Abstract Interpretation 101

I3S Seminar

Laure Gonnord

University of Lyon / LIP

May 13th, 2017

Plan

Motivation

Basic ingredients of abstract interpretation

An example of a tailored abstract domain design
- Overview
- Experimental results
- Scalable symbolic abstract domain

Conclusion

Goal: Safety

Prove that (some) memory accesses are safe:

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

- Fight against bugs and overflow attacks.

Goal: Correctness

Automatically generate loop invariants:

```
void fill_mini (int a[N],int l, int u) {
    unsigned int i=l;
    int b=a[l]
    while (i<=u){
        if(a[i]<b) b=a[i] ;
        i++ ;
    }
        //here b contains min(a[l..m])
}
```

@loop_i ,
b=min(a[l..i-1])

- Functional assurance.

Goal: Performance 1/2

Enable code motion:

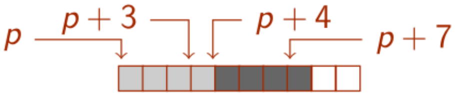
```
int main () {
    unsigned int i,j ;
    i=42 ; j=1515 ;
    while(i>0) i-- ; ✔
    foo(j);
}
```

- Safe iff the loop terminates.

Goal: Performance 2/2

Enable loop parallelism:

```
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 ;
}
```



- The two regions do not overlap.

Proving non trivial properties of software 1/2

- Basic idea: software has **mathematically defined behaviour**,
- We want to prove: behaviours $\subseteq$ acceptable behaviours
- In an **automatic way**!

The Halting Problem, proof - Program Version.

Suppose we have a "magical analyzer" A: answer $A(P, X) = 1$ if "program P terminates eventually on input X" $A(P, X) = 0$ otherwise.

```
int B(Program x) {
  if (A(x,x)==0) {
    return 1;
  } else {
    while(true) {}
  }
}
```

What is $B(B)$? (B applied to its own source code)?

- If $B(B) = 1$ then $A(B, B) = 0$ "program B does not terminate on input B". Absurd!
- If $B(B)$ loops then $A(B, B) = 1$ "program B terminates on input B". Absurd!

- **There is no magical static analyzer.**

Workarounds

What is impossible is to check reachability:

1. automatically
2. without false positives
3. without false negatives
4. on systems of unbounded state
5. with unbounded execution time

Lifting restrictions opens possibilities!

- **Abstract Interpretation** = enabling false positives.

Static Analysis with Abstract Interpretation

Warning, here IA is not artificial intelligence :-) But it is still magic!

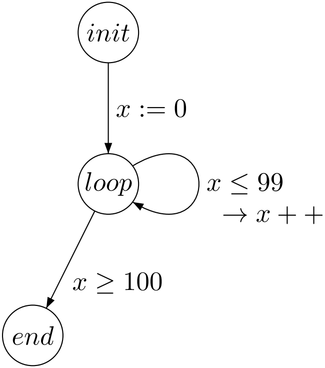
In the rest of the talk

Plan:

- Abstract Interpretation basics (in the case of numerical programs).
- Toward more efficient abstract domains for compilation.

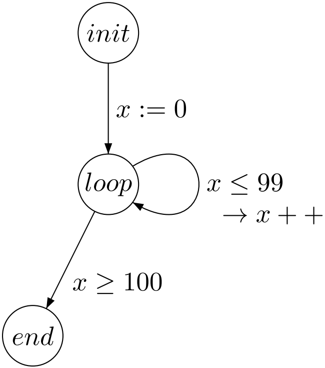
Plan

What's an invariant?



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

How to compute an invariant?



- $\{0, \ldots 100\}$: set of 101 elements, thus 101 steps. This can be **infinite!**

Solving the impossible

Magic?

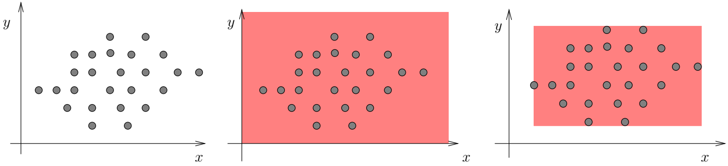
- compute until it stops, and cross fingers?
- compute but be imprecise!

