Contributions to program analysis: expressivity and scalability

Laure Gonnord

N° d'ordre: 054-2017

MÉMOIRE D'HABILITATION À DIRIGER LES RECHERCHES
préparé au sein de
l'Université Claude Bernard Lyon 1

Spécialité: Informatique

Soutenu publiquement le 09/11/2017, par :
Laure DANTHONY GONNORD

Contributions to program analysis: expressivity and scalability

---

Devant le jury composé de :

Isabelle Guérin-Lassous, Professeure, Université Lyon1 Claude Bernard	Présidente
Paul H J Kelly, Professeur, Imperial College London, United Kingdom	Rapporteur
Antoine Miné, Professeur, Université Pierre et Marie Curie, France	Rapporteur
Andreas Podelski, Professeur, Université de Freiburg, Allemagne	Rapporteur
Albert Cohen, Directeur de Recherches, Inria Paris	Examinateur
Sebastian Hack, Professeur, Université de Saarland, Allemagne	Examinateur

Résumé

Cette dernière décennie a vu l'essor des analyses statiques plus seulement réservées aux systèmes embarqués critiques (avions, métros, fusées, etc.), mais maintenant disponibles plus largement, comme le montre l'implication de compagnies telles que Google ou Facebook dans le domaine.

Néanmoins, prouver l'absence de bugs d'un logiciel, (problème que nous savons depuis Turing et Cook être intrinsèquement difficile) n'est pas le seul challenge en développement logiciel. En effet, la complexité toujours croissante des logiciels induit un besoin toujours croissant d'optimisations fiables.

La résolution de ces deux problèmes (fiabilité, optimisation) impose le développement de méthodes d'analyse statique sûres (sans faux négatifs), efficaces (en temps et en mémoire), mais suffisamment précises (avec un taux faible de faux positifs). La recherche de tels compromis est au cœur de mes travaux, qu'ils s'appliquent à la vérification formelle ou au domaine plus contraint des analyses pour les compilateurs.

Les travaux détaillés dans ce manuscrit décrivent mes activités de recherche pendant la période de 2009 à 2017. Durant cette période, j'ai contribué aux développements d'analyses statiques appliquées aux deux domaines connexes que sont la vérification de logiciels et la compilation. Les analyses proposées sont développées dans le cadre de l'interprétation abstraite, pour des domaines numériques, et plus récemment, des domaines mémoire. Elles font de plus l'objet de validation expérimentale en terme de précision et de coût, ainsi que de leur impact dans des analyses clientes (terminaison, optimisations) lorsque cela est pertinent.

Summary

This last decade was the occasion to see the rise of static analyses that are now no longer reserved for critical embedded systems (airplanes, subways, rockets, etc.). They are now more widely available, as evidenced by the novel implication of companies like Google or Facebook in the domain.

Nevertheless, proving the absence of bugs in a given software (problem which has been known to be intrinsically hard since Turing and Cook) is not the only challenge in software development. Indeed, the ever growing complexity of software increases the need for more trustable optimisations.

Solving these two problems (reliability, optimisation) implies the development of safe (without false negative answers) and efficient (wrt memory and time) analyses, yet precise enough (with few false positive answers). Finding such compromises is the core of my works, whether it deals with formal verification or the more specific domain of analyses inside compilers.

The work which is detailed in this manuscript was done over a period going from 2009 to 2017. During this period, I contributed to static analyses in the two related domains that are software verification and compilation. The proposed analyses have been developed within the abstract interpretation framework, for numerical domains, or, more recently, memory domains. They are also experimentally validated, according to precision and cost criteria, and also according to their impact on client analyses (termination, code optimisation) whenever appropriate.

Acknowledgments

First of all, I am very grateful to Prof. Paul Kelly from Imperial College London (UK), Prof. Antoine Miné from Pierre et Marie Curie University (France) and Prof. A. Podelski from Freiburg university (Germany) for having kindly accepted to serve as reviewers on my Habilitation defense committee, and for the reports they wrote on my work.

I would also like to warmy thank Prof. Isabelle Guerin-Lassous from University of Lyon (France), Dr Albert Cohen from Inria Paris (France) and Prof Sebastien Hack from Saarland University (Germany) for their kind participation to my Habilitation defense committee as examiners, and for the interesting discussions we had during the defense.

A major part of the contributions presented in this document have been obtained in collaboration with colleagues from the following research groups: Verimag PACSS group in Grenoble (France), DaRT/Emeraude group in Lille (France), Compsys (Lyon), Roma (Lyon), and the Compilation Group in the Programmming Language Laboratory of the University of Mineas Gerais (Brasil). Many thanks to all of them!

I would like to make a special thank to Maroua Maalej, my first Phd student. The work we made together will without any doubt be the fundations of my future research, thank you for these last three years of collaboration.

I want to express especially my gratitude to my colleagues from ROMA. These two last years were undoubtedly the most exciting years of research, as well as team life and support.

This manuscript has been proof-read by many colleagues and friends, special thanks to Charlotte, David, Frédéric, Loris, Matthieu, Sylvie for your suggestions and encouragement.

I am also greatful to my colleagues and friends who helped me organise the defense and the defense party, Anne-Laure, Christophe, Sebastien, Nicolas, and other people that I already cited, thank you for your help and presence on D-Day.

Finally, many thanks to my family for your constant and precious support.

Contents

1 Introduction: from Program Verification to Program Optimisation

Software reliability has always been one of the major issues in Computer Science, as the spectacular accident of Ariane 5 in 1996 illustrated. The accident is described in these terms in the accident report[1]: "Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700m, the launcher veered off its flight path, broke up and exploded". The cause of this accident was the use of the code of the preceding rocket model, Ariane 4, which was not retested in the correct environment. In the last two decades, a series of accidents caused by software has led the NASA to refresh their validation methodology [28]. Other software related accidents were even more critical in terms of human lives, for instance the Panama radiotherapy accident [23] in 2000, that caused the death of 21 patients who received an overdose of radiotherapy treatment induced by a bug in the software.

Many complementary approaches exist to deal with software reliability, from software engineering, development methodology, formal methods based development, as well as testing and proofs [71]. All these techniques have proved their efficiency in the development of *safety critical* systems such as space rockets, airplanes, nuclear plant controllers, automatic subways. Among all the tools of this ecosystem, let us cite the ASTRÉE code analyser [19] which was capable of proving the absence of runtime errors of the "primary flight control software of an Airbus model"[2].

However, all these techniques cause a huge overhead on the software development process, and for the moment, they did not have a huge impact on the development of non critical systems such as laptop operating systems or hardware. This is especially due to the fact that:

- They have a huge impact on the development duration.
- They have a cost in terms of human resources.
- The analyses and tools were developed for the specific domain of safety critical systems, where code rules can be imposed to developers, and assumptions can be made on the shape of the code they produce. These constraints are clearly unreasonable in other contexts.
- The analyses and tools were developed for development environments that do not include aggressive code optimisation, since safety-critical systems are designed to be simple and usually run on over-sized platforms.

In the last decade nevertheless, the ever-growing number of static analysis tools demonstrated the need for safer (mode generalist, less critical) code as well as the visibility of the scientific community of static analysis[3]. Start-ups as well as major companies have developed their tools for the growing market of software validation. Among the commercial tools we can cite GrammaTech's Code Sonar[4]

[1] http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html
[2] https://www.absint.com/astree/index.htm
[3] https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis
[4] https://www.grammatech.com/products/codesonar

which focuses on code security and Facebook's Infer[5] that provides a precise diagnostic of memory management.

All these techniques and tools mainly prove the absence of bugs in their specialised area of application, or gives hints to the developer for a better understanding of his code, but they address only a part of the challenges toward software reliability. I believe that the effort to design a better code and prove the absence of errors has to be continued in two complementary directions: applicability (*static analysis for every developer*) and expressivity (*static analysis for everyday problems*). My work on static analyses for verification and compilation follows this philosophy: static analyses must verify but also be used to optimise runtime and memory footprint of programs. However, static analyses inside compilers are lagging far behind the state-of-the-art of static analyses for software verification or bug tracking. Clearly, the usage in compilers avoid the use of *unsound* techniques (all faulty behaviours should be detected); moreover, the complexity of the compiler code itself advocates for the design of "easy-to-implement" static analyses and code optimisations (bugs in compilers are frequent and a major issue [103]).

The success story of the Compcert compiler[6] and the Verasco[7] project show that we can design certified (proved by the Coq theorem prover) compilers and static analysers for general C programs. However the optimisations that are proposed in these two tools are still very far from the current production compilers, in which aggressive and clever optimisations are made. To bridge the gap between the two communities (static analysis and code optimisation), we propose to design abstract-interpretation based static analyses, which are correct by-construction (as opposed to formally validated in Coq) and scalable enough to provide useful information for further optimisations.

My current research concerns the development of dedicated static analyses for verification and compilation, that are capable to express *complex properties* at a *reasonable cost*. My contributions were driven by the idea that the world of verification and the world of compilation do not sufficiently use their cross-fertilisation potential. This was declined in all topics, from my Phd [DG07] where verification of *embedded systems* was the only application domain to the synthesis of numerical invariants, to my current activities where *compute-intensive code (High Performance Computing kernels)* as well as *general purpose programs* are under concern.

1.1 Static analysis for verification

Program analysis for verification aims at automatically checking that programs fit their specifications, these specification being explicit ("this program sorts an array") or implicit ("this program does not make any array out-of-bounds access"). In the general case, a perfect (sound and complete) program analysis is impossible (due to the undecidability of the halting problem). In order to render it possible, at least one of the following must hold: unsoundness (some violations of the specification are not detected), incompleteness (some correct programs are rejected because spurious violations are detected), or the state space should be finite (and not too large, so as to be enumerated explicitly or implicitly).

Among various techniques for program analysis, *Abstract interpretation* [37] is sound, but incomplete: it over-approximates the set of behaviours of the analysed program; if the over-approximated set contains incorrect behaviours that do not exist in the concrete program, then false alarms are produced. A central question in abstract interpretation for verification is to reduce the number of false alarms, while keeping memory and time costs reasonable [19].

[5] http://fbinfer.com/
[6] http://compcert.inria.fr/
[7] http://verasco.imag.fr/wiki/Main_Page

1.2 Static analysis for compilation of general purpose software

The rise of embedded systems and high performance computers in the last decade has revealed an increasing need for code optimisation. The applications are diverse, from compute-intensive embedded applications to massively parallel scientific code on large grids, as well as general purpose programs like web browsers or system schedulers.

Program analysis has two different usage in compilers:

- reject the program or print warnings if the program is not well-formed or has *undefined* behaviour.
- compute information and propagate to further *code optimisation/transformation*.

The central question in program analysis for optimising compilers is to design specialised analyses that scale well, while remaining expressive. In addition, these analyses must remain simple enough to be implemented in production compilers and usable for client analyses inside these compilers (like loop transformations to optimise locality, scheduling, automatic parallelization *etc.*).

1.3 Challenges for static analyses

The success of abstract interpretation for *safety-critical programs* demonstrates the robustness of the framework as well as its capability to provide *precise* analysers for complex properties (from array out-of-bounds to non interference of variables in parallel programs). However, if we want to cope with more generalist programs, or use abstract-interpretation invariant generation inside production compilers, there still remains numerous challenges that are partially addressed in this manuscript.

- Analyses must apply to more generalist programming languages (as opposed to a very specific subset of C or Java) ([SMO+14, GMR15], ...), or, at the opposite, be specially designed for domain-specific languages ([GG11], [FGG12]).
- Analyses must be scalable, in order to deal with the increasing size of programs, and to be used as pre-analyses for code optimisation [SMO+14, PMB+16, MPR+17], ...).
- Nevertheless, analyses must remain expressive enough to be capable of deriving precise information when necessary ([MG11, MG16], ...)

1.4 Related publications

For the sake of brevity and uniformity, this manuscript does not cover the publications related to my Phd ([DG07], [GH06], [HMG06], [GS14]), or made during my master studies [GHR04] and my postdoctoral studies ([GB09], [GB09], [GB08]). The work we made on theoretical aspects of Domain Specific Languages [RGC11] is also not exposed in this manuscript.
This document is a synthesis of the research articles, journals, or research reports, listed below.

Numerical Domains

[FGG12] Paul Feautrier, Abdoulaye Gamatié, and Laure Gonnord. Enhancing the Compilation of Synchronous Dataflow Programs with a Combined Numerical-Boolean Abstraction. *CSI Journal of Computing*, 1(4):8:86–8:99, 2012. Unknown impact factor.

[GG11] Abdoulaye Gamatié and Laure Gonnord. Static analysis of synchronous programs in signal for efficient design of multi-clocked embedded systems. In *Proceedings of the Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES 2011)*, Chicago, USA, April 2011. Acceptation rate 33% (17/51)

[MG11] David Monniaux and Laure Gonnord. Using bounded model checking to focus fixpoint iterations. In *Proceedings of the 18th International Static Analysis Symposium, SAS'11*, Venice, Italy, September 2011. Springer. Acceptation rate 32% (22/67)

[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. Selection rate 27% (52/186)

Termination analyses

[AAG12] Guillaume Andrieu, Christophe Alias, and Laure Gonnord. SToP: Scalable termination analysis of (C) programs (tool presentation). In *International Workshop on Tools for Automatic Program Analysis (TAPAS'12)*, Deauville, France, September 2012.

