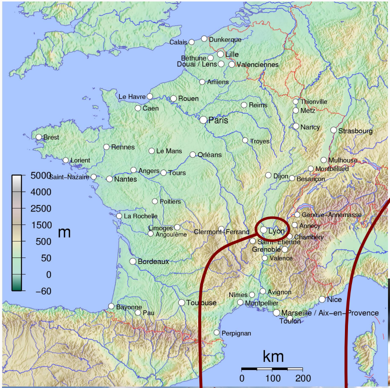
Proving array properties of programs

Experiences in Program Analysis and Compilation

Laure Gonnord

December 3rd, 2020

University Claude Bernard Lyon 1 / LIP, Lyon, France

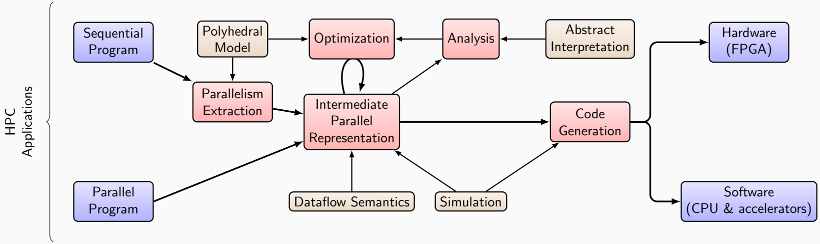


LIP:
Laboratoire
de l'Informatique du
Parallélisme

CASH : Topics + People

Optimized (software/hardware) compilation for HPC software with data-intensive computations.

⇝ Means : dataflow IR, **static analyses**, optimisations, simulation.



Christophe Alias, Laure Gonnord, Ludovic Henrio, Matthieu Moy

+ (2020) Gabriel Radanne + Yannick Zakowski

`http://www.ens-lyon.fr/LIP/CASH/`

Plan

Software needs safety and performance

- For safety-critical systems . . .
- **and** general purpose systems !

▶ Programs crash because of array out-of-bounds accesses, complex pointer behaviour, . . .

Goal : safety - ex

Prove that (some) memory accesses are safe :

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

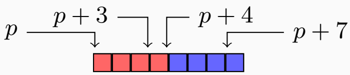
- This program has an illegal array access.

Goal : performance -ex

Enable loop parallelism :

```c
void fill_array (char *p){
   unsigned int i;
   for (i=0; i<4; i++)
        *(p + i) = 0 ;
   for (i=4; i<8; i++)
        *(p + i) = 2*i ;
}
```

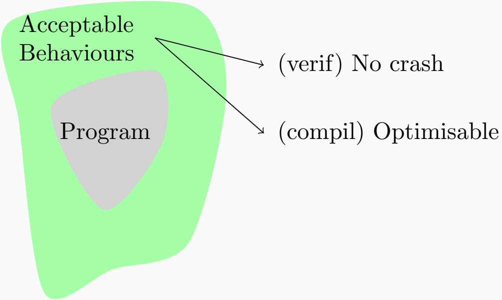
Parallel loops



- The two regions do not overlap.

Proving non trivial properties of software

- Basic idea : software has **mathematically defined behaviour**.
- **Automatically** prove properties.



There is no free lunch

i.e. no magical static analyser. It is impossible to prove interesting properties :

- automatically
- exactly
- on unbounded programs

There is no free lunch

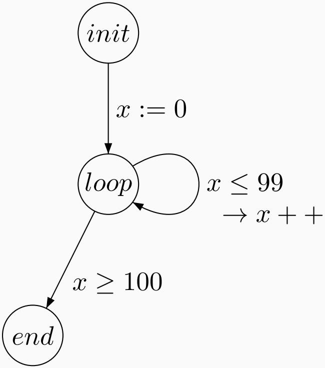
i.e. no magical static analyser. It is ~~im~~possible to prove interesting properties :

- automatically
- ~~exactly~~ with false positives !
- on unbounded programs

▶ **Abstractions** = conservative approximations.

