Contribution to static analyses: precision and scale

Habilitation thesis, November 9th, 2017

Laure Gonnord, University Lyon 1/LIP

Jury

Albert Cohen, Research Director, Inria Paris

Isabelle Guérin-Lassous, Professor, University of Lyon

Sebastian Hack, Professor, University of Saarland, Germany

Paul H J Kelly, Professor, Imperial College London, UK

Antoine Miné, Professor, Pierre & Marie Curie University

Andreas Podelski, Professor, University of Freiburg, Germany

Plan

Motivations

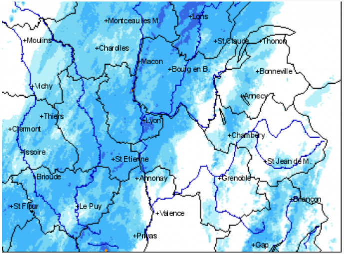
- Static analyses, examples
- Static analysis of software, how?

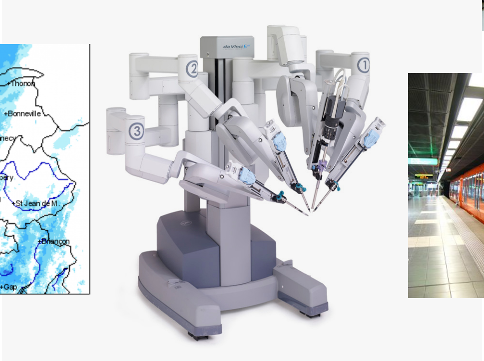
A new approach to software termination with compiler techniques

- Genesis of the first algorithm [SAS10]
- Toward scale and applicability [PLDI15]

Scaling abstract interpretation for more efficient compilers

Software is everywhere!





Software needs safety and performance

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

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, . . .

Software guarantees, how?

- Development processes: coding rules, ...
- Testing: do not cover all cases.
- Proof assistants: expensive.

► **Static analysis of programs**.

Goal: safety 1/2

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: safety 2/2

Prove program correctness/absence of functional bug:

```
void find_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 = min(a[l..u])
}
```

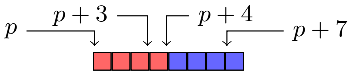
- This program finds the minimum of the sub-array.

Goal: performance 1/2

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 ;
}
```

Parallel loops



- The two regions do not overlap.

Goal: performance 2/2

Enable code motion:

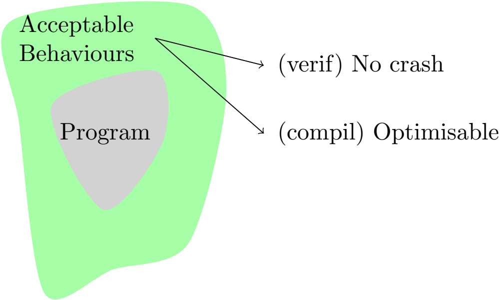
```
void code_motion(int* p1, int *p2, int *p){
  // ...
  while(p2>p1){
    a = *p;
    *p2 = 4;
    p2 --;
  }
}
```

hoist!

- If p and p_2 do not alias, then a=*p is invariant.
- Hoisting this instruction saves one load per loop.

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

▶ **Abstract Interpretation** = conservative approximations.

Contributions to static analysis 1/2

Guiding principle:

Cross fertilisation from/to different communities

- **Combination** of abstract interpretation with: logic, scheduling, compilation, optimisation. . .
- **Applications** in various domains: compilation, software verification, termination.

Contributions to static analysis 2/2

- Abstract domains/iteration strategies for numerical invariants [SAS11], [OOPSLA14].
- New numerical abstraction for the compilation of dataflow synchronous languages [LCTES11] [JoC12].
- A new approach to software termination with compiler techniques [SAS10] [PLDI15].
- Proving properties about arrays [OOPSLA14] [SAS16].
- Scaling abstract interpretation for more efficient compilers [CGO16] [CGO17] [SCP17]

Contributions to static analysis 2/2

- Abstract domains/iteration strategies for numerical invariants [SAS11], [OOPSLA14].
- New numerical abstraction for the compilation of dataflow synchronous languages [LCTES11] [JoC12].
- **A new approach to software termination with compiler techniques [SAS10] [PLDI15].**
- Proving properties about arrays [OOPSLA14] [SAS16].
- **Scaling abstract interpretation for more efficient compilers [CGO16] [CGO17] [SCP17]**

Plan

Motivations

Static analyses, examples

Static analysis of software, how?

A new approach to software termination with compiler techniques

Genesis of the first algorithm [SAS10]

Toward scale and applicability [PLDI15]

Scaling abstract interpretation for more efficient compilers

Context: transforming WHILE into FOR

Example: GCD of 2 polynomials