[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. Springer. Selection rate 37% (22/58)

[GAA12] Guillaume Andrieu, Christophe Alias and Laure Gonnord Modular termination of C programs. Research Report RR-8166, INRIA, December 2012.

[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. Selection rate 19% (58/303)

[RAPG14] Raphael Ernani Rodrigues, Péricles Alves, Fernando Pereira, and Laure Gonnord. Real-world loops are easy to predict : a case study. In *Workshop on Software Termination*, Vienne, Austria, July 2014.

Memory Analyses

[MG16] David Monniaux and Laure Gonnord. Cell morphing: from array programs to array-free Horn clauses. In Xavier Rival, editor, *23rd Static Analysis Symposium (SAS 2016)*, Static Analysis Symposium, Edimbourg, United Kingdom, September 2016. Selection rate 38% (21/55)

[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.

[MPR+17] Maroua Maalej, Vitor Paisante, Pedro Ramos, Laure Gonnord, and Fernando Pereira. Pointer Disambiguation via Strict Inequalities. In *International Symposium on Code Generation and Optimisation*, Austin, United States, February 2017. Selection rate 22% (26/116)

[PMB+16] Vitor Paisante, Maroua Maalej, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintao Pereira. Symbolic Range Analysis of Pointers. In *International Symposium on Code Generation and Optimization*, pages 791–809, Barcelon, Spain, March 2016. Selection rate 23% (25/108).

[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. Selection rate 27% (52/186)

1.5 Outline of the document

This manuscript develops the contributions made in the domain of static analysis. Each chapter exposes a different application domain and the techniques we developed for enhancing precision, scalability and applicability:

- Chapter 2 exposes our works around the correct design of static analyses for numerical programs (in our context, numerical programs are programs that manipulate integer variables), with a particular focus on scalability and applicability inside specialised or general purpose compilers.
- Chapter 3 summarises our contributions on proving termination of sequential numerical programs. This chapter illustrates how verification can take advantage of compilation techniques.
- Chapter 4 reports our works on the effective design of static analyses to deal with memory properties inside production compilers, and a more fundamental work on proving complex array properties.
- Chapter 5 gives the conclusion and draws future research directions.

2 Numerical domains

Since the seminal paper [37], abstract interpretation has proved its success for the verification of safety properties. The success story of ASTRÉE [19] shows the applicability of the method for safety critical systems (avionics). However, there still remain numerous motivations to continue working on the effective design of numerical domains:

- Classical abstract interpretation suffers from a lack of semantic-driven iteration heuristics (the algorithm usually scans the program under analysis in a deep-first search manner, regardless the meaning of the program itself). Some attempts have been made to design clever iteration strategies [36]. However they often fail when the invariant to be proved is disjunctive. Like abstract acceleration [DG07], semantic driven strategies would enable to gain both in terms of precision and time.

- The expressivity potentiality of some numerical analyses are not exploited as they could be. Synchronous languages compilers are an example of such missed opportunity of code optimisation. From the beginning of Lustre [62] and Signal [78], the choice has been make to design syntax directed simple code generation, and to avoid clever analyses as much as possible, in order to simplify the validation process of the compiler itself. However there is a need for more powerful analyses, especially to be able to compile for parallel platforms, while remaining simple enough to be embedded in synchronous compilers.

- Production compilers (GCC, LLVM, ICC) do not currently use the most advanced techniques/ abstract domains produced by the static analysis community in the last decades. Despite the original relationship between *dataflow* analyses inside compilers (liveness of variables, constant propagation), the actual number of abstract-interpretation based analyses actually implemented in production compilers are very few, mainly because of their algorithmic complexity [1]. Thus, designing numerical abstract domain that scale enough to be embedded in production compilers still remain a challenge.

Summary and Outline This chapter summarises the contributions made to improve expressivity, scalability and applicability of numerical abstract domains.
Section 2.1 describes the model of programs we use and the main context, which is abstract interpretation, and SMT-solving. Some of the tools we use are described in Section 2.2.
Then, the chapter presents two contributions that follow the recent advances in SMT solving and combine abstractions and SMT-solving:

- Section 2.3 explains how to use SMT-solvers to drive the fixpoint iterations in order to get more precise numerical invariants.

[1] While waiting an hour for a safety critical program to be analysed is accepted, in general it is not the case for a one-hour lasting compilation for a general purpose software package

- Section 2.4 shows how a new Numerical Boolean abstraction can improve the capabilities of synchronous language compilers.

The last contribution of the chapter, described in Section 2.5 is a new abstract domain based on intervals, whose design focuses on the effectivity as well as expressivity.

2.1 Model of programs, abstract interpretation, SMT-solving

We write matrices with capital letters (as A) and column vectors in boldface (as $\boldsymbol{x}$). If $\boldsymbol{x}$ has dimension d, its components are denoted $\boldsymbol{x}[i]$, with $0 \leq i < d$. Thus, its i-th component is $\boldsymbol{x}[i-1]$. Sets are represented with calligraphic letters such as $\mathscr{W}$, $\mathscr{K}$, etc.

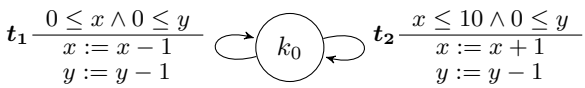
2.1.1 Integer Interpreted Automata

In the tradition of most previous work on program termination and static analysis, we first transform the program to be analysed into an abstraction: the associated *integer interpreted automaton*. This is similar to the flowcharts used a long time ago to express programs (see, e.g., Manna's book [80]) until the advent of structured programming. In fact, when one looks at real-life programs, many deviations from the strict structured model can occur, including premature loop termination, exceptions, and even the occasional `goto`. Reasoning with flowcharts abstracts the details of the syntax and semantics of the source language, which can be dealt with by an appropriate pre-processor.
In our work, a program is represented by an *affine (integer) interpreted automaton*.

Definition 1 (affine (integer) interpreted automaton)**.** *An affine (integer) interpreted automaton (AIA) is a tuple $\mathscr{A} = (\mathscr{K}, n, k_{init}, \mathscr{T})$ defined by:*

- *a finite set $\mathscr{K}$ of* control points*;*
- *n integer variables represented by a vector $\boldsymbol{x}$ of size n;*
- *an initial control point $k_{init} \in \mathscr{K}$;*
- *a finite set $\mathscr{T}$ of 4-tuples (k, g, a, k'), called* transitions*, , where $k \in \mathscr{K}$ (resp. $k' \in \mathscr{K}$) is the source (resp. target) control point, $g : \mathbb{Z}^n \mapsto \mathbb{B} = \{true, false\}$, the* guard*, is a logical formula expressed with affine inequalities $G\boldsymbol{x} + \boldsymbol{g} \geq 0$, and $a : \mathbb{Z}^n \mapsto \mathbb{Z}^n$, the* action*, assigns, to each variable valuation $\boldsymbol{x}$, a vector $\boldsymbol{x}'$ of size n, expressed by an affine expression $\boldsymbol{x}' = A\boldsymbol{x} + \boldsymbol{a}$. Here, G and A are matrices, $\boldsymbol{g}$ and $\boldsymbol{a}$ are vectors.*

To represent non-determinism or to approximate non-affine or non-analysable assignments in the program, we may have to assign the value "?", representing an arbitrary integer, to a variable, but we will not elaborate on this point.



Semantics The set of states is $\Sigma = \mathscr{K} \times \mathbb{Z}^n$. A *trace* from $(k_0, \boldsymbol{x}_0)$ to $(k, \boldsymbol{x})$ is a sequence of states $(k_0, \boldsymbol{x_0}), (k_1, \boldsymbol{x}_1), \ldots, (k_p, \boldsymbol{x}_p)$ such that $k_p = k$, $\boldsymbol{x}_p = \boldsymbol{x}$ and for each i, $0 \leq i < p$, there exists in $\mathscr{T}$ a transition (k_i, g_i, a_i, k_{i+1}) such that $g_i(\boldsymbol{x}_i) = \text{true}$ and $\boldsymbol{x}_{i+1} = a_i(\boldsymbol{x}_i)$. Given an initial valuation $\boldsymbol{v}$, a state $(k, \boldsymbol{x})$ is *reachable from* $\boldsymbol{v}$ iff (if and only if) there is a trace from $(k_{init}, \boldsymbol{v})$ to $(k, \boldsymbol{x})$. A state $(k, \boldsymbol{x})$ is *reachable* if there exists $\boldsymbol{v} \in \mathbb{Z}^n$ such that $(k, \boldsymbol{x})$ is reachable from $\boldsymbol{v}$. The set of reachable states is denoted by $\mathscr{R}$.

Invariants The guard g in a transition $t = (k, g, a, k')$ gives a necessary condition on variables $\boldsymbol{x}$ to traverse the transition t and to apply its corresponding action a. To get the exact valuations $\boldsymbol{x}$ of variables for which the action a can be performed, one would need to take into account the initial valuations and the successive conditions that led to the control point k. We denote by $\mathscr{R}_k$set of possible valuations $\boldsymbol{x}$ of variables when the control is in k:

$$\mathscr{R}_k = \{\boldsymbol{x} \in \mathbb{Z}^n \mid (k, \boldsymbol{x}) \in \mathscr{R}\}.$$

Then, there exists a trace containing the transition (k, g, a, k') iff $\boldsymbol{x} \in \mathscr{R}_k$ and $g(\boldsymbol{x})$ is true. Note that $\mathscr{R}_k$ does not depend on any initial valuation. More precisely, it is the union, for all initial valuations $\boldsymbol{v}$, of the set of vectors $\boldsymbol{x}$ such that $(k, \boldsymbol{x})$ is reachable from $\boldsymbol{v}$. In practice, it is difficult to determine the set $\mathscr{R}_k$ exactly but it is possible to give over-approximations, thanks to the notion of *invariants*. An invariant on a control point k is a formula $\phi_k(\boldsymbol{x})$ that is true for all reachable states $(k, \boldsymbol{x})$. It is *affine* if it is the conjunction of a finite number of affine conditions on program variables. The set $\mathscr{R}_k$ is then over-approximated by the integer points within a polyhedron $\mathscr{P}_k$. To compute invariants, we rely on standard *abstract interpretation* techniques, widely studied since the seminal paper of Cousot and Halbwachs [42]. These sets $\mathscr{P}_k$ represent all the information on the values of variables that can be deduced from the program by state-of-the-art analysis techniques.

2.1.2 Background and Notations in Abstract Interpretation

Let us denote by $\tau_{k,k'}$ the concrete set semantics of the edge (k, k'), ie a function that maps a set of states before the transition (k, k') to the set of states after the transition. Thus given a control point k, the set $\mathscr{R}_k$ of reachable values at this control point is the least solution of a system of semantic equations [41]:

$$\mathscr{R}_k = I_k \cup \bigcup_{(k',k)\in\mathscr{T}} \tau_{(k',k)}(\mathscr{R}_{k'}),$$

where I_k designs the possible initial values at control point k, i.e., in our case $I_k = \begin{cases} \mathbb{Z}^n & \text{if } k = k_{init} \\ \emptyset & \text{else.} \end{cases}$.

Abstract interpretation replaces the concrete sets of states $\mathscr{R}_k$ by elements of an abstract domain D. In lieu of applying exact operations τ to sets of concrete program states, we apply abstract counterparts $\tau^\sharp$.[2] An abstraction $\tau^\sharp$ of a concrete operation τ is deemed to be correct if it never "forgets" states:

$$\forall X \in D \; \tau(X) \subseteq \tau^\sharp(X) \tag{2.1}$$

We also assume an "abstract union" operation $\sqcup$, such that $X \cup Y \subseteq X \sqcup Y$. For instance, Σ can be $\mathbb{Q}^n$, D can be the set of convex polyhedra and $\sqcup$ the convex hull operation [11, 42, 61].

Solving the abstract system In order to find an inductive invariant [3], one solves a system of abstract semantic inequalities (X_k denotes possible valuations at control point k):

$$\begin{cases} \forall k \; I_k \subseteq X_k \\ \forall (k', k') \in \mathscr{T} \; \tau^\sharp_{(k',k)}(X_{k'}) \subseteq X_k. \end{cases} \tag{2.2}$$

Since the $\tau^\sharp_e$ are correct abstractions, it follows that any solution of such a system defines an inductive invariant; one wishes to obtain one that is as strong as possible ("strong" meaning "small with respect to $\subseteq$"), or at least sufficiently strong as to imply the desired properties.

[2] Many presentations of abstract interpretation distinguish the abstract element $x^\sharp \in D$ from the set of states $\gamma(x^\sharp)$ it represents. We opted not to, for the sake of brevity.

[3] An inductive invariant is a formula which is true at the initial state(s) of the transition system, and which is stable: if the valuation of variables satisfy this formula at a given control point, then after each possible transition, the new valuation also satisfies this formula.

Assuming that all functions $\tau_e^\sharp$ are monotonic with respect to $\subseteq$, and that $\sqcup$ is the least upper bound operation in D with respect to $\subseteq$, one obtains a system of monotonic abstract equations:

$$X_k = I_k \sqcup \bigsqcup_{(k',k)\in\mathcal{T}} \tau^\sharp_{(k',k)}(X_{k'}).$$

If $(D,\subseteq)$ has no infinite ascending sequences ($d_1 \subsetneq d_2 \subsetneq \ldots$ with $d_1, d_2, \cdots \in D$), then one can solve such a system by iteratively replacing the contents of the variable on the left hand side by the value of the right hand side, until a fixed point is reached. The order in which equations are iterated does not change the final result ([39], "cahotic iterations" chapter).

