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:

```c
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:

```c
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.
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 ;
}
```

The two regions do not overlap.
Goal: performance 2/2

Enable code motion:

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

- If `p` and `p2` 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.

![Diagram showing acceptable behaviours and a program, with annotations](attachment:image.png)
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 impossible 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]
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

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

```c
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!)

**Ranking function and dependencies**
Inspiration: termination is scheduling 1/2

<table>
<thead>
<tr>
<th></th>
<th>Scheduling</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>function ($\rho$)</td>
<td>$\geq 0$</td>
<td>$\geq 0$</td>
</tr>
<tr>
<td>respects</td>
<td>dependencies</td>
<td>flow</td>
</tr>
<tr>
<td>$(W, da, db)$</td>
<td>$4r - (da + db)$</td>
<td>$da + db$</td>
</tr>
</tbody>
</table>

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) → system of constraints.
- Scheduling problem → system of constraints + objective function.

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

$$\rho(k, \vec{x}) = A_k \cdot \vec{x} + \vec{b}_k \in \mathbb{N}^d$$
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

Loop Iterators (polyhedra)

Dependencies

Minimal latency

Scheduling function

LINEAR PROGRAMMING
From scheduling to termination

Loop Iterators
Invariants
(polyhedra)

$\mathcal{I}$

$\tau$

Dependencies
Control flow

Linear Programming

$\rho$

Scheduling
Ranking function

Minimal latency
Maximal termination power
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;

\begin{itemize}
  \item For each control point \( k \):
    \[ \rho(k, \vec{x}) \geq 0 \text{ for } \vec{x} \in P_k \]
  \item For each transition:
    \[ \rho(\text{dest}, \vec{x}') < \rho(\text{src}, \vec{x}) \]
\end{itemize}

Searching for ranking function \( \rho \):

\[
\rho(\text{start}, \vec{x}) = \alpha_{\text{start}}^1 i + \alpha_{\text{start}}^2 N + \alpha_{\text{start}}^3 i_0 + \alpha_{\text{start}}^4 N_0 + \alpha_{\text{start}}^5
\]
\[
\rho(w, \vec{x}) = \alpha_{w}^1 i + \ldots
\]
\[
\rho(\text{stop}, \vec{x}) = \alpha_{\text{stop}}^1 i + \ldots
\]

\( \alpha_k^i \) are unknowns
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 $\rho = \begin{cases} 
  \text{start} & \rightarrow 2 + N_0 \\
  W & \rightarrow 1 + i \\
  \text{stop} & \rightarrow 0 
\end{cases}$

Problem solved.
### Experimental results: RANK

**Sorting** arrays of size $n$:

<table>
<thead>
<tr>
<th>Name</th>
<th>LOCs</th>
<th>Time (analysis)¹</th>
<th>dim</th>
<th>Worst Case Complexity Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion</td>
<td>12</td>
<td>0.2</td>
<td>3</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>selection</td>
<td>20</td>
<td>0.4</td>
<td>3</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>bubble</td>
<td>22</td>
<td>0.4</td>
<td>3</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>shell</td>
<td>23</td>
<td>1.1</td>
<td>4</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>heap</td>
<td>45</td>
<td>2.8</td>
<td>3</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

¹User time in seconds on a Pentium 2GHz with 1Gbyte RAM
Scaling the algorithm [PLDI15]

The former technique:

\[ LP \text{ Size} = O(\# \text{vars} \times \#Bblocks \times \#\text{transitions}) \]

- scalability: all basic blocks \( \mapsto \) big constraint systems
- precision: \( \rho \) must decrease at each transition.

New technique:

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

\[ \text{We do not compute all paths} \] explicitly (Counter example-based algorithm).
Incremental generation of constraints

- Program
- Initial Guess $\rho$

Try to find a contradiction.
- Is there a counterexample where $\rho$ strictly increases? (SMT)

- No
- Yes

Failure!

Refine: add counterexample

new $\rho$

The program Terminates!
Lazy termination

Invariants:
- Constraints
- Control flow

Linear Programming

Maximal termination power

Ranking function
Lazy termination

Invariants:

- Constraints
- (some) Generators

\( \mathcal{I} \)

Control flow

\( \tau \)

Linear Programming

Maximal termination power

\( \rho \)

Ranking function
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):

<table>
<thead>
<tr>
<th>Tool</th>
<th>(constraints)</th>
<th>(variables)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rank</td>
<td>584</td>
<td>229</td>
</tr>
<tr>
<td>Termite</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>
Timing comparison

Timings exclude the front-end for **TERMITE** and **LOOPUS**.

![Timing comparison chart]
Precision comparison

![Precision Comparison Diagram](image_url)

- **% of files proved to terminate**
- **Precision comparison (higher is better)**

Legend:
- Termite
- Aprove
- Loopus
- Ultimate
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**.
```c
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;
    }
}
```

Less than information for pointers \[\text{[CGO17,SCP17]}\]

- Range information is not sufficient to disambiguate \(v[i]\) and \(v[j]\).
- We need to propagate relational information.
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;
    }
}

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

```c
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.

```c
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!

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

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

\[
\begin{align*}
& \text{(i } \geq \text{ j)?} \\
& \text{False} \\
& \quad i_F = \sigma(i) \\
& \quad j_F = \sigma(j) \\
& \quad v_i = v + i_F \\
& \quad tmp = *v_i \\
& \quad v_j = v + j_F \\
& \quad *v_i = *v_j \\
& \quad \ldots
\end{align*}
\]
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.
• 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).
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.
- UK: Peter Schrammel, Carsten Fuhs.
- PHD Student: Maroua Maalej.
References I

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.

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. 

[Fea92] Paul Feautrier. 
Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. 

Synthesis of ranking functions using extremal counterexamples. 

Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. 

Combining Range and Inequality Information for Pointer Disambiguation. 
Accepted in Oct 2017, final published version of https://hal.inria.fr/hal-01429777v2.
References II

