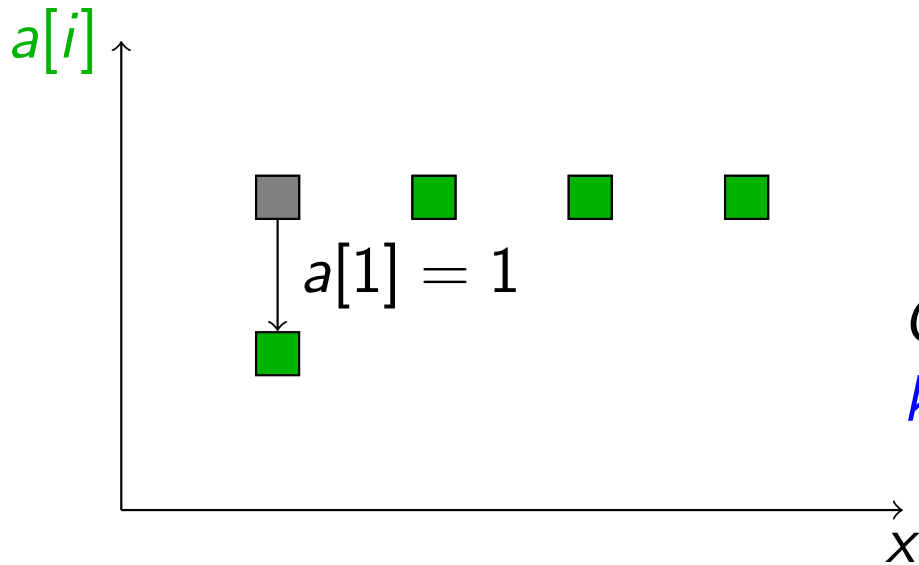

$In^{\#}: 1 \leq k \leq 4 \implies ak = 2$

# Cell Abstraction, Store $(a[i] := v)$ 2/2

$$In^{\#}(x, i, v, k, ak) \wedge i \neq k \implies Out^{\#}(x, i, v, k, ak)$$

$$In^{\#}(x, \boldsymbol{i}, v, \boldsymbol{i}, ak) \implies Out^{\#}(x, i, v, \boldsymbol{i}, \boldsymbol{v})$$

Example

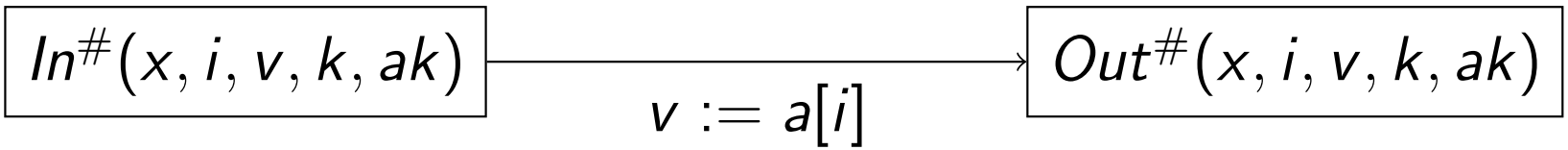



$In^{\#}: 1 \leq k \leq 4 \implies ak = 2$

$Out^{\#}: 2 \leq k \leq 4 \implies ak = 2$

$k = 1 \implies ak = 1$

# Cell Abstraction, Select ($v := a[i]$) 1/2




"v has a new value and do not forget the previous invariants"

$$(k = i) \wedge In^{\#}(x, i, v, i, ai) \implies Out^{\#}(x, i, \mathbf{ai}, i, ai)$$

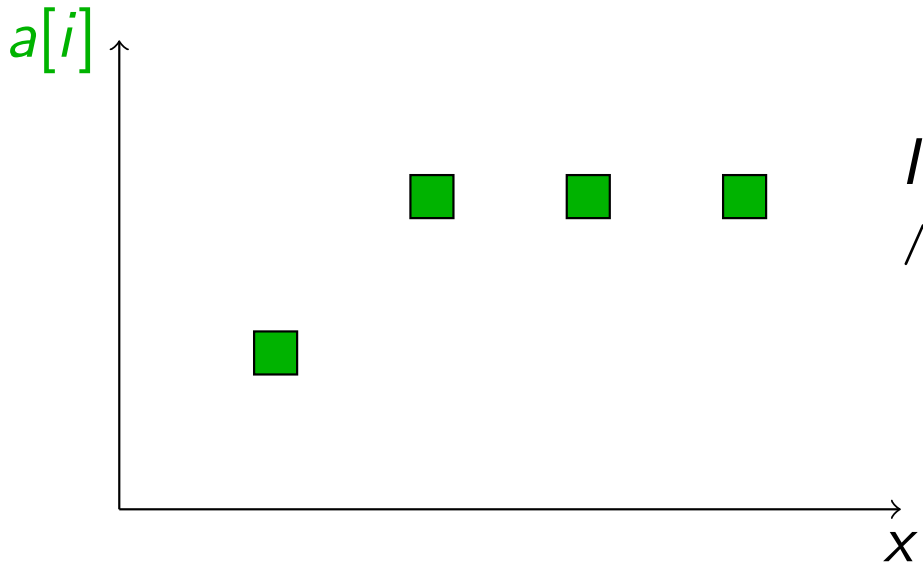
$$k \neq i \wedge In^{\#}(x, i, v, k, ak) \wedge In^{\#}(x, i, v, i, ai) \implies Out^{\#}(x, i, \mathbf{ai}, k, ak)$$

- **Non linear rules**: refers to two abstract states in the antecedent.

