

# Abstraction Refinement of Linear Programs with Arrays

A. Armando<sup>1</sup>, M. Benerecetti<sup>2</sup>, J. Mantovani<sup>1</sup>

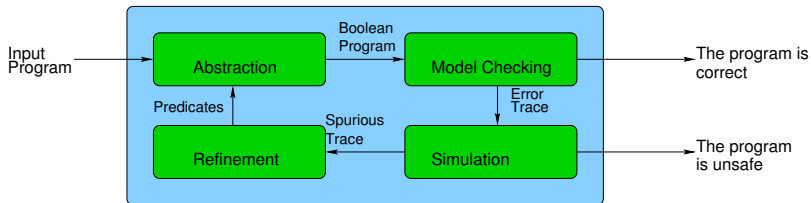
<sup>1</sup>Artificial Intelligence Laboratory, University of Genova

<sup>2</sup>Section of Computer Science, University of Napoli "Federico II"

March 28, 2007



Counterexample-guided Abstraction Refinement has proven to be very effective on specific application domains such as device drivers.



Unfortunately, the CEGAR approach does not apply successfully to programs that involve array manipulation.

### Example program

```
void main() {
  int i, a[30];
  a[1] = 9;
  i = 0;
  while(a[i] != 9) {
    a[i] = 2*i;
    i = i+1;
  }
  assert(i <= 1);
}
```

### Blast analysis

Error found! The system is unsafe :-)

## Contribution

We present an abstraction refinement procedure for Linear Programs with arrays

## Linear Programs with Arrays

- usual control-flow constructs (`if`, `while`, etc.)
- variables and array elements range over a numeric domain
- expressions involve linear combinations of variables and array elements.

Existing Procedures for Linear Programs:

- EUREKA Model Checker for Linear Programs by Armando et al.
- STING invariant generator by Sankaranarayanan et al.
- Action Language Verifier by Bultan et al.
- BLAST, SLAM, etc.

## Contribution

We present an abstraction refinement procedure for Linear Programs with arrays

## Linear Programs with Arrays

- usual control-flow constructs (`if`, `while`, etc.)
- variables and array elements range over a numeric domain
- expressions involve linear combinations of variables and array elements.

Existing Procedures for Linear Programs:

- EUREKA Model Checker for Linear Programs by Armando et al.
- STING invariant generator by Sankaranarayanan et al.
- Action Language Verifier by Bultan et al.
- BLAST, SLAM, etc.

## Contribution

We present an abstraction refinement procedure for Linear Programs with arrays

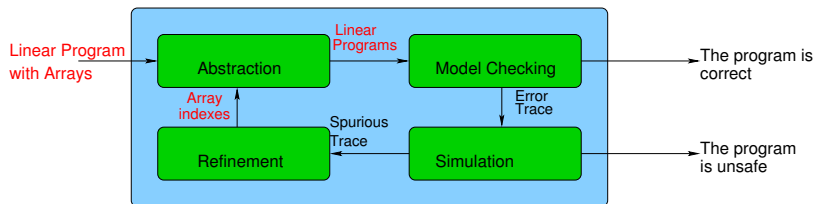
## Linear Programs with Arrays

- usual control-flow constructs (`if`, `while`, etc.)
- variables and array elements range over a numeric domain
- expressions involve linear combinations of variables and array elements.

Existing Procedures for Linear Programs:

- EUREKA Model Checker for Linear Programs by Armando et al.
- STING invariant generator by Sankaranarayanan et al.
- Action Language Verifier by Bultan et al.
- BLAST, SLAM, etc.

Instead of using **predicates**, our abstraction and refinement use **array indexes**.



# Abstraction

## Abstraction of expressions

Let  $R$  be a function mapping each array  $a$  into a subset of  $\{0, 1, \dots, \text{size}(a) - 1\}$ . Given any linear expression with arrays  $e$  in  $P$ ,  $\hat{e}$  is obtained from  $e$  by replacing

- **Array accesses:**  $a[e] \rightsquigarrow \text{abs}(a[e], [k_1, \dots, k_n])$ , where

$$\text{abs}(a[e], []) = u$$

$$\text{abs}(a[e], [k_1, k_2, \dots, k_n]) = (\hat{e} == k_1 ? a^{k_1} : \text{abs}(a[e], [k_2, \dots, k_n])),$$

and  $[k_1, \dots, k_n]$  is some permutation of  $R(a)$ .

$a^k$  is a new abstract variable representing in  $\hat{P}$  element  $a[k]$ .