```
int gcd_aux(){
  // r ≤ da, db ≤ 2r
  while (da >= r) {
    if ( da <= db && undet() ){
      tmp = db;
      db = da;
      da = tmp - 1;
    }
    else
      da = da - 1;
  }
}
```

Hard to optimise for a hardware synthesis tool:

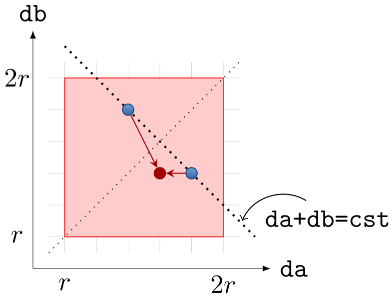
- Loop unrolling is impossible.
- Non-determinism, while loops.

▶ Need to bound the number of iterations.

Ranking function and dependencies

Proving termination: find a decreasing measure (**ranking function**).

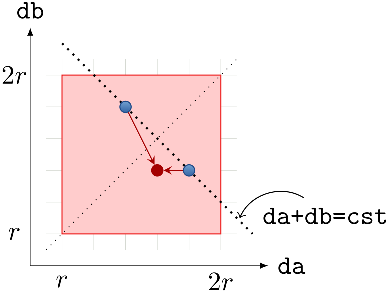
```
int gcd_aux(){
  // r ≤ da, db ≤ 2r
  while (da >= r) {
    if ( da <= db && undet() ){
      tmp = db;
      db = da;
      da = tmp - 1;
    }
    else
      da = da - 1;
  }
}
```



- Red dot values **depends on** blue ones (are computed after!)

Inspiration: termination is scheduling 1/2

	Scheduling	Termination
function (ρ)	≥ 0 $\nearrow$	≥ 0 $\searrow$
respects	dependencies	flow
$(W, da, db) \mapsto$	$4r - (da + db)$	$da + db$



- Adapt scheduling algorithms to **termination**.

Inspiration: termination is scheduling 2/2

Instruction scheduling algorithm [Fea92]:

- Compute (exactly) all the dependencies of a polyhedral kernel (syntactic restrictions) $\rightarrow$ system of constraints.
- Scheduling problem $\rightarrow$ system of constraints + objective function.

▶ Solving (Linear Programming) gives a **multidimensional schedule** of the form ($\vec{x}$ variables, k control point):

$$\rho(k, \vec{x}) = A_k.\vec{x} + \vec{b_k} \in \mathbb{N}^d$$

Inspiration: termination is scheduling 2/2

Instruction scheduling algorithm [Fea92]:

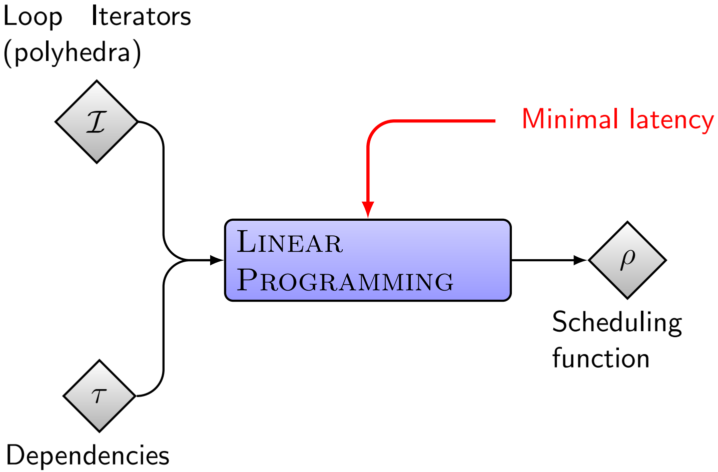
- Compute (exactly) all the dependencies of a polyhedral kernel (syntactic restrictions) $\rightarrow$ system of constraints.
- Scheduling problem $\rightarrow$ system of constraints + objective function.

▶ Solving (Linear Programming) gives a **multidimensional schedule** of the form ($\vec{x}$ variables, k control point):

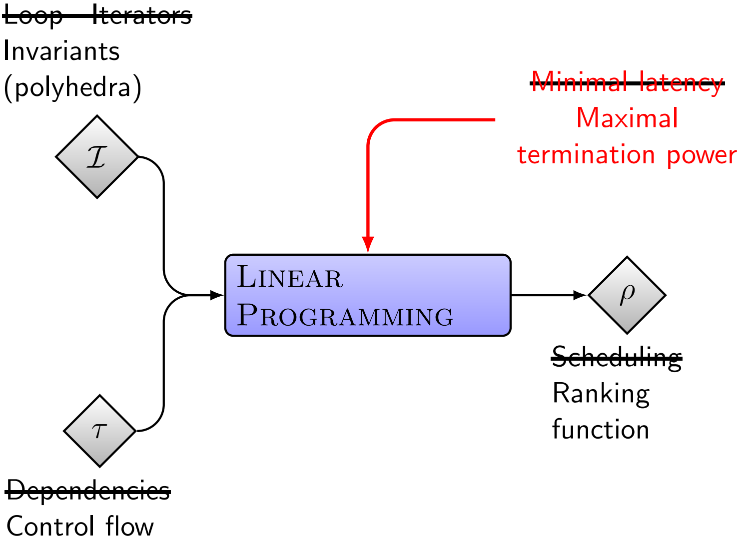
$$\rho(k, \vec{x}) = A_k.\vec{x} + \vec{b_k} \in \mathbb{N}^d$$

▶ **Adapt** to more general programs/termination.

From scheduling to termination



From ~~scheduling~~ to termination



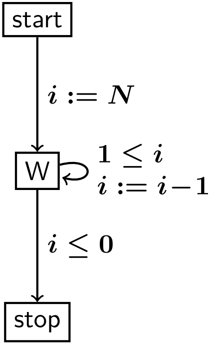
Contribution [SAS10]

Program termination with global multi-dimensional affine rankings:

- Incremental (one dimension per step).
- For a (large subset) of C programs, fully implemented
- **Worst-case computational complexity**, in case of success.

Algorithm to find 1D ranking functions