Plan

Computing (inductive) invariants



- $\{x \in \mathbb{N}, 0 \leq x \leq 100\}$ is the most precise invariant in control point `loop`.

Problems and solution

We want to :

- Compute infinite sets.
- In finite time.

▶ How ?

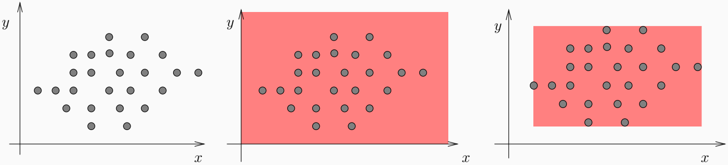
- Approximate sets (abstract domains), compute in this abstract world.
- Extrapolate (widening).

Main ingredient : abstract values

Idea : represent values of variables :

$$R_{pc} \in \mathcal{P}(\mathbb{N}^d)$$

by a **finite computable superset** $R^\sharp_{pc}$:

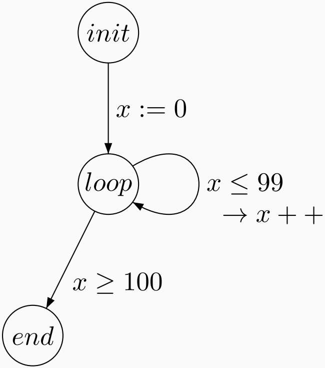


- And compute such **abstract values** for *each control point*.
- How ? mimic the program operations

$$\mathbb{N}^d \times pcs \to \mathbb{N}^d \times pcs$$

by their abstract versions.

Computing (inductive) invariants with intervals



- **ex :** Propagate range information

Example (Pagai, Verimag)

```
int main(int argc, char** argv){

  int x, y;
  x = 1;
  y = 2;
  /* reachable */
  /* invariant:
  3-2*y+x = 0
  5-y >= 0
  -2+y >= 0
  */
  while (x<8){
    x = x+2;
    y = y+1;
  }
  /* reachable */
  return 0;
}
```

see http://pagai.forge.imag.fr

Challenges in Abstract Interpretation

- More data structures : pointers, arrays, . . .
- Thousands, millions of lines of code to analyze.
- Static analyzers and compilers are complex programs (that also have bugs).

▶ Growing need for simple **specialized** analyses that **scale**

Memory Analyses
Focus on expressivity - scalability - compilers.

Plan

Abstract Interpretation in Compilers

Classical analyses (and optimisation) inside (production) compilers :

- Apart from classical dataflow algorithm, often **syntactic**.
- Usual abstract-interpretation based algorithms are too costly.
- Expressive algorithms : rely on “high level information”.

▶ Need for safe and precise quasi linear-time algorithms at **low-level**.

▶ Illustration with OOPLSA’14 paper

Collaborations with M. Maalej, F. Pereira and his team at UFMG, Brasil, slides inpired from theirs.

Designing a scalable static analyses sequence : an example

OOPSLA'14 :

- A technique to prove that (some) memory accesses are safe :
  - Less need for additional guards.
  - Based on abstract interpretation.
  - Precision and cost compromise.
- Implemented in LLVM-compiler infrastructure :
  - Eliminate 50% of the guards inserted by AddressSanitizer
  - SPEC CPU 2006 17% faster

A bit on sanitizing memory accesses

Different techniques : but all have an overhead.

