A journey on static analyses of programs

From code verification to code optimisation

Laure Gonnord

6 mai 2021

University Claude Bernard Lyon 1 / LIP, Lyon, France

Plan

Motivations

Abstract Interpretation for compilers

Scalable analyses for pointers

Code analysis for binaries

Impact on compiler optimisation passes

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]; ✗
}
```

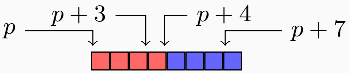
- This program has an illegal array access.

Goal : performance -ex

Enable loop parallelism :

```
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 ;
}
```

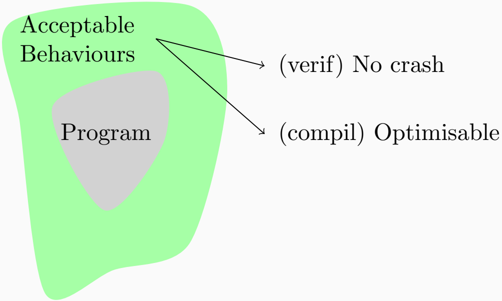
Parallel loops



- 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

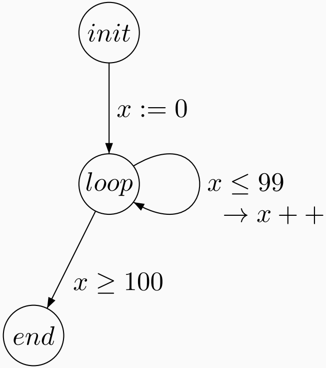
i.e. no magical static analyser. It is ~~im~~possible to prove interesting properties :

- 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 ?

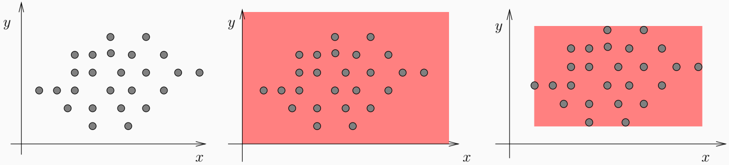
- Approximate sets (abstract domains), compute in this abstract world.
- Extrapolate (widening).

Main ingredient : abstract values

Idea : represent values of variables :

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

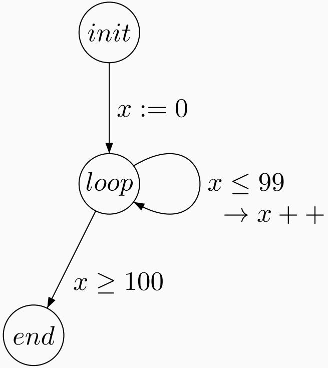
by a **finite computable superset** $R^\sharp_{pc}$:



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

There is also this magical widening stuff, let's forget it in this talk

Computing (inductive) invariants with intervals



- **ex :** Propagate range information

Example (Pagai, Verimag)

```c
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;
}
```

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.

Safe compilation ?

- **Correct-by-construction** non-optimising compilers : Lustre, Scade.
- Translation validation : specialized proof of the generated code.
- Compcert.

▶ An evolution toward more trustable compilers. But what about code optimisation ?

Motivation

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" information that is usually absent in the low level program representations adopted by compilers.

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

▶ Illustrations in the rest of the talk.

Some contributions

- Abstract domains/iteration strategies for numerical invariants [SAS11], [OOPSLA14].
- Applications to memory analysis [OOPSLA14], just in time compilers [WST14].
- **Pointer analysis** with "sparse" abstract interpretation [CGO16] [CGO17] [SCP17].
- Polyhedral analysis on **binary code** [VMCAI19]

Collaborations with M. Maalej, F. Pereira and his team at UFMG, Brasil

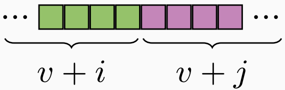
+ with C. Ballagria, J. Forget, Lille

Plan

Pointer analysis : motivating example