```
assume(N>0);
i := N;
while(i>0)
  i := i-1;
```



Searching for ranking function ρ:

$$
\begin{aligned}
\rho(start, \vec{x}) &= \alpha^1_{start}.\mathbf{i} + \alpha^2_{start}.\mathbf{N} \\
&+ \alpha^3_{start}.\mathbf{i_0} + \alpha^4_{start}.\mathbf{N_0} + \alpha^5_{start} \\
\rho(w, \vec{x}) &= \alpha^1_{w}.\mathbf{i} + \dots \\
\rho(stop, \vec{x}) &= \alpha^1_{stop}.\mathbf{i} + \dots
\end{aligned}
$$

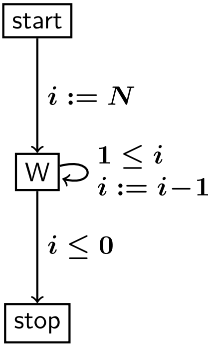
α^i_k are unknowns

The constraints are:

- For each control point k: $\rho(k, \vec{x}) \geq 0$ for $\vec{x} \in P_k$
- For each transition: $\rho(dest, \vec{x'}) < \rho(src, \vec{x})$

Algorithm to find 1D ranking functions

```
assume(N>0);
i := N;
while(i>0)
  i := i-1;
```



The previous constraints are not linear:

- Using the Farkas' lemma, linearize.
- Solve the LP Instance.
- We find $\rho = \begin{cases} \mathit{start} \rightarrow & 2 + N_0 \\ W \rightarrow & 1 + i \\ \mathit{stop} \rightarrow & 0 \end{cases}$

▶ Problem solved.

Experimental results: RANK

Sorting arrays of size n:

Name	LOCs	Time(analysis)[1]	dim	Worst Case Complexity Bound
insertion	12	0.2	3	$O(n^2)$
selection	20	0.4	3	$O(n^2)$