# Abstraction

## Abstraction of expressions

Example:

Let  $e = a[x + 1] + y$ , and let  $R(a) = \{3\}$ . Then,

$$\hat{e} = ((x + 1 == 3) ? a^3 : u) + y$$

# Abstraction

## Abstraction of assignments

...and:

- **Assignment to an array element**

$$a[i] = e; \rightsquigarrow \begin{cases} a^{k_1}, \dots, a^{k_n} = (\hat{i} == k_1 ? \hat{e} : a^{k_1}), \dots, (\hat{i} == k_n ? \hat{e} : a^{k_n}); & \text{if } n \neq 0 \\ ; & \text{if } n = 0 \end{cases}$$

Example:

Let  $R(a) = \{1, 5\}$ . Then  $a[x + 1] = y$  is abstracted by

$$a^1, a^5 = (x + 1 == 1 ? y : a^1), (x + 1 == 5 ? y : a^5);$$

# Example Abstraction

## Concrete Program $P$

```
void main() {
    int i, a[30];
[1]  a[1] = 9;
[2]  i = 0;
[3]  while(a[i]!=9) {
[4]    a[i] = 2*i;
[5]    i = i+1;
    }
[6]  assert(i<=1);
}
```

## Abstraction of $P$ w.r.t. $R(a) = \emptyset$

```
void main() {
    int i;
[1]  ;
[2]  i = 0;
[3]  while(u!=9) {
[4]    ;
[5]    i = i+1;
    }
[6]  assert(i<=1);
}
```

Abstraction of  $P$  w.r.t.  $R(a) = \emptyset$

Abstract error trace  $\tau$ :

1, 2,  
3, 4, 5,  
3, 4, 5,  
3, 6, 0

```
void main() {  
    int i;  
[1]    ;  
[2]    i = 0;  
[3]    while(u!=9) {  
[4]        ;  
[5]        i = i+1;  
        }  
[6]    assert(i<=1);  
    }
```

# Checking Trace Feasibility

We reduce the **trace feasibility** problem to the **satisfiability** of a set of quantifier-free formulae in the decidable theory resulting from the combination of Linear Arithmetic and the theory of arrays.

- Let  $\tau = i_0, i_1, \dots, i_n$  be the abstract trace
- Let  $s_i$  the concrete statement corresponding to node  $i$  of the CFG.

Then,

- We put the sequence  $s_{i_0}, \dots, s_{i_n}$  in Single Assignment Form.
- We build the set of formulae  $\Phi(\tau, P) = \bigcup_{k=1}^n \phi(s_{i_k})$ , where

$s_{i_k}$	$\phi(s_{i_k})$	condition
<code>if(c), assert(c); while(c)</code>	$\{c\}$	if $i_{k+1} = Tsucc_P(i_k)$
<code>if(c), assert(c); while(c)</code>	$\{\neg c'\}$	if $i_{k+1} = Fsucc_P(i_k)$
<code>assume(c);</code>	$\{c'\}$	if $i_{k+1} = Tsucc_P(i_k)$
<code>v<sub>j+1</sub> = e;</code>	$\{v_{j+1} = e'\}$	
<code>a<sub>j+1</sub>[e<sub>1</sub>] = e<sub>2</sub>;</code>	$\{a_{j+1} = store(a_j, e'_1, e'_2)\}$	

where  $e'_1, e'_2, c'$  are obtained from  $e_1, e_2, c$  by substituting  $a_j[e]$  with `select(aj, e)`.

# Checking Trace Feasibility

We reduce the **trace feasibility** problem to the **satisfiability** of a set of quantifier-free formulae in the decidable theory resulting from the combination of Linear Arithmetic and the theory of arrays.

- Let  $\tau = i_0, i_1, \dots, i_n$  be the abstract trace
- Let  $s_i$  the concrete statement corresponding to node  $i$  of the CFG.

Then,

- We put the sequence  $s_{i_0}, \dots, s_{i_n}$  in Single Assignment Form.
- We build the set of formulae  $\Phi(\tau, P) = \bigcup_{k=1}^n \phi(s_{i_k})$ , where

$s_{i_k}$	$\phi(s_{i_k})$	condition
<code>if(c), assert(c) ; while(c)</code>	$\{c\}$	if $i_{k+1} = Tsuccp(i_k)$
<code>if(c), assert(c) ; while(c)</code>	$\{\neg c'\}$	if $i_{k+1} = Fsuccp(i_k)$
<code>assume(c) ;</code>	$\{c'\}$	if $i_{k+1} = Tsuccp(i_k)$
<code>v<sub>j+1</sub> = e ;</code>	$\{v_{j+1} = e'\}$	
<code>a<sub>j+1</sub>[e<sub>1</sub>] = e<sub>2</sub> ;</code>	$\{a_{j+1} = store(a_j, e'_1, e'_2)\}$	

where  $e'_1, e'_2, c'$  are obtained from  $e_1, e_2, c$  by substituting  $a_j[e]$  with `select(aj, e)`.