```
void partition(int *v, int N) {
  int i, j, p, tmp;
  p = v[N/2];
  for (i = 0, j = N - 1;; i++, j--) {
    while (v[i] < p) i++;
    while (p < v[j]) j--;
    if (i >= j)
      break;
    tmp = v[i];
    v[i] = v[j];
    v[j] = tmp;
  }
}
```

v[i] = *(v+i)



Motivating example - LLVM version

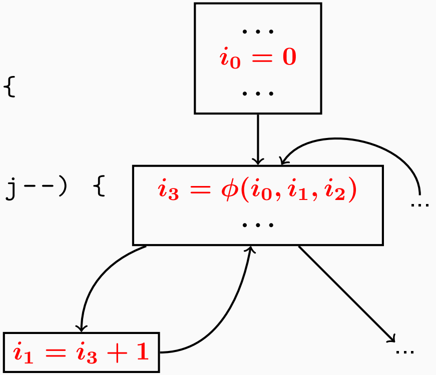
```
for.cond:                          ; preds = %for.inc, %entry
  %i.0 = phi i32 [ 0, %entry ], [ %inc18, %for.inc ]
  %j.0 = phi i32 [ %sub, %entry ], [ %dec19, %for.inc ]
  br label %while.cond

while.cond:                        ; preds = %while.body, %for.cond
  %i.1 = phi i32 [ %i.0, %for.cond ], [ %inc, %while.body ]
  %idxprom1 = sext i32 %i.1 to i64
  %arrayidx2 = getelementptr inbounds i32* %v, i64 %idxprom1
  %1 = load i32* %arrayidx2, align 4
  %cmp = icmp slt i32 %1, %0
  br i1 %cmp, label %while.body, label %while.end
```

- On a perdu une partie du contrôle (et les tableaux)

Scaling analyses : program representation (simpl.) 1/2

Static Single Assignment (**SSA**) form : each variable is defined/assigned once.

```
void partition(int *v, int N) {
   int i, j, p, tmp;
   p = v[N/2];
   for (i = 0, j = N - 1;; i++, j--) {
      while (v[i] < p) i++;
      ...
}
```



- Sparse storage of **value** information (one value range per variable name).

Our setting for scaling analyses

Classical abstract interpretation analyses :

- Information attached to $(\textit{block}, \textit{variable})$.
- A new information is computed after each statement.

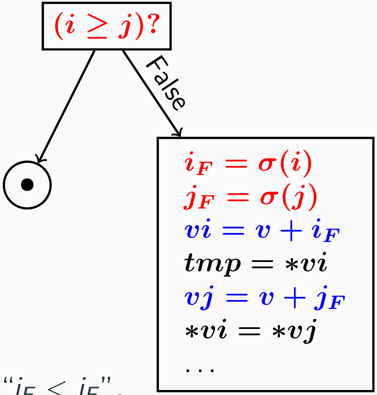
Sparse analyses $\Rightarrow$ **Static Single Information (SSI) Property :**

- Attach information to variables.
- The information must be invariant throughout the live range of the variable.

▶ Work on suitable intermediate representations.

But, in our analysis, range information is not sufficient to disambiguate `v[i]` and `v[j]`
Within **SSA** form, tests/relational information cannot be propagated !