Solving with extrapolation Many interesting abstract domains, including that of convex polyhedra, have infinite ascending sequences. One then classically uses an extrapolation operator known as $\triangledown$ and denoted by $\triangledown$ in order to enforce convergence within finite time. The iterations then follow the "upward iteration scheme":

$$X_k := X_k \triangledown \left(X_k \sqcup \bigsqcup_{(k',k)\in\mathcal{T}} \tau^\sharp_{(k',k)}(X_{k'}) \right) \tag{2.3}$$

where the contents of the left hand side gets replaced by the value of the right hand side. The convergence property is that any sequence u_n of elements of D of the form $u_{n+1} = u_n \triangledown v_n$, where v_n is another sequence, is stationary [41]. It is sufficient to apply widening only at a set of program control nodes P_W such that all cycles in the control flow graph are cut. Then, through a process of *chaotic iterations* [39, Def. 4.1.2.0.5, p. 127], one converges within finite time to an inductive invariant satisfying Rel. 2.2.

Definition 2 (Widening for intervals)**.** *The widening operator on intervals is defined as* $[x_l, x_r] \triangledown [x'_l, x'_r] = [x"_l, x"_r]$ *where* $x"_l = \begin{cases} x_l & \text{if } x_l = x'_l \\ -\infty & \text{else.} \end{cases}$, *and* $x"_r = \begin{cases} x_r & \text{if } x_r = x'_r \\ +\infty & \text{else.} \end{cases}$.

Algorithm 1 Abstract Interpretation: classic Algorithm

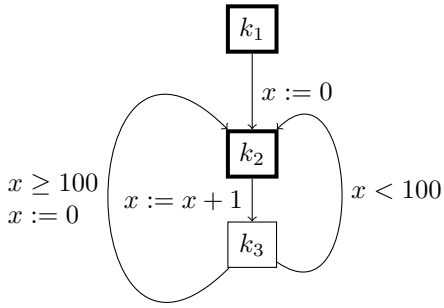
```
1:  A ← {k_init};                                   ▷ Initialise the work-list A
2:  while A is not empty do                         ▷ Fixpoint Iteration
3:      Choose k1 ∈ A
4:      A ← A \ {k1}
5:      for all outgoing edge (e) from k1 do
6:          Let k2 be the destination of e :
7:          if k2 ∈ P_W then
8:              X_temp ← X_k2 ▽ (X_k2 ⊔ τ#_e(X_k1))         ▷ Widening node;
9:          else
10:             X_temp ← X_k2 ⊔ τ#_e(X_k1) ;
11:         end if
12:         if X_temp ⊈ X_k2 then                   ▷ The value must be updated
13:             X_k2 ← X_temp;
14:             A ← A ∪ {k2};
15:         end if
16:     end for;
17: end while;                                      ▷ End of Iteration
18: Possibly narrow.
19: return all X_ki s;
```

Algorithm, implementation A naive implementation of the upward iteration scheme described above is to maintain a work-list of program points k such that X_k has recently been updated and replaced by a strictly larger value (with respect to $\subseteq$), pick and remove the foremost member k, apply the corresponding rule $X_k := \ldots$, and insert into the work-list all k' such that $(k, k') \in \mathcal{T}$ (This algorithm is formally described in Algorithm 1).
This algorithm also performs an improvement of the obtained invariant; once an inductive invariant is found, it is possible to improve it by iterating the $\psi^\sharp$ function defined as $Y = \psi^\sharp(X)$, noting $X = (X_k)_{k\in\mathcal{K}}$ and $Y = (Y_k)_{k\in\mathcal{K}}$, with $Y_k = I_k \sqcup \bigsqcup_{(k',k)\in\mathcal{T}} \tau^\sharp_{(k',k)}(X_{k'})$. If X is an inductive invariant, then for any n, ${\psi^\sharp}^n(X)$ is also an invariant. This technique is an instance of *narrowing iterations*, which may help recover some of the imprecision induced by widening [41, §4].

Example 1 (Simple example for abstract interpretation)**.** *Consider the simple example of Figure 2.1.*

```
int x = 0;
while (true) {
    x = x+1;
    if (x >= 100) x = 0;
}
```

(a) Listing for the simple example



(b) Its CFG

Figure 2.1 – A simple example

The classic algorithm (with the interval abstract domain) performs on this control flow graph (Control Flow Graph (CFG)) the following iterations :

- *Initialisation :* $X_{k_1} \leftarrow (-\infty, +\infty)$, $X_{k_2} \leftarrow X_{k_3} \leftarrow X_{k_4} \leftarrow \emptyset$.
- *Step 1:* $X_{k_2} \leftarrow [0,0]$, *then the transition to* k_3 *is enabled,* $X_{k_3} \leftarrow [1,1]$, *then the return edge to* k_2 *gives the new point* $x = 1$ *to* X_{k_2}, *the new interval is then* $X_{k_2} = [0,1]$ *after performing the convex hull. Widening according to Definition 2 gives the interval* $X_{k_2} = [0,\infty)$.
- *Step 2:* X_{k_3} *becomes* $[1,+\infty)$. *The second transition from* k_3 *to* k_2 *is thus enabled, and the back edge to* k_2 *gives the point* $x = 0$ *to* X_{k_2}. *At the end of step 2 the convergence is reached.*
- *The narrowing sequence on the interval* $X_{k_2} = [0,\infty)$ *performs an intersection with the constraint* $x < 100$. *After propagation, we end up with the stable invariant* $X_{k_2} = [0,100]$.

In the rest of the manuscript, we will denote by $\mathcal{I}_k$ any inductive invariant at control k.

2.1.3 SMT-solving

Satisfiability modulo theory (SMT) solving consists in deciding the satisfiability of a first-order formula with unknowns and relations lying in certain theories. For instance, the following formula taken from [83]:

$$(x \leq 0 \vee x + y \leq 0) \wedge y \geq 1 \wedge x \geq 1$$

has no solution in $\mathbb{R}^2$. A SMT-solver reports whether a formula is satisfiable, and if so, may provide a model of this satisfaction, for instance $(x = 0, y = 1)$ is such a model for $(x \leq 0 \vee x + y \leq 0) \wedge y \geq 1$.
SMT encompasses various theories as diverse as character strings, inductive data structures, bit-vector arithmetic, and ordinary differential equations. In all the following work however, we focus on the numerical fragments such as quantifier-free linear integer arithmetic (QF_LIA) or quantifier-free linear real arithmetic (QF_LRA) like in the previous example. These theories subsume Boolean satisfiability (SAT), the canonical NP-complete problem. However they remain decidable.

The reader might refer to [83], [17] and [76] for a more complete introduction on SMT and the different state-of-the-art solvers/techniques, as well as its applications to program analysis.

2.2 Tools for Abstract Interpretation and SMT-solving

2.2.1 Computing numerical invariants from C-programs

There are many tools for the computations of numerical invariants. Here is a non exhaustive list of some recent ones:

- NBAC[4] implements the classic LRA in combination to dynamic partitioning ([72]). Contrary to ASPIC, the tool is dedicated to the verification of properties of LUSTRE programs. The method performs forward and backward analysis from a minimal control structure, and the CFG is partitioned w.r.t. the analysis results (and the proof goal). Our technique can be used to improve the precision of invariants during each forward/backward analysis.

- LASH[5] and FASTER[6] use acceleration techniques to compute, when possible, the exact reachability sets of counter automata. Theoretical results concerning the acceleration of some subclasses of loop have been obtained this last ten years (for difference bound constraints [35], a subclass of affine guarded functions [22, 52] and more recently for octagonal relations [24]). However, the tools based on these algorithms are not fully automatic (LASH), or are not guaranteed to terminate (FASTER), in particular for nested loops.

- STING[7], and INVGEN[8] use a combination of LRA and Farkas' lemma to discover numerical invariants. The main drawback of the method is the use of template invariants, which prevents the analysis to discover any invariant which is not of the right form. To improve the precision, INVGEN performs an execution of the program to add some additional constraints, which increase the global analysis time.

- There are many C parsers and experimental compilers, including CIL [85], LLVM [77] and Suif [108]. Extracting an automaton from their intermediate representation, let alone do the complex approximations and transformations that are necessary prior to precise numerical invariant generation is not a straightforward activity. Moreover, at the time of the RANK toolchain design (see Chapter 3), LLVM was not mature enough to make the bet of designing a preprocessing based on it.

- Pagai [67] is a numerical invariant generation designed at Verimag (Grenoble) since 2012. It implements the more recent developments in abstract interpretation with numerical abstract domains, such as [65]. Pagai performs classic abstract interpretation on the LLVM intermediate representation.

During my PhD and since then, I maintain ASPIC, a numerical polyhedral invariant generator for interpreted automata (based on a variation on abstract interpretation called *abstract acceleration*, that was the main contribution of my Phd thesis). In 2009/2010, Paul Feautrier made a C parser that is able to generate automata in the ASPIC input format. We used this tool chain for some of the works detailed in the manuscript. Some implementation details are described in [FG10].

[4] http://pop-art.inrialpes.fr/~bjeannet/nbac/index.html
[5] http://www.montefiore.ulg.ac.be/~boigelot/research/lash
[6] http://www.lsv.ens-cachan.fr/fast/
[7] http://theory.stanford.edu/~srirams/Software/sting.html
[8] http://www.model.in.tum.de/~guptaa/invgen/

2.2.2 SMT-solvers

SMT solvers can be used as a library, from an application programming interface, typically from C/C++ or Python, or as an independent process, that reads and writes outputs from a common format called SMT-lib[12]. In the prototypes we developed we often used the SMT-lib format to first prototype examples, then either we chose to use their API (TERMITE- see section 3.2.3 - uses Z3's API) or a *pipe* to the SMT-lib format (this is the case for SYNC2SMT in Section 2.4) which uses a Yices version of the format).

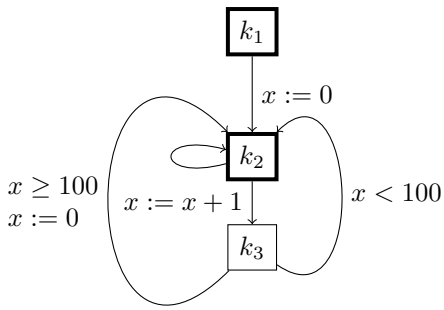
2.3 Enhancing fixpoint iterations with SMT-solvers

In this section, I detail the contribution of the SAS'11 paper [MG11]. The method is based on the study of the two following examples, where classic abstract interpretation with widening and narrowing fail to compute precise invariants. Here again, the abstract domain we choose is the interval abstract domain.

Example 2 (Motivating example 1)**.** *Consider Figure 2.2 that depicts a simplification of a fragment of an actual industrial reactive program: indexing of a circular buffer used only at certain iterations of the main loop of the program, chosen non-deterministically. Its representation with an affine automaton is depicted in the right of the figure.*

```
int x = 0;
while (true) {
  if (nondet()) {
    x = x+1;
    if (x >= 100) x = 0;
  }
}
```

(a) Listing for the circular buffer



(b) Its CFG

Figure 2.2 – The circular buffer example

If we perform the classic analysis like we did for the (very similar) Example 1, widening yields $[0, +\infty)$*, and this is not improved by narrowing because of the epsilon transition around* k_2*!* [9]
However, if we focus on the loop: `assume(nondet()); x = x+1; assume(x < 100);`*, we could iterate on this loop first, compute the result* $[0, 99]$ *and continue to perform the classic analysis, this invariant is inductive for the original loop, thus the analysis stops with this invariant. Our method will formalise this idea.*

Example 3 (Motivating example 2)**.** *Listing 2.1 depicts a C implementation of* $y = \sin(x)/x - 1$*, with the* $-0.01 \leq x \leq 0.01$ *range implemented using a Taylor expansion around zero in order to avoid loss of precision and division by zero as* $\sin(x) \simeq x \to 0$*.*
Consider the following listing:

Listing 2.1 – Motivating example 2

```
if (x >= 0) { xabs = x; } else { xabs = -x; }
```

[9] On this example, it is possible to compute the $[0, 99]$ invariant by so called "widening up-to" [60, Sec. 3.2], or with "thresholds" [19]: essentially, the analyser notices syntactically the comparison $x < 100$ and concludes that 99 is a "good value" for x, so instead of widening directly to $+\infty$, it first tries 99. This method only works if the interesting value is a syntactic constant.

```
if (xabs >= 0.01) {
  y = sin(x) / x - 1;
} else {
  xsq = x*x;  y = xsq*(-1/6. + xsq/120.);
}
```

In order to prove that there cannot be a division by zero in the first branch of the second if-then-else, one would need the non-convex property that $x \geq 0.01 \vee x \leq -0.01$. *An analysis representing the invariant at that point in a domain of convex properties (intervals, polyhedra, etc.) will fail to prove the absence of division by zero (incompleteness).*
Obviously, we could represent such properties using disjunctions of convex polyhedra [10], but this leads to combinatorial explosion as the number of polyhedra grows: at some point heuristics are needed for merging polyhedra in order to limit their number; it is also unclear how to obtain good widening operators on such domains. The same expressive power can alternatively be obtained by considering all program paths separately ("merge over all paths") and analysing them independently of each other. In order to avoid combinatorial explosion, the trace partitioning *approach [91] applies merging heuristics. In contrast, our method will rely on the power of modern SMT-solving techniques.*

We have seen two examples of programs where classic polyhedral analysis fails to compute good invariants. How could we improve on these results?