Main ingredient: abstract values - intuition

Idea: represent values of variables:

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

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



- And compute such **abstract values** for *each control point*.

Second ingredient: abstract operations

Idea: mimic the program operations

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

by their abstract versions.

- And **execute/interpret** the abstract program!

Example: Interval abstract domain

Try to compute an **interval** for each variable at each program point using:

```
assume(x >= 0 && x<= 1);
assume(y >= 2 && y <= 3);
assume(z >= 3 && z <= 4);
t = (x+y) * z;
```

Interval for *z*? $[6, 16]$

- Interval **abstract** operations : $+, -, \times$ on intervals : interval arithmetic

Abstract tests

```
assume(x >= 0 && x<= 10);
if (x <= 5)
   y = x-3
else
   y = x+3;
z = y+1
```

We have to speculate, and invent an abstract union at the end of the test: $[a, b] \sqcup [c, d] = [min(a, b), max(b, d)]$.

Loops?

Loops are special cases of tests

```
int x=0;
while (x<1000) {
  x=x+1;
}
```

Loop iterations (with union) $[0,0]$, $[0,1]$, $[0,2]$, $[0,3]$,...

- Stricly growing interval during 1000 iterations, then stabilizes : $[0,1000]$ is an **invariant**.

Some nice properties

  - If the computation stabilizes, all sets are super-sets of the actual values of the program variables.
  - If the abstract domain is simple enough (signs) this termination is guaranteed.
- :-(It is not the case for intervals.

Termination Problem

Third problem to cope with : **stopping the computation** :

- Too many computations.
- Unbounded loops.

One solution. . .

Extrapolation!

$[0,0]$, $[0,1]$, $[0,2]$, $[0,3] \to [0,+\infty)$

Push interval:

```
int x=0; /* [0, 0] */
while /* [0, +infty)*/ (x<1000) {
  /* [0, 999] */
  x=x+1;
  /* [1, 1000] */
}
```

Yes! $[0, \infty[$ is stable!

Extrapolation with widening

Widening operator for intervals : ($I_1 \nabla I_2$ with $I_1 \subseteq I_2$)

$$[a, b] \nabla [c, d] = [\text{if } c < a \text{ then } -\infty \text{ else } a,$$

$$\text{if } d > b \text{ then } +\infty \text{ else } b]$$

On the example (on board):

```
int x=0;
while (x<1000)    x=x+1;
```

- At the loop control point, the computation of $next = f^\sharp(previous)$ is replaced by $previous \nabla next$.

Computing inductive invariants as intervals - Summary

- Representation : intervals. The union leads to an overapproximation.
- We don't know how to compute $R(P)$ with P interval (The statements may be too complex, ...)
  ▶ Replace computation by simpler over-approximation $R(X) \subseteq R^\sharp(X)$.
- The convergence is ensured by **extrapolation/widening**.

▶ We always compute $\phi^\sharp(X)$ with : $\phi(X) \subseteq \phi^\sharp(X)$
In the end, **over-approximation** of the least fixed point of ϕ.

Computing inductive invariants as intervals - Property

Theorem

(Cousot/Cousot 77) Iteratively computing the reachable states from the entry point with the interval operators and applying widening at entry nodes of loops converges in **a finite** *number of steps to a overapproximation of the least invariant (aka* **postfixpoint***).*

Demo!

Pagai tool (Verimag)

Design your own abstract domain!

- Abstract values must have a **lattice structure**.
- *concretization*(*abstraction*(*val*)) bigger than *val*.
- The abstract operations must be correctly designed.
- If the lattice is infinite height, then the widening operator must satisfy the non ascending chain condition (see Cousot/Cousot 1977).

▶ There are generic analyzers where you only have to provide your domain operations.