```
void partition(int *v, int N) {
  ...
    if (i >= j)
        break;
    tmp = v[i];
    v[i] = v[j];
}
```



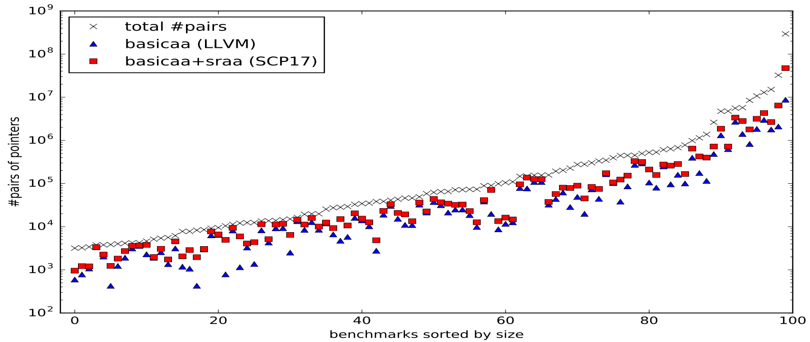
- $i \geq j$ is invariant nowhere in classical SSA.
- The σ renaming (**e-SSA**) enables to propagate "$i_F < j_F$".

Contributions on static analyses for pointers

(with Maroua Maalej) [CGO16, CGO17, SCP17]

- A new sequence of static analyses for pointers.
- Based on semi-relational sparse abstract domains :
  - In CGO'16 : $p \mapsto loc + [a, b]$.
  - In CGO'17 : adaptation of Pentagons.
- Implemented in LLVM.
- Used as oracles for a common pass called `AliasAnalysis`.
- Experimental evaluation on classical benchmarks.

Experimental results [SCP17]



- Comparison with LLVM basic alias analysis.
- Our sraa outperforms basicaa in the majority of the tests.
- The combination outperforms each of these analyses separately in every one of the 100 programs.

Plan

Motivations

Abstract Interpretation for compilers

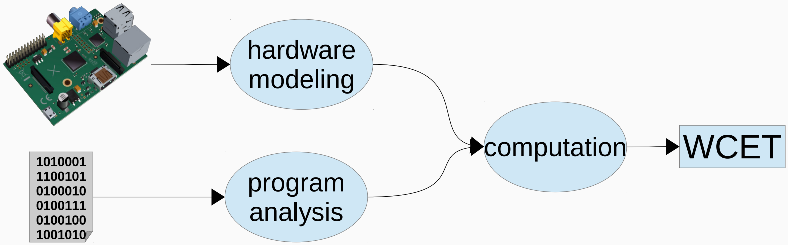
Scalable analyses for pointers

Code analysis for binaries

Impact on compiler optimisation passes

Context, Contribution

Real-time systems, scheduling needs precise worst-case execution time evaluation : **on the binary**.



- A new abstract interpretation on binary with polyhedra.

Slides from C. Ballabriga.

Analysing binary is difficult

No control, no variable, no type, only data locations.

- Look for memory accesses in the binary code
- Not always obvious that 2 accesses refer to the same data location

Aliasing example (Pseudo-Assembly)

```
SET     r1, #42
ADD     r4, sp, #8
STORE   r1, [r4 - #4]
...
LOAD    r3, [sp + #4]
ADD     r3, r3, #1
```

- LOAD and STORE access the same statically-unknown address

A taste of the abstract analysis

Abstract state shape : $(P(values), \text{register mapping}, \text{mem mapping})$.

- A memory value is represented by the dereferencing of a polyhedra variable
- The "memory mapping" encode (de)references

```
SET     r3 , #42
STORE   r3 , [sp + #4]
```

▶ Here : $(\langle x_3 = 42, x_4 = x_5 + 4 \rangle, \{r_3 : x_3, sp : x_5\}, \{x_4 : x_3\})$

Experimental evaluation

- We developed a prototype called *Polymalys*
  - Implements the approach in C++
  - Is a plugin for OTAWA
  - OTAWA is an open-source modular tool for WCET static analysis
- OTAWA handles :
  - ELF binary loading
  - CFG reconstruction
  - Architecture-independent instruction abstraction

Evaluation : benchmarks

			Bounded loops				Time (ms)	
Bench	*LoC*	*Loops*	Polymalys	*SWEET*	*Pagai*	*oRange*	Polymalys	*Pagai*
crc	16	1	1	1	1	1	150	40
fibcall	22	1	1	1	1	1	230	50
janne_complex	26	2	1	2	1	1	870	140
expint	56	3	3	2	3	3	850	9140
matmult	84	5	5	5	5	5	3640	1380
fdct	149	2	2	2	2	2	12450	2150
jfdctint	165	3	3	3	3	3	10920	1960
fir	189	2	2	2	2	1	11630	390
edn	198	12	12	12	9	12	25190	15660
ns	414	4	4	4	4	4	1740	380
gemver	186	10	10	N/A	10	10	12136	6029
covariance	138	11	11	N/A	11	11	7248	836
correlation	168	13	13	N/A	N/A	13	9129	25062
nussinov	143	8	8	N/A	8	8	7272	2811
floyd-warshall	112	7	7	N/A	2	7	2904	468

- We ran the tools on Mäalardalen and Polybench
- Strength of our tool in this comparison :
  - Works on binary
  - Tends to better estimate loop bounds
  - Reasonable analysis time, guaranteed to terminate

Plan

Evaluating analyses in LLVM

LLVM compiler :

- comes with a test infrastructure and benchmarks.
- analysis and optimisation passes log information.
- you can add your own pass, but **where ?**