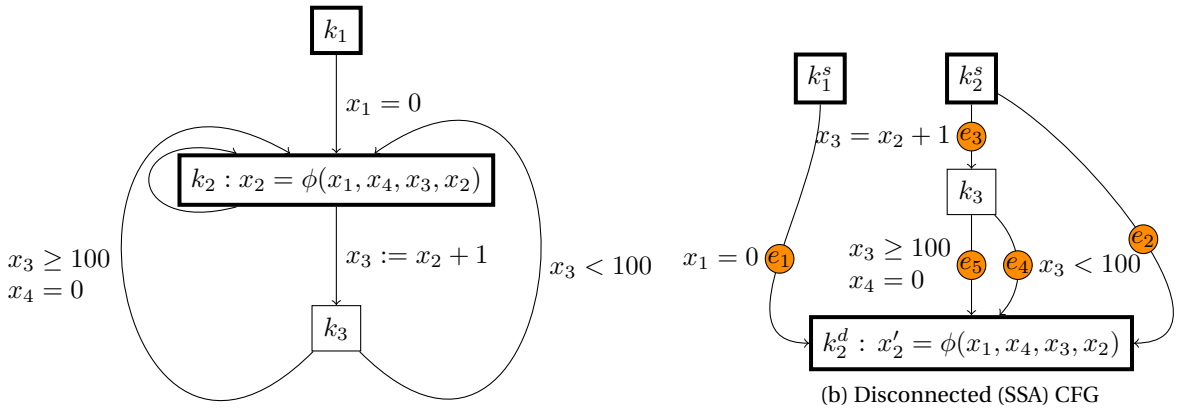
- In order to get rid of the imprecision in Example 3, one could "explode" the control-flow graph: in lieu of a sequence of n if-then-else, with n merge nodes with 2 input edges, one could distinguish the 2^n program paths, and having a single merge node with 2^n input edges. As already pointed out, this would lead to exponential blowup in both time and space.
- One way to get rid of imprecision of classic analysis on the program from Fig. 2.2 would be to consider each path through the loop at a time and compute a local invariant for this path. Again, the number of such paths could be exponential in the number of tests inside the loop.

The contribution of our article [MG11] is a generic method that addresses both of these difficulties. First, we encode the program in a way that enables the search for "interesting paths" for which we will compute a precise invariant. Then, we transform Algorithm 1 in order to integrate this "path focusing." In the rest of the section, I describe an instantiation of the algorithm on affine automata, but the article deals with arbitrary transitions systems.

2.3.1 Finding interesting paths in a program

We first give a way to express the concrete semantics of the program in term of an SMT formula called ρ. The Control Flow Graph of the program is first transformed into the Single Static Assignment (SSA) form, where all variables are *statically assigned once* [43]; standard techniques exist for converting to SSA.
As usual in abstract interpretation, we consider a set $K_W \subseteq \mathcal{K}$ of *widening nodes*. Let us take a superset K_R of K_W "*abstraction points*". K_R can be taken equal to K_W, or may include other nodes. In the sequel, the nodes in K_R are in bold.

Example 2 (continuing from p. 12)**.** *In Figure 2.3 we depict the transformations we do on the SSA CFG of the program of Figure 2.2 in order to encode its semantics inside a first-order affine formula. Figure 2.3(a) gives the SSA version of the CFG:* $x_2 = \phi(x_1, x_4, x_3, x_2)$ *is a merge SSA statement that encodes the fact that the* x_2 *variable is updated with one of the four values* x_1, x_2, x_3, x_4 *depending on the preceding node/transition: for instance, if the previous node was* k_1, *now* x_2 *has the value* x_1. *Figure 2.3(b) gives the disconnected SSA form graph where* k_1 *and* k_2 *have been split. The resulting formula is depicted below. For instance , the transition from* k_1^s *to* k_2^d *is encoded as* $(e_1 = (x_1 = 0) \wedge b_1^s)$ *and* $b_2^d = e_1 \vee \ldots$. *This kind of transformation is classic in model-checking.*



(a) SSA CFG

(b) Disconnected (SSA) CFG

$$\big(e_1 = (x_1 = 0) \wedge b_1^s) \wedge (e_2 = b_2^s \wedge \neg c_2^s) \wedge (e_3 = (x_3 = x_2 + 1) \wedge b_2^s \wedge c_2^s) \wedge (e_4 = b_3 \wedge x_3 < 100) \wedge (e_5 = b_3 \wedge x_3 \geq 100 \wedge x_4 = 0) \wedge (b_2^d = e_1 \vee e_4 \vee e_5 \vee e_2) \wedge (b_3 = e_3) \wedge (x_2' = ite(e_1, x_1, ite(e_5, x_4, ite(e_4, x_3, x_2)))\big)$$

Figure 2.3 – SSA CFG and its disconnected version for the motivating example, and the corresponding SMT formula (ρ). $ite(b, e_1, e_2)$ is "if b then the value of e_1 else the value of e_2". To each node k_x corresponds a Boolean b_x and an optional choice variable c_x; to each edge, a Boolean e_y.

In order to find a path from program point $k_1 \in K_R$, with variable state x_1, to program point $k_2 \in K_R$, with variable state x_2, we simply conjoin ρ with the formulas $x_1 \in X_{k_1}$ and $x_2 \notin X_{k_2}$, with x_1, x_2, expressed in terms of the SSA variables. For instance, if X_{k_1} and X_{k_2} are convex polyhedra defined by systems of linear inequalities, one simply writes these inequalities using the names of the SSA-variables at program points k_1 and k_2.
We apply SMT-solving over that formula. The result is either "unsatisfiable", in which case there is no path from k_1, with variable values x_1, to k_2, with variable values x_2, such that $x_1 \in X_{k_1}$ and $x_2 \notin X_{k_2}$, or "satisfiable", in which case SMT-solving also provides a model of the formula (a satisfying assignment of its free variables); from this model we easily obtain such a path, unique by construction of ρ.
Indeed, a model of this formula yields a trace of execution: those b_k predicates that are true designate the program points through which the trace goes, and the other variables give the values of the program variables.

2.3.2 Final algorithm: path focusing for abstract interpretation

Algorithm 2 consists in the iteration of the path finding method of Sec. 2.3.1, coupled with forward abstract interpretation along the paths found and, optionally, path acceleration.
The proof of correctness and termination is done in the paper. We also extend the algorithm, in the spirit of [GH06] on the special case of self loops, in order to perform more precise iterations. Let us illustrate this algorithm on the motivating example.

Example 2 (continuing from p. 12)**.** *Let us perform Algorithm 2 on the modified CFG of the running example, that we recall in Figure 2.4:*

- *Initialisation:* $K_R = \{k_1, k_2\}$, $A \leftarrow \{k_1\}$.
- *Step 1 : Is there a feasible path from control point* k_1 *to control point* k_2*? Yes. On the Figure, the obtained model corresponds to the transition from* k_1^s *to* k_2^d*, and leads to the interval* $X_{k_2} = [0, 0]$.
- *Step 2 : Is there a path from* k_2 *with* $x = 0$ *to* k_2 *with* $x \neq 0$*? Yes, there is such a path* (e_3, e_4)*, on which we now focus. This path is considered as a loop and we therefore do a local iteration*

Algorithm 2 Path-focused Fixpoint Iterations

1: Compute SSA-form of the control flow graph.
2: Choose K_R, compute the disconnected graph (P', E') accordingly.
3: $\rho \leftarrow$ computeFormula(P', E') ▷ Precomputations
4: $A \leftarrow \emptyset$;
5: **for all** $k \in K_R$ such that $I_k \neq \emptyset$ **do**
6: $\quad A \leftarrow A \cup \{k\}$
7: **end for**;
8: **while** A is not empty **do** ▷ Fixpoint Iteration on the reduced graph
9: $\quad$ Choose $k_1 \in A$
10: $\quad A \leftarrow A \setminus \{k_1\}$
11: $\quad$ **repeat**
12: $\quad\quad res \leftarrow \text{SmtSolve}\left(\rho \wedge b_{k_1} \wedge x_1 \in X_{k_1} \wedge \bigvee_{k_2 \mid (k_1,k_2) \in E'} \left(b_{k_2} \wedge x_2 \notin X_{k_2}\right)\right)$
13: $\quad\quad$ **if** res is not "unsat" **then**
14: $\quad\quad\quad$ Compute $e' \in E'$ from res ▷ Extraction of path from the model (§2.3.1)
15: $\quad\quad\quad Y \leftarrow \tau^{\sharp}_{e'}(X_{k_1})$ ▷ Computation of the "effect" of the path onto the abstract value
16: $\quad\quad\quad$ **if** $k_2 \in K_W$ **then**
17: $\quad\quad\quad\quad X_{temp} \leftarrow X_{k_2} \triangledown \left(X_{k_2} \sqcup Y\right)$ ▷ Final point k_2 is a widening point
18: $\quad\quad\quad$ **else**
19: $\quad\quad\quad\quad X_{temp} \leftarrow X_{k_2} \sqcup Y$
20: $\quad\quad\quad$ **end if**
▷ at this point $X_{temp} \not\subseteq X_{k_2}$ otherwise k_2 would not have been chosen
21: $\quad\quad\quad X_{k_2} \leftarrow X_{temp}$
22: $\quad\quad\quad A \leftarrow A \cup \{k_2\}$
23: $\quad\quad$ **end if**
24: $\quad$ **until** res="unsat"
25: **end while** ▷ End of Iteration
26: Possibly narrow.
27: Compute X_{k_i} for $k_i \notin K_R$
28: **return** all X_{k_i}

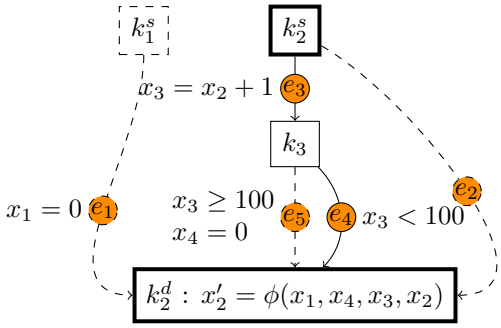


Figure 2.4 – Modified CFG and SMT-requests for the running example

with widenings (loopiter). X_{k_2} becomes $[0,1]$, then after widening $[0,\infty]$. A narrowing step gives finally $X_{k_2} = [0,99]$, which is thus the result of loopiter.

- *Step 3 : Is there a path from k_2 with $x \in [0,99]$ to k_2 with $x' \notin [0,99]$? No.*

The iteration thus ends with the desired invariant.

2.3.3 Validation

We validated this technique manually, the implementation inside ASPIC was unfortunately not mature enough to perform more experiments. However the method was implemented in the PAGAI tool [67] by Julien Henry, combined with other methods [56] and the impact of path focusing has been evaluated in the paper [65].

2.4 A combined numerical-Boolean abstraction for synchronous programs

This contribution has been published in the conference paper [GG11] and an extended version in the journal [FGG12].

In this work, we propose an enhancement of the compilation of synchronous programs with a combined numerical-Boolean abstraction. While our approach applies to synchronous dataflow languages in general, here, we consider the SIGNAL language for illustration. With this new abstraction, we determine the absence of reaction captured by empty clocks; mutual exclusion captured by two or more clocks whose associated signals never occur at the same time; or hierarchical control of component activations via clock inclusion. The abstraction we propose can be used early in the development process (validate simple properties while writing code), or be used to provide information to optimise or improve the quality of the generated code.

Overview of the approach Given the performance level reached by recent progress in *Satisfiability Modulo Theory* (SMT) (see Section 2.1.3) we use an SMT solver to reason on the new abstraction.
The advocated approach is depicted by Fig. 2.5. Given a synchronous dataflow program P, we define a corresponding abstraction, used to check the satisfiability of properties of interest, *i.e.*, those involving numerical expressions.
Once identified, all properties of interest are concretized into synchronous dataflow programs, which are later composed with the initial program P. The resulting composed program is equivalent to P in which properties involving numerical expressions have been made explicit in a form that is suitably addressable by a synchronous language compiler. Then, it becomes easier for the compiler to do an efficient analysis and code generation. Notice that an important advantage of this contribution is its *modular*, *i.e.*, non-intrusive, implementation regarding compilers. This clearly facilitates its integration to a given compiler and makes it easy to isolate a bug in the global framework (in comparison to a compiler-intrusive solution).