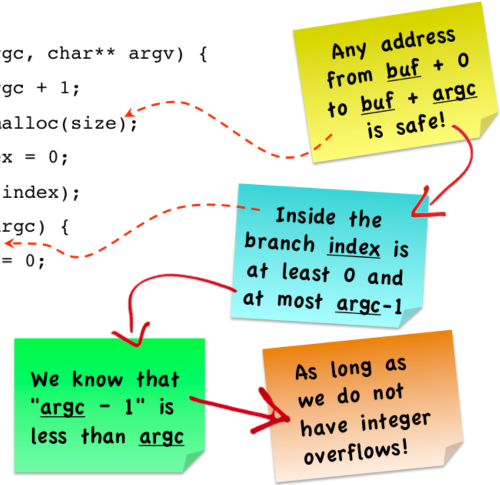
Ex : Address Sanitizer

- Shadow every memory allocated : 1 byte $\rightarrow$ 1 bit (allocated or not).
- Guard every array access : check if its shadow bit is valid. ▶ slows down SPEC CPU 2006 by 25%

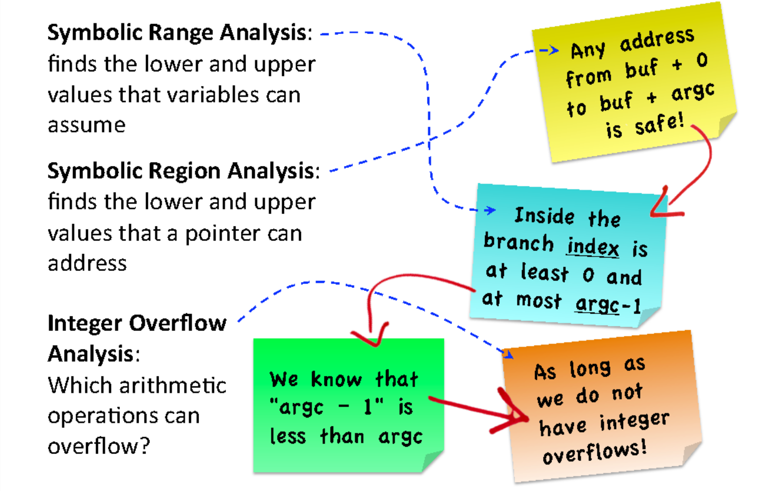
▶ We want to **remove these guards**.

Green Arrays : overview 1/2

```c
int main(int argc, char** argv) {
  int size = argc + 1;
  char* buf = malloc(size);
  unsigned index = 0;
  scanf("%u", &index);
  if (index < argc) {
    buf[index] = 0;
  }
  return index;
}
```



Green Arrays : overview 2/2



Symbolic ranges : How to ensure scalability ?

The idea is to work on the intermediate representation to ensure the following key property :

SSI Property
All abstract values are **stable** on their live ranges.

How ? Splitting variables **work on the Intermediate representation**.

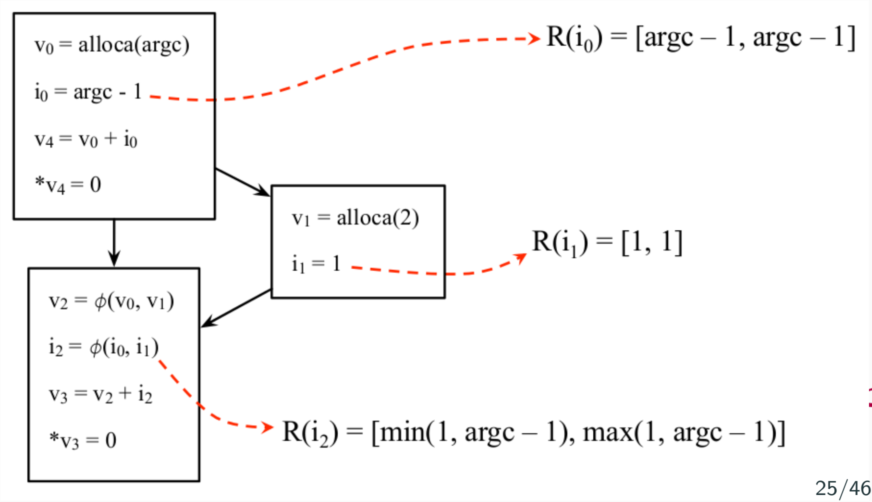
Symbolic Ranges (SRA) : Running example

```
int main(int argc){
  int* v  = malloc(sizeof(int)*argc);
  int  i  = argc - 1;
  v[i] = 0;
  if (?) {v = realloc(sizeof(int)*2); i=1 ;}
  v[i] = 0;
}
```

- Are all accesses to v **safe** ?