bubble	22	0.4	3	$O(n^2)$
shell	23	1.1	4	$O(n^3)$
heap	45	2.8	3	$O(n^2)$

[1] User time in seconds on a Pentium 2GHz with 1Gbyte RAM

Scaling the algorithm [PLDI15]

The former technique:

$$LP\ Size = O(\#vars \times \#Bblocks \times \#transitions)$$

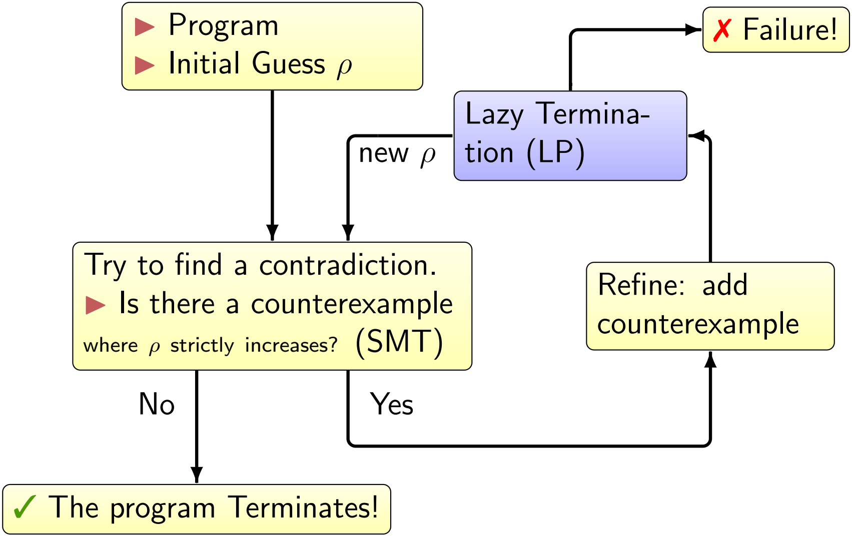
- scalability: all basic blocks $\rightsquigarrow$ big constraint systems
- precision: ρ must decrease at **each** transition.

New technique:

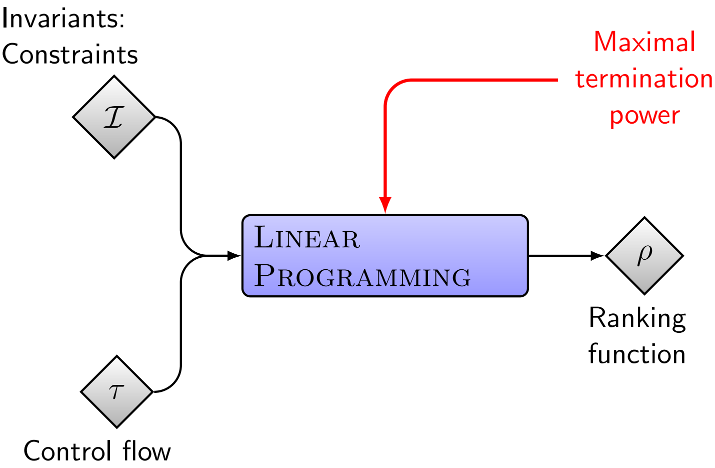
- only considers **a cut-set** of basic blocks.
- considers loops as single transitions.

▶ **We do not compute all paths** explicitly (Counter example-based algorithm).

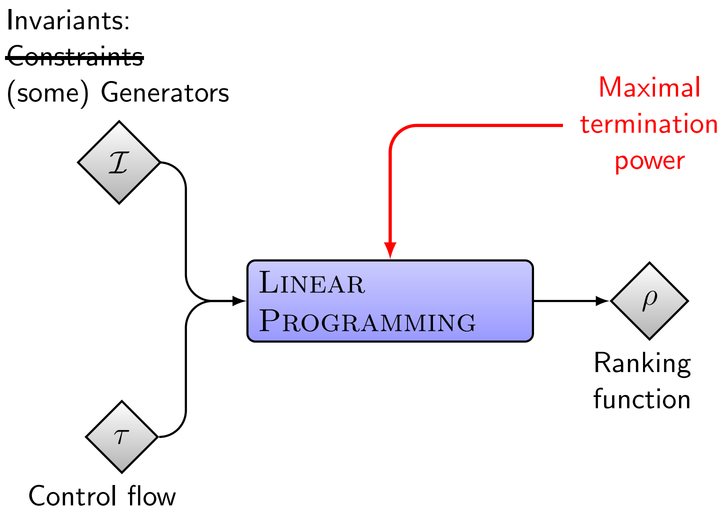
Incremental generation of constraints



Lazy termination

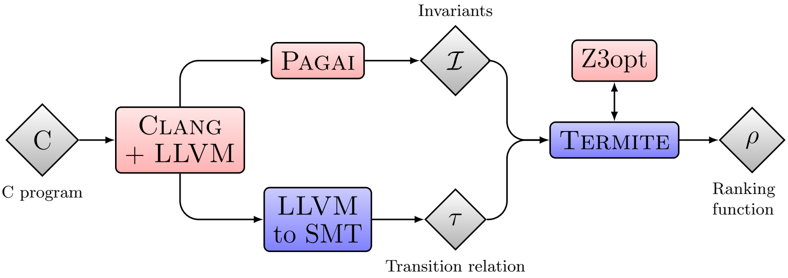


Lazy termination



Experiments

Implementation: `http://termite-analyser.github.io/`



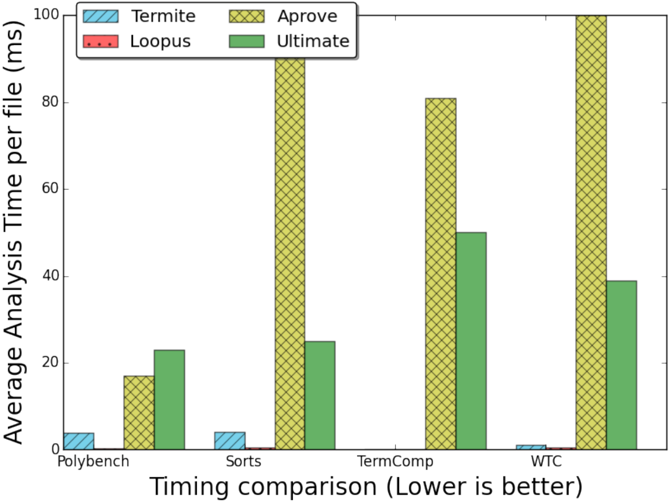
- **Benchmarks**: PolyBench, sorts, TermComp, WTC
- **Machine:** Intel(R) Xeon(R) @ 2.00GHz 20MB Cache.

Comparison: Linear Programming instances sizes

On WTC benchmark (average per file):

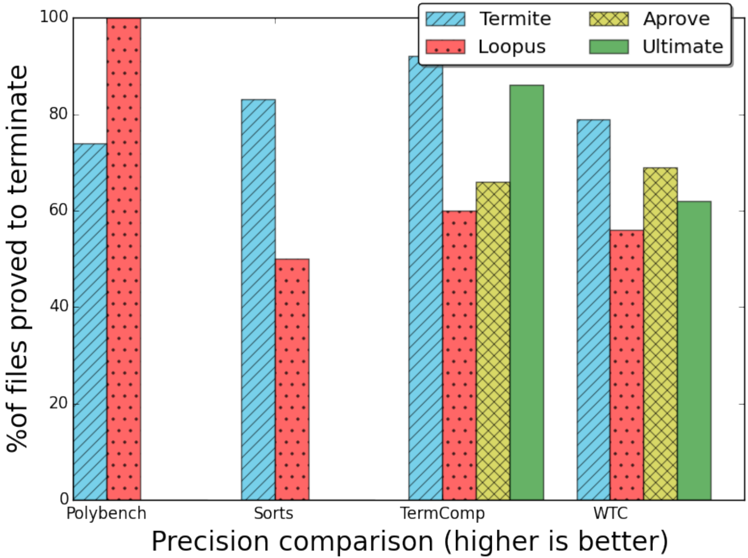
Tool	(constraints)	(variables)
RANK	584	229
TERMITE	5	2

Timing comparison



Timings exclude the front-end for TERMITE and LOOPUS.

Precision comparison



Conclusion of this part

From compilation to static analyses:

- Application domain: hardware synthesis.
- Adaptation of a scheduling algorithm to more general programs.
- Scaling static analyses techniques and apply to more realistic programs.
- Future work: back to scheduling (data structures).

Plan

Motivations

- Static analyses, examples
- Static analysis of software, how?

A new approach to software termination with compiler techniques

- Genesis of the first algorithm [SAS10]
- Toward scale and applicability [PLDI15]

Scaling abstract interpretation for more efficient compilers

Motivation

Classical analyses inside compilers:

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

Motivation

Classical analyses inside compilers:

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

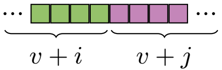
▶ Need for safe and precise quasi linear-time algorithms at **low-level**.

▶ Illustration with **pointer analysis**.

Less than information for pointers [CGO17,SCP17]