Plan

Challenges in Abstract Interpretation

  - Precision of the abstract domain.
  - 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**

Designing a scalable static analysis: 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

Outline

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;
}
```

Any address from buf + 0 to buf + argc is safe!

Inside the branch index is at least 0 and at most argc-1

We know that "argc - 1" is less than argc

As long as we do not have integer overflows!

Green Arrays : overview 2/2

Symbolic Range Analysis: finds the lower and upper values that variables can assume

Symbolic Region Analysis: finds the lower and upper values that a pointer can address

Integer Overflow Analysis: Which arithmetic operations can overflow?

Any address from buf + 0 to buf + argc is safe!

Inside the branch index is at least 0 and at most argc-1

We know that "argc - 1" is less than argc

As long as we do not have integer overflows!

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 (v, i in the last example).

(technical stuff later if there remains time)

Outline

Experimental setup

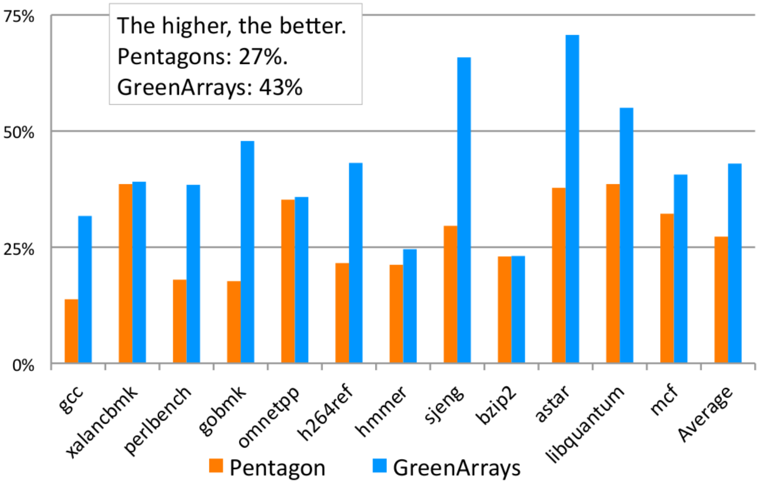
- **Implementation**: LLVM + AddressSanitizer
- **Benchmarks**: SPEC CPU 2006 + LLVM test suite
- **Machine**: Intel(R) Xeon(R) 2.00GHz,with 15,360KB of cache and 16GB or RAM
- **Baseline**: Pentagons
  - Abstract interpretation that combines "less-than" and "integer ranges".†

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

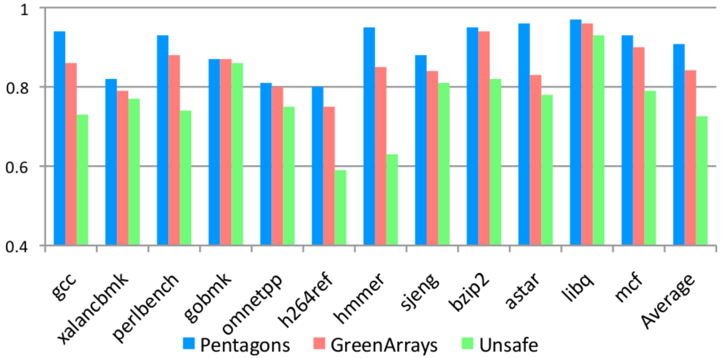
$P(j) = (\text{less than } \{i\}, [0, 8])$

†: 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%.

Outline

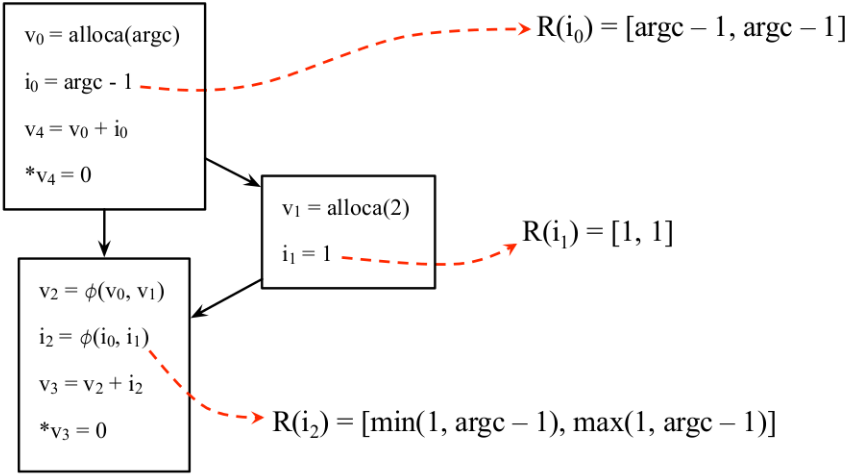
Symbolic Ranges (SRA): Running example

```
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**?