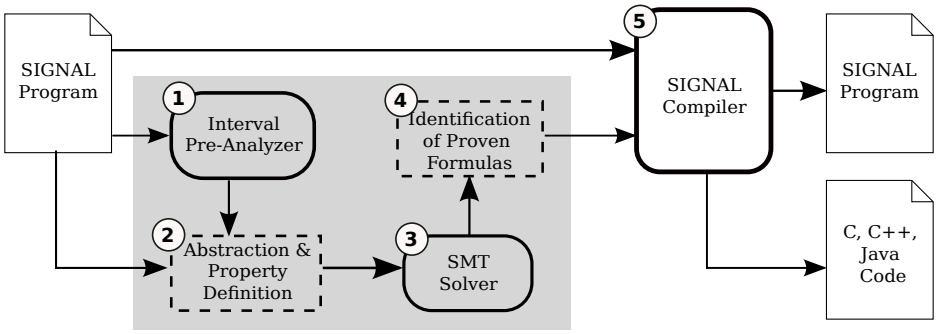


Figure 2.5 – An overview of our approach: steps and tools.

2.4.1 Preliminaries: the SIGNAL language

SIGNAL [54, 78] is a data-flow relational language that handles unbounded series of typed values $(x_t)_{t\in\mathbb{N}}$, called *signals*, implicitly indexed by discrete time, and denoted as `x`. For instance, a signal can be either of *Boolean* or *integer* or *real* types. At any logical instant $t \in \mathbb{N}$, a signal may be present, at which point it holds a value; or absent and denoted by $\perp$ in the semantic notation. There is a particular type of signal called `event`. A signal of this type always holds the value *true* when it is present. The set of instants at which a signal `x` is present is referred to as its *clock*, noted `^x`. A *process* is a system of equations over signals, specifying relations between values and clocks of the signals. A *program* is a process.

Primitive constructs SIGNAL relies on six primitive constructs: the *core language*. The syntax of the constructs is given below, with some informal explanations. Their formal semantics can be found in the paper.

- *Instantaneous relations:* `y:= R(x1,...,xn)` where `y, x1,...xn` are signals and `R` is a pointwise n-ary relation extended canonically to signals. This construct imposes `y, x1,...xn` *i)* to be simultaneously present, i.e. `^y = ^x1 = ... = ^xn` (i.e. *synchronous* signals), and *ii)* to hold values satisfying `y = R(x1,...,xn)` whenever they occur.

- *Delay:* `y:= x $ 1 init c` where `y, x` are signals and `c` is an initialisation constant. It imposes *i)* `x` and `y` to be synchronous, i.e. `^y = ^x`, while *ii)* `y` must hold the value carried by `x` on its previous occurrence.

- *Under-sampling:* `y:= x when b` where `y, x` are signals and `b` is of Boolean type. This construct imposes *(i)* `y` to be present only when `x` is present and `b` holds the value *true* while *(ii)* `y` holds the value of `x` at those logical instants. Let us denote by $[b]$ (resp. $[\neg b]$) denotes the set of instants where `b` is *true* (resp. *false*). Then *i)* rephrases itself into `^y = ^x` $\cap\, [b]$ (where $[b] \cup [\neg b] =$ `^b` and $[b] \cap [\neg b] = \emptyset$),

- *Deterministic merging:* `z:= x default y` where `x, y` and `z` are signals. This construct imposes *i)* `z` to be present when either `x` or `y` are present, i.e. `^z = ^x` $\cup$ `^y`, while *ii)* `z` holds the value of `x` if present, otherwise that of `y`.

- *Composition:* $\mathtt{P} \equiv \mathtt{P}_1 \,|\, \mathtt{P}_2$ where $\mathtt{P}_1$ and $\mathtt{P}_2$ are processes. It denotes the union of equations defined in processes, leading to the conjunction of the constraints associated with these processes. A signal variable cannot be assigned a value in both processes $\mathtt{P}_1$ and $\mathtt{P}_2$. SIGNAL adopts single assignment. A variable defined in $\mathtt{P}_1$ can be an input of $\mathtt{P}_2$, and vice versa. The composition operator is commutative and associative.

- *Restriction (or Hiding):* $\mathtt{P} \equiv \mathtt{P}_1$ `where x`, where $\mathtt{P}_1$ and `x` are respectively a process and a signal. It states that `x` is a local signal of process $\mathtt{P}_1$. Process `P` holds the same constraints as $\mathtt{P}_1$.

Syntactic sugar: clock operators, sub-processes. The core language of SIGNAL is expressive enough to derive new constructs of the language for programming comfort and structuring. In particular, SIGNAL allows one to explicitly manipulate clocks through some derived constructs that can be rewritten in terms of primitive ones. For instance, the clock extraction statement `y:= ^x`, meaning `y` is defined as the clock of `x`, is equivalent to `y:= (x = x)` in the core language.
For syntactical convenience, SIGNAL enables a modular definition of processes by providing a notion of *sub-process* (or local process). The statement $\mathtt{P}_1$ `where` $\mathtt{P}_2$, where $\mathtt{P}_1$ and $\mathtt{P}_2$ are processes, denotes the fact that the latter process is a sub-process of the former process.

Example 4 (Bathtub example)**.** *The simple* SIGNAL *process shown in Listing 4 specifies the status of a bathtub [16]. It has no input signal (line 02), but has three output signals (line 03).*

```
   process Bathtub =
2  (?
    ! integer level; boolean alarm, ghost_alarm; )
4  (|(| level := zlevel + faucet - pump
      | zlevel := level$1 init 1
6     | faucet := zfaucet + (1 when zlevel <= 4)
      | zfaucet := faucet$1 init 0
8     | pump := zpump + (1 when zlevel >= 7)
      | zpump := pump$1 init 0 |)
10  |(| overflow := level >= 9
      | scarce := 0 >= level
12    | alarm := scarce or overflow
      | ghost_alarm:= (true when scarce when overflow)
14                default false |)|)
   where
16   integer zlevel,zfaucet,zpump,faucet,pump;
     boolean overflow,scarce;
18 end;
```

The signal `level`, defined at line 04, reflects the water level in the bathtub at any instant. It is determined by considering two signals, `faucet` and `pump`, which are respectively used to increase and decrease the water level. These signals are increased by one under some specific conditions (lines 06 and 08), in order to maintain the water level in a suitable range of values.
An `alarm` signal is defined at line 12 whenever the water overflows (line 10) or becomes scarce (line 11) in the bathtub. An additional "ghost" alarm is defined at line 13/14, which is not expected to occur. Here, it is just introduced to illustrate one limitation of the static analysis of SIGNAL*. The clock of this signal is not completely specified in `Bathtub`. As stated before, this clock is the union of those associated with the two arguments of the `default` operator. The clock of the left argument is exactly known. The clock of the right-hand one is contextual because the argument is a constant (that is, a constant signal is always available whenever required by its context of usage): it is equal to the difference of `ghost_alarm`'s clock and first argument's clock. Since, this difference cannot be defined exactly from the program, further clock constraints on `ghost_alarm` will be required from the environment of `Bathtub` for an execution.*
Our abstraction will demonstrate that the `ghost_alarm` signal will never occur as well as both signals `pump` and `faucet`.

2.4.2 A new numerical-boolean abstraction for SIGNAL traces

We define an abstraction for SIGNAL program analysis. All considered programs are supposed to be in the syntax of the core language.
Our abstraction for program P is a logical formula Φ on the variables and clocks of P in a decidable theory (here, linear arithmetic of integers or reals) such that at any logical instant in an execution of

P, the current values of signals and clocks satisfy Φ. In other words, at any instant in an execution of P, its variables and clocks are a model of Φ.
Let `P` be a SIGNAL program. We denote by $X_P = \{x_1, x_2 \ldots x_n\}$ the set of all variables of `P`. Here, we consider scalar variables only. With each variable x_i (numerical, Boolean or `event`), we associate two abstract values: $\widehat{x_i}$ and $\widetilde{x_i}$ encoding respectively its clock and values. Intuitively, in such a couple, the first component encodes the clock of a signal, where *true* and *false* respectively mean presence and absence of instant in the clock. The second component encodes the value taken by a signal according to its presence.

The abstract semantics of the program, is a set of couples of the form $(\widehat{\ }, \widetilde{\ })$ where:

- function$\widehat{\ }$: $X_P \rightarrow \mathbb{B} = \{true, false\}$ assigns to a variable a Boolean value;
- function$\widetilde{\ }$: $X_P \rightarrow \mathbb{R} \cup \mathbb{B}$ assigns to a variable a numerical or Boolean value.

The abstract semantics (a set of couples $(\widehat{\ }, \widetilde{\ })$) will be represented as a first order logic formula Φ_P in which atoms are $\widetilde{x_i}$ and $\widehat{x_i}$, and the operators are usual logic operators and integer comparison functions. This abstraction into a first order logic formula naturally induces a loss of precision, as we will see later.
Intuitively, the abstract semantics encodes in the Φ formula all Boolean and numeric relations between variables and clocks that are valid "for all instants." This semantics will not be sufficient to prove any relation between variables at a given instant and their value an instant before, for instance.

Abstraction for expressions The abstraction of a given numerical SIGNAL expression $nexp$ (resp. a Boolean expression $bexp$) is a numerical expression (resp. a Boolean expression) that expresses its behaviour in terms of a SMT formula ϕ. The abstraction is straightforward (expressions are the same at every instant), more details can be found in [GG11].

Example 5. *Let* $b = (x + y = 4)$ `and` $(y < 10)$ *be a Boolean expression. Its abstraction is* $\phi(b) = \widetilde{x} + \widetilde{y} = 4 \wedge \widetilde{y} < 10$.

Abstraction of signal primitives The abstraction Φ_P of a given process `P` is the intersection of the abstractions of all its statements stm_i:

$$\Phi_P = \bigwedge_i^n \Phi(stm_i)$$

where each $\Phi(stmt)$ is a formula in quantifier-free linear integer arithmetic (QF_LIA) or quantifier-free linear real arithmetic (QF_LRA), defined according to the syntax of process definition.

- *Instantaneous relations:* `y:= R(x1,...,xn)`. The abstraction Φ of instantaneous relations is defined as follows:

  $$\wedge_{i=1}^n (\widehat{y} \Leftrightarrow \widehat{x_i}) \wedge \left(\widehat{y} \Rightarrow \widetilde{y} = \phi(exp)\right)$$

  where `R(x1,...xn)` is denoted by exp (for boolean expressions, = means $\Leftrightarrow$).

  These expressions express the equalities between clocks and values that are induced by SIGNAL semantics.

- *Delay:* `y:= x $ 1 init c`. The abstraction Φ of the delay construct is defined as follows:

  $$\widehat{y} \Leftrightarrow \widehat{x}$$

The abstraction here only expresses the equalities between clocks. A better abstraction could be performed if the user (or a pre-analysis) provides *invariants* for numerical variables. In that case, the global abstraction would be :

$$(\widehat{y} \Leftrightarrow \widehat{x}) \wedge \Big(\widehat{y} \Rightarrow \big(invar(\widetilde{x})[\widetilde{x}/\widetilde{y}] \vee (\widetilde{y} = c)\big)\Big)$$

where $invar(\widetilde{x})[\widetilde{x}/\widetilde{y}]$ denotes the substitution of $\widetilde{y}$ in a formula that expresses a constraint on x's values. Such an invariant can be a result of abstract interpretation techniques.

- *Under-sampling:* `y:= x when b`. The abstraction Φ of the under-sampling construct is defined as follows:

$$\Big(\widehat{y} \Leftrightarrow (\widehat{x} \wedge \widehat{b} \wedge \widetilde{b})\Big) \wedge (\widehat{y} \Rightarrow \widetilde{y} = \widetilde{x})$$

which expresses the fact that the signal y is present if and only if both signals b and x are present and b is *true*. The constraints on values are straightforward.

- *Deterministic merging:* `z:= x default y`. The abstraction Φ of the deterministic merging construct is defined as follows:

$$\begin{array}{l}\big(\widehat{y} \Leftrightarrow (\widehat{x} \vee \widehat{z})\big) \wedge \\ \Big(\widehat{y} \Rightarrow \big((\widehat{x} \wedge (\widetilde{y} = \widetilde{x})) \vee (\neg\widehat{x} \wedge (\widetilde{y} = \widetilde{z}))\big)\Big)\end{array}$$

The clock of variable y is the union of the clocks of x and z, and values are determined according to the presence of x.