```
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;
  }
}
```

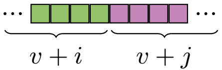
`v[i] = *(v+i)`



Less than information for pointers [CGO17,SCP17]

```
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)



- Range information is not sufficient to disambiguate v[i] and v[j].
- We need to propagate **relational information**.

Our setting for scaling analyses

Classical abstract interpretation analyses:

- Information attached to (*block*, *variable*).
- A new information is computed after each statement.

Sparse analyses ⇒ **Static Single Information (SSI) Property [Ana99]**:

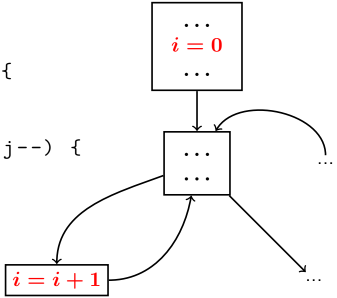
- Attach information to variables.
- The information must be invariant throughout the live range of the variable.

- A simple assignment breaks SSI!
- Work on suitable intermediate representations

Scaling analyses: program representation 1/2

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

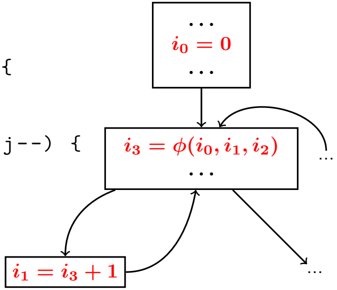
```
void partition(int *v, int N) {
  int i, j, p, tmp;
  p = v[N/2];
  for (i = 0, j = N - 1;; i++, j--) {
    while (v[i] < p) i++;
    ...
}
```