Skip technical stuff

Symbolic Ranges (SRA): On the SSA form



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.

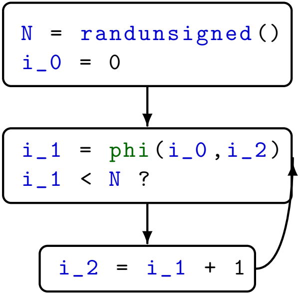
► Easy to implement (simple algorithm, simple data structure).

SRA on SSA form: constraint system

$$
\begin{array}{rcl}
v = \bullet & \Rightarrow & R(v) = [v, v] \\
v = o & \Rightarrow & R(v) = R(o) \\
v = v_1 \oplus v_2 & \Rightarrow & R(v) = R(v_1) \oplus^I R(v_2) \\
v = \phi(v_1, v_2) & \Rightarrow & R(v) = R(v_1) \sqcup R(v_2) \\
\text{other instructions} & \Rightarrow & \emptyset
\end{array}
$$

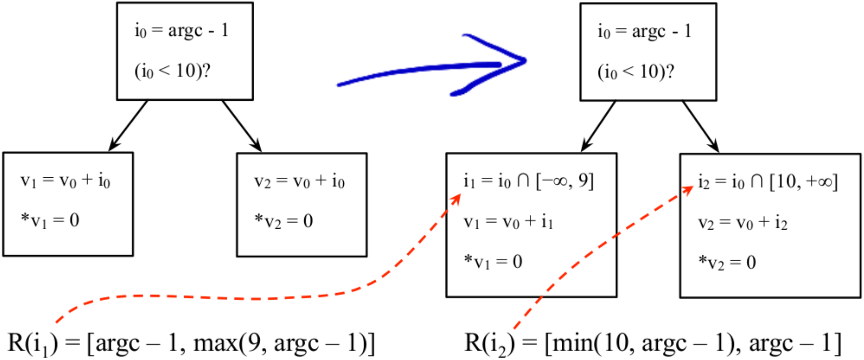
$\oplus^I$: abstract effect of the operation $\oplus$ on two intervals.
$\sqcup$: convex hull of two intervals. ▶ All these operation are performed symbolically thanks to **GiNaC**

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.

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

Rule for live-range splitting :

$t = a < b$
`br` (t, ℓ)

$a_f = \sigma(a)$
$b_f = \sigma(b)$

ℓ
$a_t = \sigma(a)$
$b_t = \sigma(b)$

$$\Rightarrow \quad \begin{aligned} R(a_t) &= [R(a)_\downarrow, \min(R(b)_\uparrow - 1, R(a)_\uparrow)] \\ R(b_t) &= [\max(R(a)_\downarrow + 1, R(a)_\downarrow), R(b)_\uparrow] \\ R(a_f) &= [\max(R(a)_\downarrow, R(a)_\uparrow), R(a)_\uparrow] \\ R(b_t) &= [R(b)_\downarrow, \min(R(a)_\uparrow, R(b)_\uparrow)] \end{aligned}$$

- All simplications are done by GiNaC.

SRA + live-range on an example

```
N = randunsigned()
i_0 = 0
```

```
i_1 = phi(i_0,i_2)
i_1 < N ?
```

```
i_t = sigma(i_1)
i_2 = i_t + 1
```

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

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

Plan

In the paper (OOPSLA'14)

A complete formalisation of all the analyses :

- Concrete and abstract semantics.
- Safety is proved.
- Interprocedural analysis.

- `https://code.google.com/p/ecosoc/`

Remaining question : improving precision of the symbolic range analysis ?

Take away message

Abstract Interpretation is a powerful tool!