# Cell Abstraction, Select $(v := a[i])$ 2/2

$$(k = i) \wedge In^{\#}(x, i, v, i, ai) \implies Out^{\#}(x, i, \mathbf{ai}, i, ai)$$

$$k \neq i \wedge In^{\#}(x, i, v, k, ak) \wedge In^{\#}(x, i, v, i, ai) \implies Out^{\#}(x, i, \mathbf{ai}, k, ak))$$



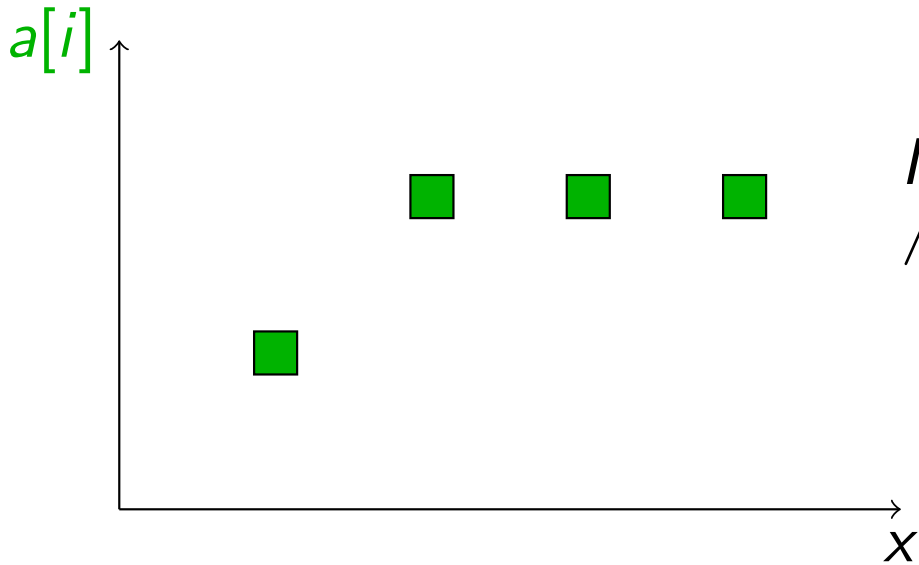

$In^{\#}$: $2 \leq k \leq 4 \Rightarrow ak = 2$
$\wedge k = 1 \Rightarrow ak = 1$

$\boldsymbol{v := a[1]}$

# Cell Abstraction, Select $(v := a[i])$ 2/2

$$(k = i) \wedge In^{\#}(x, i, v, i, ai) \implies Out^{\#}(x, i, \mathbf{ai}, i, ai)$$

$$k \neq i \wedge In^{\#}(x, i, v, k, ak) \wedge In^{\#}(x, i, v, i, ai) \implies Out^{\#}(x, i, \mathbf{ai}, k, ak))$$




$In^{\#}$: $2 \leq k \leq 4 \Rightarrow ak = 2$
$\wedge k = 1 \Rightarrow ak = 1$

$\boldsymbol{v := a[1]}$

- $Out^{\#} : (k = 1 \Rightarrow ak = 1 \wedge v = 1) \wedge (k \neq 1 \Rightarrow \ldots \wedge v = 1)$

# Extensions

- N-cell abstraction (proves **sortedness**): $(x, k, ak, k', ak)$.
- Multisets (prove **permutation**): use ghost variables:
  - Attach a **ghost variable** $\hat{a}$ s.t. $\hat{a}[i] = card\{k, a[k] = i\}$
  - Rewrite each store $a[i] := e;$ into:

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

  - Use the former abstraction for $a$ and $\hat{a}$.

# Plan

# A note about implementation

Our tool, VAPHOR:

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

- Demo page:
`http://laure.gonnord.org/pro/demopage/vaphor/`.

# 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.

# Experimental results

| 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/simpl.) | 2 | ~~unsat~~ | 83 | sat | 48 | timeout(300s) | |
| selection_sort (permutation) | 1 | timeout(300s) | | sat | 9.24 | timeout(300s) | |
| bubble_sort_simplified | 2 | sat | 5.98 | sat | 2.77 | sat | 158.70 |
| insertion_sort | 2 | sat* | 53.83 | timeout(300s) | | timeout(300s) | |

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

Do not pay too much attention to speeds!

# 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.

# A note on related work

- smashing
- exploding
- use slices

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

# Work in Progress

(Joint work with D. Monniaux and J. Braine)

- Consolidate the prototype: our translation is now from Horn clauses to Horn clauses.
- Implement some simplifications during code generation, and the extensions to other data structures.
- Lists and other container classes.