# Checking Trace Feasibility

Line	Original Statement	Renamed Statement	$\Phi(\tau, P)$
[1]	<code>a[1] = 9 ;</code>	<code>a<sub>1</sub>[1] = 9 ;</code>	$a_1 = \text{store}(a_0, 1, 9)$
[2]	<code>i = 0 ;</code>	<code>i<sub>1</sub> = 0 ;</code>	$i_1 = 0$
[3]	<code>while(a[i] != 9)</code>	<code>while(a<sub>1</sub>[i<sub>1</sub>] != 9)</code>	$\text{select}(a_1, i_1) \neq 9$
[4]	<code>a[i] = 2 * i ;</code>	<code>a<sub>2</sub>[i<sub>1</sub>] = 2 * i<sub>1</sub> ;</code>	$a_2 = \text{store}(a_1, i_1, 2 * i_1)$
[5]	<code>i = i + 1 ;</code>	<code>i<sub>2</sub> = i<sub>1</sub> + 1 ;</code>	$i_2 = i_1 + 1$
[3]	<code>while(a[i] != 9)</code>	<code>while(a<sub>2</sub>[i<sub>2</sub>] != 9)</code>	$\text{select}(a_2, i_2) \neq 9$
[4]	<code>a[i] = 2 * i ;</code>	<code>a<sub>3</sub>[i<sub>2</sub>] = 2 * i<sub>2</sub> ;</code>	$a_3 = \text{store}(a_2, i_2, 2 * i_2)$
[5]	<code>i = i + 1 ;</code>	<code>i<sub>3</sub> = i<sub>2</sub> + 1 ;</code>	$i_3 = i_2 + 1$
[3]	<code>while(a[i] != 9)</code>	<code>while(a<sub>3</sub>[i<sub>3</sub>] != 9)</code>	$\neg(\text{select}(a_3, i_3) \neq 9)$
[6]	<code>assert(i &lt;= 1) ;</code>	<code>assert(i<sub>3</sub> &lt;= 1) ;</code>	$\neg(i_3 \leq 1)$

# Checking Trace Feasibility

The proof  $\Pi$  of unsatisfiability is the proof of the sequent

$$\Phi(\tau, P) \vdash \perp$$

in the combined theory of Linear Arithmetic and the theory of arrays.

$$\frac{\frac{\frac{\Phi(\tau, P) \vdash \phi_1 \quad \Phi(\tau, P) \vdash \phi_2}{\dots}}{\dots} \quad \Phi(\tau, P) \vdash \phi_3}{\dots} \quad \Phi(\tau, P) \vdash \phi_4}{\dots} \quad \Phi(\tau, P) \vdash \phi_5}{\dots} \quad \Phi(\tau, P) \vdash \phi_6}{\dots} \quad \Phi(\tau, P) \vdash \phi_7}{\dots} \quad \Phi(\tau, P) \vdash \perp$$

where  $\phi_i \in \Phi(\tau, P)$ .



- If the concrete trace is unfeasible, we want to make the same trace unfeasible in the refined program  $\hat{P}'$ .
- The computation of  $R'$  is based on the idea of turning  $\Pi$  into a proof of the unsatisfiability of  $\Phi(\tau, \hat{P}')$ .
- Idea: to “capture” all necessary accesses to array elements within the proof.

We add to the premises of each leaf sequent  $\Phi(\tau, P) \vdash \varphi$  of  $\Pi$  a formula  $Q(e, a)$  for each term of the form  $\text{select}(a_k, e)$  occurring in  $\varphi$ .

$$\begin{array}{c}
 \frac{Q(i_2, a) \vdash \text{select}(a_2, i_2) \neq 9 \vdash i_2 = i_1 + 1}{\vdash \text{select}(a_2, i_1 + 1) \neq 9} \quad \vdash a_2 = \text{store}(a_1, i_1, 2 * i_1)}{\vdash \text{select}(\text{store}(a_1, i_1, 2 * i_1), i_1 + 1) \neq 9} \quad (\text{ArrayAx}) \\
 \frac{\vdash (i_1 + 1 = i_1 ? 2 * i_1 : \text{select}(a_1, i_1 + 1)) \neq 9}{\vdash \text{select}(a_1, i_1 + 1) \neq 9} \quad \vdash i_1 = 0}{\vdash \text{select}(a_1, 1) \neq 9} \quad \vdash a_1 = \text{store}(a_0, 1, 9)}{\vdash \text{select}(\text{store}(a_0, 1, 9), 1) \neq 9} \quad (\text{ArrayAx}) \\
 \frac{\vdash (1 = 1 ? 9 : \text{select}(a_0, 1)) \neq 9}{\vdash \perp}
 \end{array}$$

$Q(e, a)$  is a placeholder for the formula  $\bigvee_{k \in R'(a)} e = k$ . However, since  $R'(a)$  is unknown at this stage, we use  $Q(e, a)$  in place of its expanded version.