- *Composition:* $\mathtt{P} \equiv \mathtt{P}_1 \,|\, \mathtt{P}_2$. The abstraction Φ of the composition operator is defined as follows:

$$\Phi \equiv \Phi_{P1} \wedge \Phi_{P2}$$

Example 4 (continuing from p. 18)**.** *By applying our abstraction to the* `Bathtub` *code, which is divided into P_1 (lines 04 to 09) and P_2 (lines 10 to 14) according to the process hierarchy, we obtain $\Phi_{Bathtub} = \Phi_{P_1} \wedge \Phi_{P_2}$, where Φ_{P_1} equals to:*

$$\begin{array}{l}\Phi_{P_1} = (\widehat{level} \Leftrightarrow \widehat{zlevel} \Leftrightarrow \widehat{faucet} \Leftrightarrow \widehat{pump} \Leftrightarrow \widehat{bzfaucet}) \wedge (\widetilde{level} = \widetilde{zlevel} + \widetilde{faucet} - \widetilde{pump}) \\ \wedge \big(\widehat{zfaucet} \Leftrightarrow (\widehat{zlevel} \wedge \widetilde{zlevel} \leq 4)\big) \wedge \big(\widehat{zfaucet} \Rightarrow \widetilde{faucet} = (\widetilde{zfaucet} + 1)\big) \wedge (\widehat{pump} \Leftrightarrow \widehat{zpump}) \\ \wedge \big(\widehat{zpump} \Leftrightarrow (\widehat{zlevel} \wedge \widetilde{zlevel} \geq 7)\big) \wedge \big(\widehat{zpump} \Rightarrow \widetilde{pump} = (\widetilde{zpump} + 1)\big)\end{array}$$

For Φ_{P_2}, we first rewrite equation at line 13/14 as follows:

```
(| y1 := true when scarce
 | y2 := y1 when overflow
 | ghost_alarm := y2 default false |)
```

Then, we obtain:

$$\begin{array}{l}\Phi_{P_2} = (\widehat{overflow} \Leftrightarrow \widehat{level} \Leftrightarrow \widehat{scarce}) \wedge \big(\widetilde{overflow} \Leftrightarrow (\widetilde{level} \geq 9)\big) \wedge (\widehat{scarce} \Leftrightarrow (\widetilde{level} \leq 0) \\ \wedge (\widehat{alarm} \Leftrightarrow \widehat{scarce} \Leftrightarrow \widehat{overflow}) \wedge \widehat{alarm} \Rightarrow \big(\widehat{alarm} \Leftrightarrow (\widehat{scarce} \vee \widehat{overflow})\big) \\ \wedge \big(\widehat{y_2} \Leftrightarrow (\widehat{scarce} \wedge \widehat{overflow} \wedge \widehat{scarce} \wedge \widehat{overflow})\big) \wedge (\widehat{y_2} \Rightarrow \widetilde{y_2}) \wedge (\widehat{ghost} \Leftrightarrow (\widehat{y_2} \vee \widehat{false})) \\ \wedge \Big(\widehat{ghost} \Rightarrow \Big(\big(\widehat{y_2} \wedge (\widehat{ghost} \Leftrightarrow \widetilde{y_2})\big) \vee \big(\neg\widehat{y_2} \wedge \neg\widehat{ghost}\big)\Big)\Big)\end{array}$$

Our abstraction is sound, in the sense that it preserves the behaviours of the abstracted programs: if a (safety) property is *true* on the abstraction, then it is also the case on the program. A proof of its soundness is given in [GG11].

2.4.3 Application to the analysis and compilation of SIGNAL programs

We have implemented the previous abstraction as a standalone tool called `Sync2Smt` (5k Ocaml LOC). Our toolchain is depicted in Figure 2.5.

Validation on common patterns We first validate our method on a few SIGNAL program patterns for which our abstraction helps in detecting some "clocks bugs" ("timing anomalies", in the literature). Such properties cannot be detected currently by the SIGNAL compiler because they involve numerical expressions, which are not addressed by a Boolean abstraction. Our abstraction allows their easy detection. These examples can be found in [FGG12].

Using the abstraction for clock analysis In [FGG12], we also demonstrate the relevance of our abstraction for analysing clock properties that combine both logical and numerical expressions. For instance, checking the mutual exclusion between multiple computation nodes whose activation conditions consist of such clocks, is useful to address sharing problems in a GALS system[10]. The GALS model is used to design multi-clock distributed system where each computation node holds its own clock providing a local (synchronous) vision of time. In addition, establishing that some nodes or events in a system never occur, via empty clocks, can serve to guarantee that undesired behaviours never happen, or conversely to detect that some expected behaviours are never observed. This kind of observation can sightly improve the quality of the C code generated by the SIGNAL compiler, compared to the quite simplistic actual compilation scheme. On the one hand, dead code elimination is made possible thanks to information resulting from the analysis of our abstraction. As an example, we explain the impact of our analysis on the compilation of the `Bathtub` example.

Example 4 (continuing from p. 18). *Given the formula $\Phi_{Bathtub}$ obtained previously, as the abstraction of the bathtub* SIGNAL *specification, the main properties of interest are the following*[11]*:*

1. *`pump` and `faucet` have disjoint clocks:* $\neg(\widehat{faucet} \wedge \widehat{pump})$,
2. *The water cannot overflow and be scarce at the same time:* $\neg\big(\widehat{scarce} \wedge \widehat{overflow} \wedge \widehat{scarce} \wedge \widehat{overflow}\big)$,
3. *`alarm` and `level` have the same clock:* $\widehat{alarm} \Leftrightarrow \widehat{level}$.

The conjunction of these three properties will be sufficient to prove that the `ghost_alarm` signal never occurs (equivalently, has an empty clock), and also that all other signals have empty clocks.
These properties are easily verified on the abstraction of `Bathtub` process. As a result, their corresponding concretisations can be safely composed with `Bathtub` without changing its semantics. Possible concretisations of the above properties in SIGNAL *are as follows:*

1. `faucet ^* pump ^= ^0` *(`^=` denotes the equality on clocks, `^*` the intersection)*
2. `true when scarce when overflow ^= ^0`
3. `alarm ^= level`

By composing these statements with `Bathtub`*, one obtains the semantically equivalent process, named* `Bathtub_Bis`*, shown in the following:*

```
process Bathtub_Bis =
(?
 ! integer level; boolean alarm, ghost_alarm; )
(|(| level := zlevel + faucet - pump
    ...
   | ghost_alarm:=(true when scarce when overflow)
                  default false |)
 |(| true when scarce when overflow ^= ^0
   | faucet ^* pump ^= ^0
   | alarm ^= level |) |)
 where
```

[10] Globally Asynchronous Locally Synchronous
[11] These properties cannot be verified by the current version of the SIGNAL compiler.

```
  integer zlevel,zfaucet,zpump,faucet,pump;
  boolean overflow,scarce;
end;
```

The result of its analysis performed by the compiler is now as follows:

```
(| CLK_ghost_alarm := ^ghost_alarm
 | CLK_ghost_alarm ^= ghost_alarm
 | (| ghost_alarm := not CLK_ghost_alarm |)
 |);%^0 ^= level ^= alarm
        ^= zlevel ^= zfaucet ^= zpump
      ***WARNING: null clock signals%
```

The whole set of constraints inferred by the compiler is now restricted to the fact that the `ghost_alarm` *signal is always equal to* `false`*. The compiler has also detected that the clocks of the other signals are all empty (lines* `04/04b`*). Finally, the corresponding generated code is provided below, where the dead code is avoided.*

```
01:   { ghost_alarm = FALSE;
02:    /* produce output value
03:         for the signal ghost_alarm */ } ...
```

2.5 Revisiting interval analysis for scalability

In this section we present our contribution on improving both precision and scalability of range analyses. This work is the first analysis of the paper [SMO+14], whose goal was to design a complete toolchain for securing memory accesses in the C programming language, and will be explained in detail in Chapter 4.

Range analysis, as originally defined by Cousot and Cousot [37], associates variables with integer intervals. This approach enables several compiler optimisations, but it is not effective to validate memory accesses, as demonstrated by Logozzo and Fähndrich [79].
The program in Figure 2.6 illustrates this deficiency: a traditional range analysis will not find, in this program, constants onto which to rely upon, to prove that variables `i` and `j` only access valid positions of array `p`.

Example 6 (Motivating example)**.** *Figure 2.6 shows examples of three types of analyses, including the Symbolic Range Analysis that we describe in Section 2.5.1. The information associated with a variable depends on which part of the program we are; hence, the figure shows the results of each analysis at three different regions of the code. In this example, classic range analysis can only infer positiveness of variables. Pentagons can infer also that* j *is always strictly less than* N *inside the loop. Octagons are more precise since they are able to find that* $m = i$ *at control points* b *and* c*, for instance.*

We could also use state-of-the-art relational analyses from the abstract interpretation community like octagons, but they do not scale enough for our target which is compilation.
The objective of this work is thus:

- to improve the precision of range analysis by considering symbolic ranges, originally defined by Blume and Eigenmann [20];
- while still remaining scalable, that's why we design a *sparse analysis*.

2.5.1 Sparse Symbolic Range Analysis

To scale an analysis, the Static Single Information (Static Single Information (SSI)) framework [5] propose to focus on the preservation of the following invariants:

- We attach one abstract value per variable (and not to a set of variables).

```
unsigned N = read();
int* p = alloc(N);
int i = 0;
int m = 0;
int j = N - 1;
while (i < j) {
  p[i] = -1;
  p[j] = 1;
  i++;
  j--;
  m++;
}
p[m] = 0;
```

	Range analysis	Pentagons	Symbolic range analysis	Octagons
(a)	$\mathcal{F}(\mathtt{N}) = [0, +\infty]$ $\mathcal{F}(\mathtt{i}) = [0, 0]$ $\mathcal{F}(\mathtt{j}) = [-1, +\infty]$ $\mathcal{F}(\mathtt{m}) = [0, 0]$	$\mathcal{F}(\mathtt{N}) = \{\}, [0, +\infty]$ $\mathcal{F}(\mathtt{i}) = \{\}, [0, 0]$ $\mathcal{F}(\mathtt{j}) = \{\mathtt{N}\}, [-1, +\infty]$ $\mathcal{F}(\mathtt{m}) = \{\}, [0, 0]$	$\mathcal{F}(\mathtt{N}) = [\mathtt{N}, \mathtt{N}]$ $\mathcal{F}(\mathtt{i}) = [0, 0]$ $\mathcal{F}(\mathtt{j}) = [\mathtt{N}-1, \mathtt{N}-1]$ $\mathcal{F}(\mathtt{m}) = [0, 0]$	$\mathcal{F}(\mathtt{i}, \mathtt{i}, +) = \mathtt{i} \geq 0$ $\mathcal{F}(\mathtt{i}, \mathtt{i}, -) = -\mathtt{i} \geq 0$ $\mathcal{F}(\mathtt{N}, \mathtt{j}, +) = \mathtt{N} - \mathtt{j} \geq 1$ $\mathcal{F}(\mathtt{N}, \mathtt{j}, -) = \mathtt{j} - \mathtt{N} \geq -1$ $\mathcal{F}(\mathtt{m}, \mathtt{i}, +) = \mathtt{i} - \mathtt{m} \geq 0$ $\mathcal{F}(\mathtt{m}, \mathtt{i}, -) = \mathtt{m} - \mathtt{i} \geq 0$
(b)	$\mathcal{F}(\mathtt{i}) = [0, +\infty]$ $\mathcal{F}(\mathtt{j}) = [1, +\infty]$ $\mathcal{F}(\mathtt{m}) = [0, +\infty]$	$\mathcal{F}(\mathtt{i}) = \{\mathtt{j}, \mathtt{N}\}, [0, +\infty]$ $\mathcal{F}(\mathtt{j}) = \{\mathtt{N}\}, [1, +\infty]$ $\mathcal{F}(\mathtt{m}) = \{\}, [0, +\infty]$	$\mathcal{F}(\mathtt{i}) = [0, \mathtt{N}-2]$ $\mathcal{F}(\mathtt{j}) = [1, \mathtt{N}-1]$ $\mathcal{F}(\mathtt{m}) = [0, +\infty]$	$\mathcal{F}(\mathtt{i}, \mathtt{i}, +) = \mathtt{i} \geq 0$ $\mathcal{F}(\mathtt{j}, \mathtt{i}, -) = \mathtt{j} - \mathtt{i} \geq 0$ $\mathcal{F}(\mathtt{j}, \mathtt{i}, +) = \mathtt{j} + \mathtt{i} \geq 1$
(c)	$\mathcal{F}(\mathtt{i}) = [0, +\infty]$ $\mathcal{F}(\mathtt{j}) = [-1, +\infty]$ $\mathcal{F}(\mathtt{m}) = [0, +\infty]$	$\mathcal{F}(\mathtt{i}) = \{\mathtt{N}\}, [0, +\infty]$ $\mathcal{F}(\mathtt{j}) = \{\mathtt{N}\}, [-1, +\infty]$ $\mathcal{F}(\mathtt{m}) = \{\}, [0, +\infty]$	$\mathcal{F}(\mathtt{i}) = [0, \max(0, \mathtt{N}-1)]$ $\mathcal{F}(\mathtt{j}) = [-1, \max(0, \mathtt{N}-2)]$ $\mathcal{F}(\mathtt{m}) = [0, +\infty]$	$\mathcal{F}(\mathtt{i}, \mathtt{i}, +) = \mathtt{i} \geq 0$ $\mathcal{F}(\mathtt{m}, \mathtt{i}, -) = \mathtt{m} - \mathtt{i} \geq 0$ $\mathcal{F}(\mathtt{m}, \mathtt{i}, +) = \mathtt{m} + \mathtt{i} \geq 0$

Figure 2.6 – A comparison between four different types of static analyses. Only a few relations are shown for Octagons.

- The abstract state of any variable must be invariant in every program point where this variable is alive. A variable v is alive at a label ℓ if there is a path in the program's control flow graph from ℓ to another label ℓ' where (i) v is used and (ii) v is not redefined along this path.