Symbolic Ranges (SRA) : ex within the SSA form

SSA = Static Single Assignment

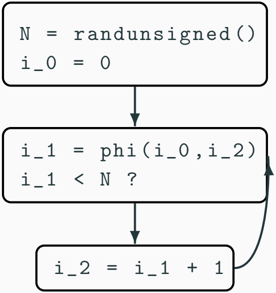


SRA on SSA form : a sparse analysis

- An abtract interpretation-based technique.
- Very similar to classic range analysis.
- One abstract value (R) **per variable** : sparsity.

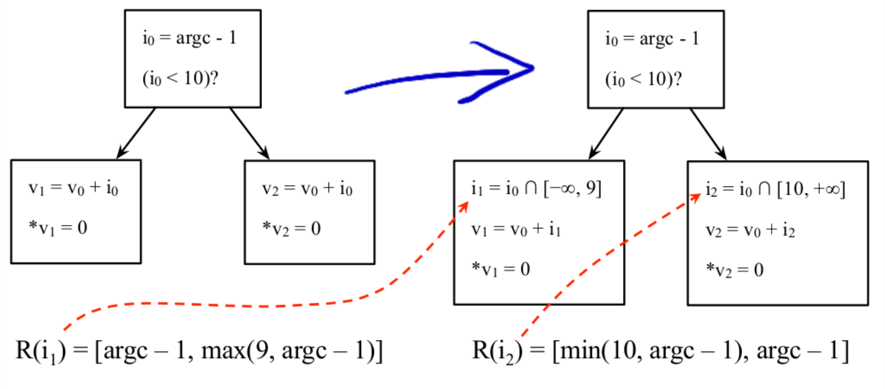
▶ Easy to implement (simple algorithm, simple data structure).

SRA on SSA form : an example



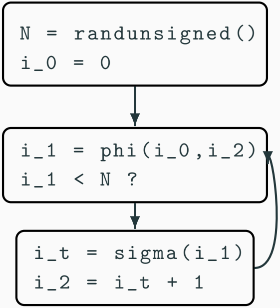
- $R(i_0) = [0, 0]$
- $R(i_1) = [0, +\infty]$
- $R(i_2) = [1, +\infty]$

Improving precision of SRA : live-range splitting 1/2



- e-SSA form.

SRA + live-range on an example



$$R(i_t) = [R(i_1)\downarrow, min(N-1, R(i_1)\uparrow)]$$

- $R(i_0) = [0, 0]$
- $R(i_1) = [0, N]$

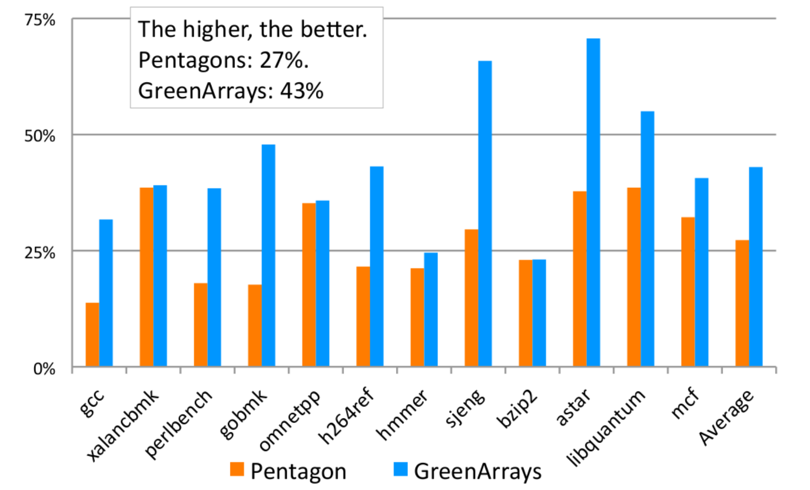
- **Implementation**: LLVM + AddressSanitizer
- **Benchmarks**: SPEC CPU 2006 + LLVM test suite
- **Machine**: Intel(R) Xeon(R) 2.00GHz,with 15,360KB of cache and 16GB or RAM
- **Baseline**: Pentagons
  - Abstract interpretation that combines "less-than" and "integer ranges".†

```
int i = 0;
unsigned j = read();
if (...)
    i = 9;
if (j < i)
  ...
```