The sequent tree obtained in this way is then updated by re-applying all the inference rules of  $\Pi$  on the new leaf sequents.

$$\begin{array}{c}
 \frac{Q(i_2, a) \vdash \text{select}(a_2, i_2) \neq 9 \vdash i_2 = i_1 + 1}{Q(i_1 + 1, a) \vdash \text{select}(a_2, i_1 + 1) \neq 9 \quad \vdash a_2 = \text{store}(a_1, i_1, 2 * i_1)} \\
 \frac{Q(i_1 + 1, a) \vdash \text{select}(\text{store}(a_1, i_1, 2 * i_1), i_1 + 1) \neq 9}{Q(i_1 + 1, a) \vdash (i_1 + 1 = i_1 ? 2 * i_1 : \text{select}(a_1, i_1 + 1)) \neq 9} \quad (\text{ArrayAx}) \\
 \frac{Q(i_1 + 1, a) \vdash \text{select}(a_1, i_1 + 1) \neq 9 \quad \vdash i_1 = 0}{Q(1, a) \vdash \text{select}(a_1, 1) \neq 9 \quad \vdash a_1 = \text{store}(a_0, 1, 9)} \\
 \frac{Q(1, a) \vdash \text{select}(\text{store}(a_0, 1, 9), 1) \neq 9}{Q(1, a) \vdash (1 = 1 ? 9 : \text{select}(a_0, 1)) \neq 9} \quad (\text{ArrayAx}) \\
 \frac{Q(1, a) \vdash (1 = 1 ? 9 : \text{select}(a_0, 1)) \neq 9}{Q(1, a) \vdash \perp}
 \end{array}$$

This leaves us with a sequent tree  $\Pi'$  whose root sequent is of the form

$$\Phi(\tau, P), Q(e'_1, a), \dots, Q(e'_q, a) \vdash \perp,$$

where  $a \in A_P$ .

We choose  $R'$  in such a way that

$$R'(a) = R(a) \cup \bigcup_{i=1, \dots, q} \{e'_i\}$$

if  $e'_1, \dots, e'_q$  are numeric constants, and  $R' = \{0, 1, \dots, \text{size}(a) - 1\}$  otherwise.

## Original program

```
void main(){
  int i, a[30];
  a[1] = 9;
  i = 0;
  while(a[i]!=9){
    a[i] = 2*i;
    i = i+1;
  }
  assert(i<=1);
}
```

## Abstract program w.r.t. $R(a) = \{1\}$

```
void main() {
  int i, a1;
  a1 = (1==1)?9:a1;
  i = 0;
  while(((i==1)?a1:u)!=9) {
    a1 = (i==1)?2*i:a1;
    i = i+1;
  }
  assert(i<=1);
}
```

## Soundness

If our CEGAR procedure returns SAFE, then the input program has no error trace.

## (Relative) Completeness

If the input program has no error trace and all calls to the model checking procedure terminate, then our procedure terminates and returns SAFE.

# Experiments

In order to assess the effectiveness of the approach we have developed in EUREKA a prototype implementation of the CEGAR procedure outlined in this talk.

- Feasibility checks are done by CVC Lite
- We have run EUREKA against C programs featuring a non trivial interplay between array manipulation and arithmetic.
- The benchmarks are families of programs parametric in a positive integer  $N$ : the size of the arrays in the programs and/or the number of loop iterations increase as  $N$  increases.

Benchmark	EUREKA			BLAST	SATABS	
	$N$	(Time)	refined/total array elements	$N$ (Time)	$N$	(Time)
String copy	1000*	(153.78)	1/2 $N$	Incorrect	10	(144.69)
Gray code	25	(230.26)	16/28	Incorrect	Inconclusive	
Partition	40	(178.02)	1/ $N$	Incorrect	Inconclusive	
Bubble sort	8	(91.92)	$N/N$	Incorrect	2	(30.42)
Selection sort	6	(104.42)	$N/N$	Incorrect	2	(115.86)

## Summing up:

- We have proposed an alternative abstraction and refinement schema for a subset of the C programming language that employs linear arithmetic and arrays.
- Our procedure uses **array indexes** instead of **predicates**.
- Our implementation called EUREKA shows largely favourable results when compared with classic CEGAR approaches.

## Future work:

- Embedding of classic CEGAR into our abstraction/refinement.
- Comparison with interpolation in refinement.

Thank you!