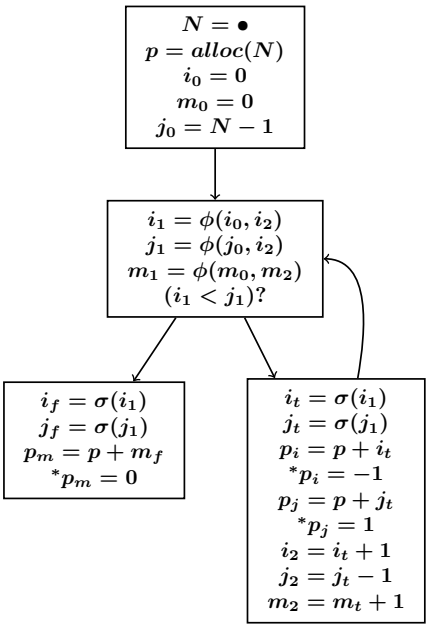
The strategy for the second point is to split the live ranges of variables: splitting a given variable v, at program label ℓ, by inserting a copy $v' = v$ at ℓ and renaming every use of v to v' in points dominated[12] by ℓ. According to Tavares *et al.* [104], it is enough to split live ranges at places (control points) where information originates: for instance, for a numeric range analysis, a new version of x should be "invented" each time a new value is assigned to x (which is what is done in the classical SSA form), but also after each test, which we will see in the sequel in the context of Symbolic Range Analysis.

Splitting Required by Symbolic Range Analysis. For the range analysis we want to perform, it is sufficient to propagate the information from the definition of variables and from conditional tests that use these variables. Thus, to make this analysis sparse, we must split live ranges at these places.

Splitting at definitions creates the Static Single Assignment representation. Splitting at conditionals creates the representation that Bodik *et al.* have called the *Extended Static Single Assignment form (Extended Static Single Assignment Form (eSSA))* [21]. In the sequel, we consider programs into eSSA form [13]: for each conditional, we name the variables created at the "true" side of the branch a_t and b_t and the variables created at the "false" side of it a_f and b_f. As a convenience, we shall mark these copies with a σ, indicating that they have been introduced due to live range splitting at conditionals. We emphasise that these σ's are just a notation to help the reader to understand our way to split live ranges and have no semantics other than being ordinary copies[14]. We borrow this notation from Ananian's work [5].

Example 6 (continuing from p. 22)**.** *Figure 2.7b shows the program in Figure 2.6 after live range splitting.*

[12] A node ℓ dominates a node n if every path from the entry node to n must go through ℓ.
[13] We design a better splitting strategy in the paper, that we do not recall here for the sake of brevity.
[14] Bodik *et al.* [21] would name similar instructions π-functions



(a) The program in Figure 2.6 converted into extended static single assignment form

	0th	1th	2nd	$3^{rd}(+\nabla)$
N	$\emptyset$	$[0,N]$	$[0,N]$	$[0,N]$
i_0	$\emptyset$	$[0,0]$	$[0,0]$	$[0,0]$
m_0	$\emptyset$	$[0,0]$	$[0,0]$	$[0,0]$
j_0	$\emptyset$	$[-1,N-1]$	$[-1,N-1]$	$[-1,N-1]$
i_1	$\emptyset$	$[0,0]$	$[0,max]$	$[0,max]$
j_1	$\emptyset$	$[-1,N-1]$	$[-1,N-1]$	$[-1,N-1]$
m_1	$\emptyset$	$[0,0]$	$[0,1]$	$[0,+\infty]$
i_t	$\emptyset$	$[0,N-2]$	$[0,N-2]$	$[0,N-2]$
j_t	$\emptyset$	$[1,N-1]$	$[1,N-1]$	$[1,N-1]$
i_2	$\emptyset$	$[1,N-1]$	$[1,N-1]$	$[1,N-1]$
j_2	$\emptyset$	$[0,N-2]$	$[0,N-2]$	$[0,N-2]$
m_2	$\emptyset$	$[1,1]$	$[1,2]$	$[1,+\infty]$
i_f	$\emptyset$	$-$	$-$	$[-1,max]$
j_f	$\emptyset$	$-$	$-$	$[-1,max]$

(b) Symbolic range analysis: Steps

Figure 2.7 – Running example: steps of the analysis within the SymBoxes domain, $max = max(0, N-1)$.

Symbolic Range Analysis on E-SSA control flow From the tailored intermediate representation that is the E-SSA form, we now have to construct our symbolic abstract domain SymBoxes that will associate to each variable x of the program a symbolic range $R(x)$.
A *symbolic interval* is a pair $R = [l, u]$, where l and u are symbolic expressions (ie expressions on variables that are either constants known at compile time, or input values: random values, user-provided values, function arguments...).
We denote by $(R_\downarrow, R_\uparrow)$ the lower (l) and upper-bound (u) of the interval R.
We define the partially ordered set of (symbolic) intervals $S^2 = (S \times S, \sqsubseteq)$, where the ordering operator is defined as: $[l_0, u_0] \sqsubseteq [l_1, u_1]$, if $l_1 \leq l_0 \wedge u_1 \geq u_0$

We then define the semi-lattice SymBoxes of symbolic intervals as $(S^2, \sqsubseteq, \sqcup, \emptyset, [-\infty, +\infty])$, where the join operator "$\sqcup$" is defined as: $[a_1, a_2] \sqcup [b_1, b_2] = [\min(a_1, b_1), \max(a_2, b_2)]$. Our lattice has a least element $\emptyset$ and a greatest element $[-\infty, +\infty]$.
Clearly, this lattice is infinite; therefore, in order to end up the computation of the set of constraints we use a widening operator defined by (under the assumption $R_1 \sqsubseteq R_2$):

$$R_1 \nabla R_2 = [l, u], \text{ where } \begin{cases} l = R_{1\downarrow} & \text{if } R_{1\downarrow} = R_{2\downarrow} \\ l = -\infty & \text{otherwise} \\ u = R_{1\uparrow} & \text{if } R_{1\uparrow} = R_{2\uparrow} \\ u = +\infty & \text{otherwise} \end{cases}$$

This is the extension of the classic widening on intervals to symbolic intervals: a lower (resp. upper) bound of a given symbolic interval can only be stable or diverge towards $-\infty$ (resp. $+\infty$), thus our widening operator will ensure the convergence of our analysis.

Abstract Interpretation with constraints To apply the abstract interpretation framework, we also have to give an interpretation of the operations of the program. This is done in Figure 2.8.

- assignments after reads give only symbolic information.
- assignments to expressions makes use of "abstract transformers."

- ϕ functions are "join nodes", thus we perform a (symbolic) interval union.
- On intersection nodes (tests), we perform a (symbolic) interval intersection.

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

Figure 2.8 – Constraints for the symbolic range analysis.

The abstract transformers denoted by $\oplus^I$ in Figure 2.8 are for instance:

- $[l_0, u_0] +^I [l_1, u_1] = [l_0 + l_1, u_0 + u_1]$, $[l, u] +^I \emptyset = [l, u]$.
- $[l_0, u_0] \times^I [l_1, u_1] = [\min(T), \max(T)]$, where $T = \{l_0 \times l_1, l_0 \times u_1, u_0 \times l_1, u_0 \times u_1\}$, $[l, u] \times^I \emptyset = [l, u]$.

Solving From a given program we thus generate a system of constraints that we solve using a Kleene iteration. The analysis is sparse since we do not need the control points any more. The information is only attached to variables. Widening is applied on a ϕ node only after 3 iterations of symbolic evaluation ("delayed widening"). This decision is arbitrary, and a larger number of iterations might increase the precision of our results. However, in our experiments we only observed very marginal gains when using more iterations.

Example 6 (continuing from p. 22)**.** *Figure 2.7b shows the steps that our analysis performs on the program of Figure 2.7a. The order in which we evaluate constraints is given by the reverse post-ordering of the program's control flow graph. This ordering tends to reduce the number of iterations of our fixpoint algorithm [86, p.421]. However, any order of evaluation would led to the same result. In this example, we let "max" be* $max(0, N-1)$.

On the evaluation of symbolic expressions. As shown in the preceding paragraphs, we have to evaluate symbolic expressions (simplification of expressions like $max(e_1, e_2)$, equality tests, ...). We rely on GiNaC [13], a library for symbolic manipulation, to perform these operations[15]. For the equality test, if GiNaC is not able to prove a given equality between symbolic expressions, we conservatively assume that the two expressions are not comparable. As a consequence, we may widen an expression to $+\infty$ even if it was stable.

Proposition 1. *The former analysis always returns an over-approximation of the actual ranges of the variables of the program (no matter the valuations of the symbols would be).*

2.5.2 Validation

Runtime We run our experiments in a twelve-core Intel(R) Xeon(R) CPU E5-2620 at 2.00GHz, with 15,360KB of cache, and 16GB of RAM. Neither our compiler, nor our benchmarks, run in parallel. The run time of our analysis compares favourably to the run time of equivalent algorithms, as Figure 2.9 shows.

[15] GiNaC is available at `http://www.ginac.de/`

The figure compares the runtime of our symbolic range analysis, Rodrigue *et al.*'s range analysis, and Pentagons, for the 60 largest benchmarks in the LLVM test suite and SPEC CPU 2006. These programs gave us over 5.18 million bytecodes to analyse. We tried to our best to be as faithful to the original description of Pentagons as possible.

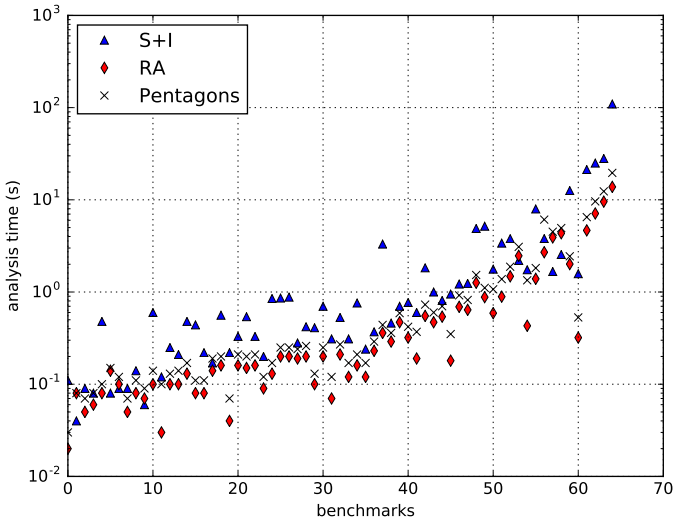


Figure 2.9 – Runtime comparison of three non-relational analysis. Timings are in seconds. **S**: symbolic range analysis. **I**: integer overflow elimination. **RA**: Rodrigues *et al.*'s range analysis [93]. **Pentagons** [79]. Y axis is runtime, in seconds. Each X point is a benchmark, sorted by size (from 4k to 1M instructions).

Precision The precision of our analysis is evaluated through its use in a bigger framework called *GreenArrays*, that we will study later (Section 4.2).

2.6 Conclusion

In this chapter I summarised the contributions made in the domain of numerical programs analysis. This chapter is an illustration of the impact of combining techniques from different communities, from SMT-solving to abstract interpretation or compilation, or from abstract interpretation to compilation. One lesson of this work is that the effort of bringing a technique to another community is non trivial, however, it deserves to be made.

The last contribution advocates for the design of more *scalable* abstract domains. We strongly believe that such abstract domains must be thought in term of scalability and expressivity at the same time. The SSI-framework proposes guidelines to design such abstract domains, however, there is still an open research question for fully relational analyses: can they scale? are there ways to degrade their expressivity in a generic way so as to improve their precision?

3 Static Termination Analysis

Proving termination is one key application of invariant generation and static analyses techniques. This problem, which was studied since the early days of Computer Science, and its companion problem, which is the computation of the WCCC (worst-case computational complexity), has many applications:

- for proving full correctness: since the seminal paper of Floyd [53], termination and *partial correctness* are complementary. However, much more attention has been given to partial correctness: we affix assertions to each program point and prove that they are consequences of the assertions of its predecessors in the program control graph. The assertions at the entry point of the program are its *preconditions*, the assertions at loop entry points are *invariants*, while the assertions at its exit point must entail correctness, according to some set of requirements [1]). However, this method proves only partial correctness, i.e., that the program gives the correct result if and when it terminates. Termination thus still has to be proved to fully prove the correctness of programs, and despite the early definition of the problem, has clearly received less attention from the program analysis community.

- for proving safety of reactive programs: in reactive programs, we have to prove functional correctness as well as *reactivity*, i.e. that the code inside the infinite reactive loop executes itself in finite bounded time. Moreover, in order to parametrise the sensors and actuators of a given platform, we have to compute a *safe bound* on the sampling rate. An accurate worst-case execution time can be derived from WCCC results, with the help of abstract static cache analyses such as [7]. Static analyses can improve the precision of the global worst-case execution time computation, as shown for instance in [66].

- for detecting *hotspots* in intensive computations and thus perform best-effort optimisation in these particular pieces of code. This is a particular case of *prediction*. In static compilers, loops with a high number of iterations may benefit from aggressive code motion strategies, or polyhedral-based optimisation [50] whose cost might be too high to be applied at any point of the program.

Summary of the chapter The main idea of this work was initially to explore the link between scheduling and termination [44, 45], through making a parallel between schedules and ranking functions. Using a classical algorithm of the program scheduling literature, combined with numerical invariants computed by abstract interpretation, we first demonstrate the expressivity of the method in [ADFG10]. This work was further extended to deal with larger programs [AAG12] by classical static analyses techniques adapted to the particular case of termination.