Scaling analyses: program representation 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++;
    ...
}
```

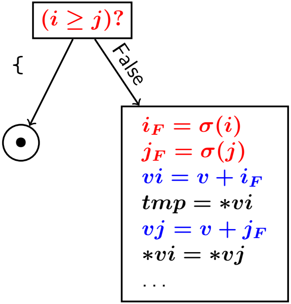


- Sparse storage of **value** information (one value range per variable name).

Scaling analyses: program representation 2/2

Within **SSA** form, tests 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.
- The σ renaming (**e-SSA**) enables to propagate "$i_F < j_F$".

Scaling analyses: relational information

Recall the SSI setting:

- Information must be invariant throughout the live range of the variable. ✓
- Attach information to variables (and not blocks).

▶ Work on semi-relational domains, for instance:

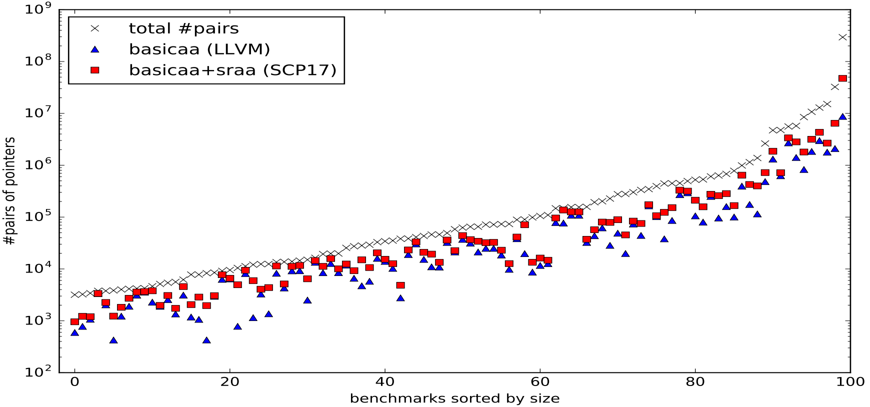
- Parametric ranges [OOPSLA14] $x \mapsto [0, N+1]$
- Pentagons [LF10]: $x \mapsto \{u, t\}$ means $u, t \leq x$.

Contributions on static analyses for pointers

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

- A new sequence of static analyses and associated queries.
- Based on semi-relational sparse abstract domains.
- Implemented in LLVM.
- 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.

Conclusion of this part

Static analyses for compilers:

- Application domain: code optimisation.
- Adaptation of abstract interpretation algorithms inside this particular context.
- Algorithmic and compilation techniques to scale.
- Future work: more relational domains (and data structures).

Perspectives

Cross fertilisation from/to different communities

- Scheduling and compilation techniques for **data-structures**:
  - expand the polyhedral model for trees, lists, ...
- Static analyses and optimised compilation for **dataflow programs**:
  - combine static scheduling and code optimisation.
- Tools: optimisation, constraint solving, rewriting...

Collaborations - Coauthors

- Lyon: Christophe Alias, Alain Darte, Paul Feautrier, Jean-Philippe Babau.
- Grenoble: Nicolas Halbwachs, David Merchat, David Monniaux, Pascal Raymond.
- Lille: Abdoulaye Gamatié, Vlad Rusu.
- Rennes: Benoît Combemale.
- Brasil: Fernando Pereira, Leonardo Barbosa, Henrique Nazaré, Izabela Maffra, Willer Santos, Leonardo Oliveira.
- UK: Peter Schrammel, Carsten Fuhs.
- PHD Student: Maroua Maalej.
- Students: Guillaume Andrieu, Gabriel Radanne, Raphael Rodrigues, Vitor Paisante, Ramos Pedro.

References I

[ADFG10] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord.
Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs.
In Proceedings of the 17th International Static Analysis Symposium (SAS'10), Perpignan, France, September 2010.

[ADFG13] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord.
Rank: a tool to check program termination and computational complexity.
In Workshop on Constraints in Software Testing Verification and Analysis (CSTVA'13), Luxembourg, March 2013.

[Ana99] Scott Ananian.
The static single information form.
Master's thesis, MIT, September 1999.

[Fea92] Paul Feautrier.
Some efficient solutions to the affine scheduling problem. part ii. multidimensional time.
International Journal of Parallel Programming, 21(6):389–420, Dec 1992.

[GMR15] Laure Gonnord, David Monniaux, and Gabriel Radanne.
Synthesis of ranking functions using extremal counterexamples.
In Proceedings of the 2015 ACM International Conference on Programming Languages, Design and Implementation (PLDI'15), Portland, Oregon, United States, June 2015.

[LF10] Francesco Logozzo and Manuel Fähndrich.
Pentagons: A weakly relational abstract domain for the efficient validation of array accesses.
Sci. Comput. Program., 75(9):796–807, 2010.

[MPMQPG17] Maroua Maalej, Vitor Paisante, Fernando Magno Quintao Pereira, and Laure Gonnord.
Combining Range and Inequality Information for Pointer Disambiguation.
Science of Computer Programming, 2017.
Accepted in Oct 2017, final published version of https://hal.inria.fr/hal-01429777v2.

References II

[MPR+17] Maroua Maalej, Vitor Paisante, Pedro Ramos, Laure Gonnord, and Fernando Pereira.
Pointer Disambiguation via Strict Inequalities.
In International Symposium of Code Generation and Optimization (CGO'17), Austin, United States, February 2017.

[PMB+16] Vitor Paisante, Maroua Maalej, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintao Pereira.
Symbolic Range Analysis of Pointers.
In International Symposium of Code Generation and Optimization (CGO'16), pages 791–809, Barcelon, Spain, March 2016.

[SMO+14] Henrique Nazaré Willer Santos, Izabella Maffra, Leonardo Oliveira, Fernando Pereira, and Laure Gonnord.
Validation of Memory Accesses Through Symbolic Analyses.
In Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages And Applications (OOPSLA'14), Portland, Oregon, United States, October 2014.