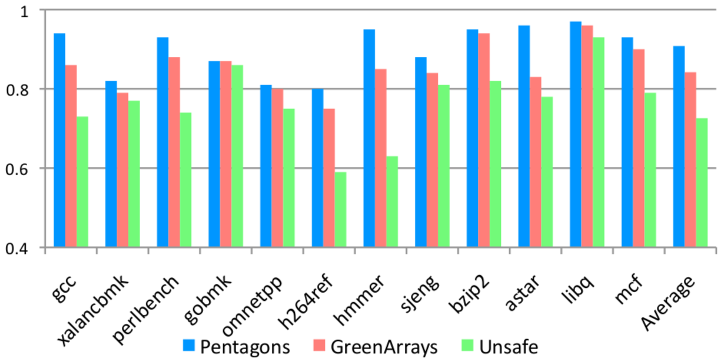
P(j) = (less than {i}, [0, 8])

†: Pentagons: A weakly relational abstract domain for the efficient validation of array accesses, 2010, Science of Computer Programming

Percentage of bound checks removed



Runtime improvement



The lower the bar, the faster. Time is normalized to AddressSanitizer without bound-check elimination. Average speedup: Pentagons = 9%. GreenArrays = 16%.

Conclusion and Research Questions

In the presented work :

- Work on an appropriate **intermediate representation**.
- Safety is proved.
- Interprocedural analysis.

On this part :

- More relational analyses ?
- Combination of analysis/optimisation ?
- Inside LLVM ecosystem ?

Plan

Goal : more functional properties

- Array bound check is "Index-based verification."
- What about relationship between indices and contents ?

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

Program Verification as solving Horn Clauses

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

- Abstract semantics as Horn Clauses :

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

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

▶ Toward a new IR for verification : at least expressive.

Contributions : SAS 2016 [Gonnord Monniaux] and NSAD 2020 [Braine Gonnord]

- (SAS16) A new abstraction for *programs* with arrays :
  - with **tunable** precision.
  - into Horn clauses (a special type of formula) without arrays.
  - extensible to other data structures (maps, . . .).
- (NSAD20) (with J. Braine) : a general technique to abstract data-structures in Horn problems.
- WIP (J. Braine) : other nice results such as **completeness**.

Some of the next slides are adapted from Julien Braine's talk at NSAD 2020.

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.**

▶ State-of-the-art solvers (Z3/PDR, Z3/Spacer, Eldarica) are really not performant.

Difficulties of datastructures

Example of data structures

1. **Arrays**
2. Sets
3. Maps
4. Trees
5. Graphs

Interesting Invariants

1. Involve a non bounded number of elements $\forall i, a[i] = 0$
2. Involve relations between elements $\forall i, j, i < j \Rightarrow a[i] < a[j]$
3. Involves the structure $\forall n1, n2, n2 \in child(n1) \Rightarrow n1 < n2$

We need quantified invariants !

Focus : arrays

Data abstraction

Idea : define relations between abstracted and concrete elements :

Data-abstraction σ

1. Definition : $\sigma : A \to \mathcal{P}(B)$
2. Encoded by an explicit formula $F_\sigma(a, a^\#) \equiv a^\# \in \sigma(a)$

- This is a Galois connection.