```
clang -c -emit-llvm $1 -o $name.bc
opt -mem2reg -instnamer $name.bc -o $name.rbc
sage-opt -load $lib_path/$ssify_so -break-crit-edges -ssify -set 1000  $name.rbc -o $name.rbc
sage-opt -stats -load $lib_path/$python_so -load $lib_path/$sage_so -load $lib_path/$sra_so -load $lib_path/$rbaa_so -lo
ad $lib_path/$licm2_so -range-based-aa -aa-eval -no-aa -tbaa -targetlibinfo -basicaa -notti -verify -simplifycfg -domtre
e -sroa -lower-expect -targetlibinfo -no-aa -tbaa -basicaa -notti -ipsccp -globalopt -deadargelim -domtree -instcombine
-simplifycfg -basiccg -prune-eh -inline-cost -inline -functionattrs -argpromotion -sroa -domtree -lazy-value-info -jump-
threading -correlated-propagation -simplifycfg -domtree -instcombine -tailcallelim -simplifycfg -reassociate -domtree -l
oops -loop-simplify -lcssa -loop-rotate -licm2 -loop-unswitch -instcombine -scalar-evolution -loop-simplify -lcssa -indv
ars -loop-idiom -loop-deletion -loop-unroll -memdep -mldst-motion -domtree -memdep -range-based-aa -gvn -view-cfg -memde
p -memcpyopt -sccp -domtree -instcombine -lazy-value-info -jump-threading -correlated-propagation -domtree -memdep -dse
-range-based-aa -adce -simplifycfg -domtree -instcombine -barrier -domtree -loops -loop-simplify -lcssa -branch-prob -bl
ock-freq -scalar-evolution -loop-vectorize -instcombine -scalar-evolution -slp-vectorizer -simplifycfg -domtree -instcom
bine -loops -loop-simplify -lcssa -scalar-evolution -loop-unroll -strip-dead-prototypes -globaldce -constmerge -verify $
name.rbc -o $name.opt.rbc
```

▶ Evaluating the impact of a given analysis is a **nightmare** !

Impact on our alias analysis on LLVM code motion 1/2

Loop invariant code motion (LICM) :

hoist!

```
void code_motion(int* p1, int *p2, int *p){
  // ...
  while(p2>p1){
    a = *p;
    *p2 = 4;
    p2 --;
  }
}
```

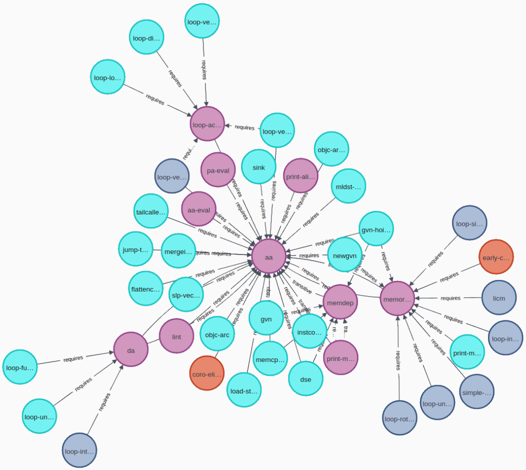
- If p and p_2 do not alias, then a=*p is invariant.

Impact of our analyses (excerpt) 2/2

Program	#Inst	#moved	
		O3	O3+our analysis (CGO16)
fixoutput	369	1	**5**
compiler	3515	0	0
bison	15645	165	**179**
archie-client	5939	0	0
TimberWolfMC	98792	1287	**1447**
allroots	574	0	0
unix-smail	5435	3	3
plot2fig	3217	3	3
bc	10632	18	**19**
yacr2	6583	144	**190**
ks	1368	8	**11**
cfrac	7353	5	**6**
espresso	50751	301	**398**
gs	55281	20	X

More in Maroua Maalej's thesis.

Understanding LLVM internals



⇝ charting the compiler. Figure from S. Michelland. More in the CAPESA project

Summary

Static analyses for compilers :

- Adaptation of abstract interpretation algorithms inside this particular context (**internal representations**).
- Algorithmic and compilation techniques to scale.
- Future work : more relational domains (and data structures), more applications, continue work on binary. . .