Examples

Simple Example : Sign abstraction

1. Sign abstraction : $\sigma(i \in \mathbb{Z}) = ite(i >= 0, \{Pos\}, \{Neg\})$
2. $F_\sigma(i, i^\#) \equiv ite(i >= 0, i^\# = Pos, i^\# = Neg)$

Some array abstractions

1. Array smashing : $F_\sigma(a, v) \equiv \exists i, a[i] = v$
2. Array slicing/partitioning on i : $F_\sigma((a, i), (sliceid, v, i')) \equiv \exists j, sliceid = ite(j < i, 0, ite(j = i, 1, 2)) \wedge v = a[j] \wedge i' = i$
3. **1-Cell Morphing** [Gonnord Monniaux SAS16] : $F_\sigma(a, (q, v)) \equiv v = a[q]$

Cell Morphing subsumes the others

Data abstraction technique

Algorithm
Replace $P(a, \overrightarrow{x})$ by $\forall a^{\#}, F_{\sigma}(a, a^{\#}) \Rightarrow P^{\#}(a^{\#}, \overrightarrow{x})$ everywhere

Result
Given a Horn problem H, $H^{\#}$ has a solution iff H has a solution S such that $\gamma \circ \alpha(S) = S$. This implies soundness.

Problem
We have added a quantifier alternation depth ! Solvers do not handle them !

Example : cell morphing

Initial clause : Array initialization loop

$\forall a, a', i, n, \ P(a, i, n) \wedge i < n \wedge a' = a[i \leftarrow 0] \Rightarrow P(a', i+1, n)$

Abstracted clause using cell morphing

$\forall a, a', i, n, \ (\forall q, v, v = a[q] \Rightarrow P^{\#}(q, v, i, n)) \wedge$

$i < n \wedge a' = a[i \leftarrow 0] \Rightarrow (\forall q', v', v' = a'[q'] \Rightarrow P^{\#}(q', v', i+1, n))$

Simplified

$\forall a, a', i, n, q', \ (\forall \boldsymbol{q}, P^{\#}(q, a[q], i, n))$

$\wedge\, i < n \wedge a' = a[i \leftarrow 0] \Rightarrow P^{\#}(q', a'[q'], i+1, n)$

How to remove the quantifier $\forall q$?

Eliminating the quantifiers

Technique
Replace an infinite conjunction ($\forall$) by a finite one. The finite set most be chosen wisely !

Chosen finite conjunction for Cell abstraction
Idea : focus on the cells that matter in the clause !

In practice : use the cell indices that are used in a read

Example, continued

- Clause : $\forall a, a', i, n, q', (\forall \boldsymbol{q}, P^{\#}(q, a[q], i, n)) \wedge i < n \wedge a' = a[i \leftarrow 0] \Rightarrow P^{\#}(q', a'[q'], i+1, n)$
- Indices used in a read operation : q'
- Clause after elimination of quantifier q
  $\forall a, a', i, n, q', P^{\#}(q', a[q'], i, n) \wedge i < n \wedge a' = a[i \leftarrow 0] \Rightarrow P^{\#}(q', a'[q'], i+1, n)$

A few experiments [I.Dillig T.Dillig Aiken]

Setting

1. Benchmarks written in toy java language
2. Solving with Z3, 30s timeout
3. Comparison : Z3 directly, Vaphor tool from [GM SAS16], $Cell_1$

	#exp	Noabs		Vaphor		$Cell_1$	
		👍	👎	👍	👎	👍	👎
NotHinted	12	0	0	1	0	0	0
Hinted	12	0	0	5	0	12	0
Buggy	4	4	0	4	0	4	0

Analysis

1. No unsound results but Requires hints
2. Hints allow to solve the problems ⇒ Abstraction is good
   ⇒**Z3 has trouble on our non quantified integer problems**
3. Great improvement compared to Vaphor for hinted problems

Conclusion & Research Questions

On the current work :

- WIP : completeness results, and experimental deeper evaluations.
- Other data structures : trees.

On this part :

- Horn Clauses are a good intermediate representation but perhaps not mature enough : embed more structural properties ?
- What about scalability ?