[1] The Frama-C tool, developed at the CEA, is capable of checking such *Hoare triples* `http://frama-c.com/`

More recently, in [GMR15] we showed how to drastically decrease the size of the constraint systems we solve while computing ranking functions using ideas from the static analysis community (namely, counter-example guided refinement).
Another experience in termination was the study [RAPG14] of the Mozilla Firefox Just-In-Time compiler in which detecting "intensive" loops is crucial for performance.
In all these works, from theory to practice, the results we obtain show the relationships and the cross fertilisation of compilation techniques and static analyses techniques.

Outline Section 3.1 defines the main theoretical tools used for proving termination of sequential programs. Section 3.2 exposes the two main algorithmic contributions published in [ADFG10] and [GMR15] as well as their experimental evaluation. Section 3.3 describes our techniques to improve the scalability of the tools. Finally Section 3.4 summarises our case study on the Mozilla JIT compiler.

3.1 Model of programs, termination, ranking functions

In this chapter, we also consider programs as Integer Interpreted automata, as described in Section 2.1.1. We also assume that they have been obtained through a safe abstraction procedure. The problem is now to prove termination of a given automaton, i.e. to prove that there is no infinite path from the initial states.
The standard technique for proving termination is to consider ranking functions to well-founded sets. A well-founded set $\mathscr{W}$ is a set with a (total or partial) order $\preceq$ (we write $a \prec b$ if $a \preceq b$ and $a \neq b$) such that there is no infinite descending chain, i.e., no infinite sequence $(x_i)_{i\in\mathbb{N}}$ with $x_i \in \mathscr{W}$ and $x_{i+1} \prec x_i$ for all $i \in \mathbb{N}$.

Definition 3. *A* ranking *function (ρ) is a function $\rho : \mathscr{K} \times \mathbb{Z}^n \to \mathscr{W}$, from the automaton states to a well-founded set $(\mathscr{W}, \preceq)$, whose values decrease at each transition $t = (k, g, a, k')$:*

$$\boldsymbol{x} \in \mathscr{R}_k \wedge g(\boldsymbol{x}) = \mathit{true} \wedge \boldsymbol{x}' = a(\boldsymbol{x}) \Rightarrow \rho(k', \boldsymbol{x}') \prec \rho(k, \boldsymbol{x}) \tag{3.1}$$

where $\mathscr{R}_k$ are the reachable valuations at control point k (any over-approximation $\mathscr{I}_k$ can replace i). It is said affine *if it is affine in the second parameter (the variables).*

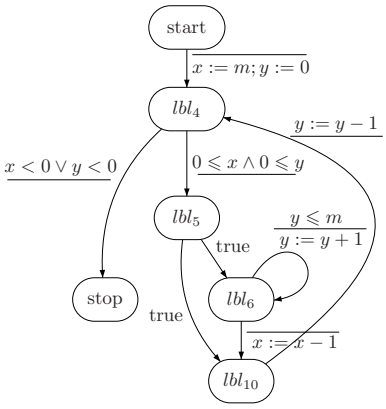
Definition 4. *A ranking function is* one-dimensional *if its co-domain is $(\mathbb{N}, \leq)$. It is k-*dimensional *(or multi-dimensional of dimension k) if its co-domain is $(\mathbb{N}^k, \preceq_k)$, where the order $\preceq_k$ is the standard lexicographic order on integer vectors.*

Obviously, the existence of a ranking function implies program termination for any valuation $\boldsymbol{v}$ at the initial control point k_{init}. A well-known property is that an integer interpreted automaton terminates for any initial valuation if and only if it has a ranking function (not necessarily expressible in our $(\mathbb{N}^k, \preceq_k)$ setting). Furthermore, if it terminates and has bounded non-determinism [2], there is a one-dimensional ranking function, which is not necessarily affine.

Example 7 (Motivating example)**.** *An example program is given in Fig. 3.1, with its corresponding automaton. The control points are labelled for convenience, and transitions are depicted with arrows indexed by $\frac{g}{a}$ (g is omitted when $g = true$). State names are assigned arbitrarily by our parser.*
The C code features two nested loops, which do not fit into the structured programming model, since the inner counter, `y`*, is modified in the outer loop. The* `indet` *function abstracts non-determinism or an intractable test.*
The outcome of non-determinism is that, in the corresponding automaton, both transitions out of state lbl_5 have a true guard. The right of Fig. 3.1 successively gives, assuming $m > 0$, the invariants as found

[2] Each non deterministic choice lies in a finite set.

```
y = 0;
x = m;
while(x>=0 && y>=0){
  if(indet()){
    while(y <= m && indet())
      y++;
    x--;
  }
  y--;
}
```



c. point	Invariant
lbl_4	$m \geq x > 0, m \geq y > 0$
lbl_5	$m \geq x \geq 0, m \geq y \geq 0$
lbl_6	$m \geq x \geq 0, m+1 \geq y \geq 0$
lbl_{10}	$\begin{cases} m \geq x \geq -1, m+1 \geq y \geq 0 \\ 2m \geq x+y \end{cases}$

c. point	Ranking Function
$start$	$2m+4$
lbl_4	$(2x+3, 3y+3)$
lbl_5	$(2x+3, 3y+2)$
lbl_6	$(2x+2, m-y+1)$
lbl_{10}	$(2x+3, 3y+1)$
	WCCC **(whole program)**
	$5+7m+4m^2$

Figure 3.1 – Illustrating example of [ADFG10]. True guards and ε actions are ommited.

by ASPIC *(an abstract-interpretation based invariant generator, see Section 2.2.1), followed by the bi-dimensional rankings and the corresponding WCCC computed by* RANK*, our tool that implements the algorithms of [ADFG10]. These rankings expressions are positive and lexicographically decrease along each transition. For instance, the first component of the ranking function decreases from* $2x+3$ *at* lbl_5 *to* $2x+2$ *at* lbl_6*, then* $2x+3$ *at* lbl_{10}*, but since* x *is changed to* $x-1$ *by the corresponding transition, the ranking has really decreased.*

3.2 Computing ranking functions of imperative programs

Unlike previous work [26, 59] where the construction of invariants is coupled with the termination proof or evaluation of iteration bounds, the invariants $\mathscr{I}_k$ on all control points of the programs are pre-computed (The non-precision of the invariant will be the unique source of loss of precision of our method, as we will see later). In both papers [ADFG10] and [GMR15], we use ASPIC [3] to compute these invariants.

3.2.1 Key observation: ranking functions are schedules

The polyhedral model is a collection of techniques developed around a common intermediate representation of programs: integer polyhedra. Such a mathematical representation of programs inherits nice properties from the underlying mathematical structure. For instance, when loop transformations are represented as affine functions, compositions of transformations are also affine functions due to closure. The polyhedral representation was linked to loop programs by an analysis proposed by Feautrier [48] that provides exact dependence analysis [4] information where statement instances (i.e., statements executed at different loop iterations) and array elements are distinguished. The exact dependence information through this analysis and the use of linear programming techniques to explore the space of legal schedules [49] is what constitutes the base of the polyhedral model for loop transformations. The key idea of all these techniques is that any loop transformation is valid if *dependencies are not broken* (in other words, if statement A depends on statement B, then B will be always scheduled before A).

In the polyhedral framework, a schedule can be described as a function that associates (logical) dates with operations that is strictly increasing from some starting date (say, 0) and fulfilling the dependencies, as described in Example 8.

[3] http://laure.gonnord.org/pro/aspic/aspic.html

[4] In the context of polyhedral optimisation, a statement A depends on another statement B if computes a result which will be later used by A.

Example 8 (Illustration for the polyhedral model framework). *Figure 3.2 is a loop kernel computing the Smith-Waterman optimal sequence alignment algorithm*[5]*. The polyhedral model provide us tools to automatically compute the dependencies that are depicted on the right: each point represents an execution of the block S computing* `H[i][j]` *for a given i and j in* $[\![0, N]\!]$*. Such an operation is written* $\langle S, i, j\rangle$*. The arrows represent the dependencies towards a given* $\langle S, i, j\rangle$*. For instance the diagonal arrow means that* `H[i][j]` *requires the value of* `H[i-1][j-1]` *to be computed.*

```
for(i=0; i<=N; i++)
  for (j=0; j<=N; j++)
    //Block S
    {
      m1[i][j] = Integer.MIN_VALUE;
      for(k=1; k<=i; k++)
        m1[i][j] = max(m1[i][j],H[i-k][j] + W[k]);

      m2[i][j] = Integer.MIN_VALUE;
      for(k=1; k<=j; k++)
        m2[i][j] = max(m2[i][j],H[i][j-k] + W[k]);

      H[i][j] = max(0,H(i-1,j-1)+s(a[i],b[i]),
                    m1[i][j],m2[i][j]);
    }
```

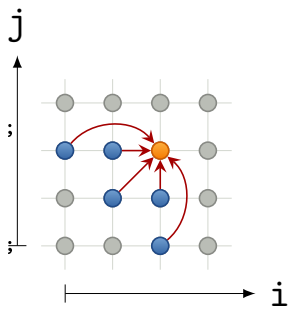


Figure 3.2 – Polyhedral scheduling: an example. The Smith-Waterman sequence alignment algorithm and its dependencies. Each point (i, j) represents an execution of the block S, denoted by $\langle S, i, j\rangle$.

From these dependencies, we are able to describe valid schedules, *for instance* $S(i, j) \rightarrow (i, j)$ *or* $S(i, j) = i + j$ *are valid since they respect dependencies. Clearly these two schedules grow from 0 and are bounded.*

Similarly, in a numerical automaton, a ranking function associates a non-negative expression to each computation of the program that strictly decreases each time the program takes a transition (e.g., change in loop iterators). The main idea was to use the previous work on multidimensional polyhedral loop scheduling [49], but in the larger scope of general control flow graphs (instead of static `for` loops and array optimisation) and programs operating on numerical variables.

From monodimensional ranking function to multidimensional Another observation is that a multidimensional (affine) ranking function can be computed iteratively using an algorithm that generates a monodimensional (affine) function *on a maximal set of transitions.*
This algorithm relies on a notion of "maximal termination power" property that we enforce for the ranking functions synthesised in both papers. In the same way, we prove that despite this *greedy* approach, our technique is *weakly complete* (which means that if our invariants are precise enough to prove that there exist a multidimensional affine ranking function, then we find it). Furthermore the multidimensional affine ranking functions that our algorithms compute have the least possible number of dimensions.

Synthesising 1D-ranking functions To find a suitable function ρ at Line 3 for a subset T of transitions ($\mathcal{T}$), we use linear programming. The set of inequalities that we need to solve are the following:

- The ρ function must be non-negative on T

[5]See https://en.wikipedia.org/wiki/Smith-Waterman_algorithm. We consider two sequences of the same length N.

Algorithm 3 Multidimensional Termination

Input: An automaton with $\mathscr{T}$ its set of transitions, $\mathscr{I}_k$ invariants
Output: Yes+ρ/Abort

1: $i = 0$; $T = \mathscr{T}$; ▷ Initialise T to the set of all transitions
2: **while** T is not empty **do**
3: Find a 1D affine function ρ which is decreasing on all $t \in T$ and strictly decreasing on a maximal subset $T' \subseteq T$.
4: Let $\rho_i = \rho$; $i = i + 1$; ▷ ρ_i defines the i-th component of ρ
5: If $T' = \emptyset$ **return** false ▷ No multi-dimensional affine ranking.
6: $T \leftarrow T'$ ▷ The transitions have level i
7: **end while**;
8: $d = i$; **return** true; ▷ There is a d-dimensional ranking

- The ρ function should decrease on all transitions of T, and strictly decrease on a maximal subset of T.

In the sequel, I will denote by:

- $\rho = \boldsymbol{x} \mapsto \boldsymbol{\lambda} \cdot \boldsymbol{x} + \lambda_0$ the (monodimensional) ranking function to be discovered, where λs are unknown.
- $\mathscr{I}$ the invariant on the (unique) control point we are considering with m affine constraints $\mathscr{I} = \{\boldsymbol{x} \mid \bigwedge_{i=1}^{m} \boldsymbol{a}_i.\boldsymbol{x} - b_i \geq 0\}$.
- $\tau = \cup_{t \in \mathscr{T}} \tau_t$ the transition relation for the whole program. For a given transition $t \in \mathscr{T}$ we denote by $\mathscr{Q}_t$ the polyhedron that encodes the input/output transition relation on $(\boldsymbol{x}, \boldsymbol{x}')$ induced by t:$\mathscr{Q}_t = \{\boldsymbol{y} = (\boldsymbol{x}, \boldsymbol{x}') \mid Q_t \boldsymbol{y} + \boldsymbol{q_t} \geq \boldsymbol{0}\}$.

The constraint we have on ρ are thus expressed by:

$$\forall x \in \mathscr{I}, \rho(x) \geq 0 \tag{3.2}$$

$$\forall (x, x') \in \tau, x \in \mathscr{I}, \rho(x) - \rho(x') \geq 0 \tag{3.3}$$

The way we solve them will be different in the following next sections. We will illustrate them on the running example.
The two methods rely on the affine form of Farkas' lemma [96]:

Lemma 1 (Farkas' lemma, affine form)**.** *An affine form $\phi : \mathbb{R}^n \to \mathbb{R}$ with $\phi(\boldsymbol{x}) = \boldsymbol{c}.\boldsymbol{x} + c_0$ is non-negative everywhere in a non-empty polyhedron $\{\boldsymbol{x} \mid A\boldsymbol{x} + \boldsymbol{a} \geq \boldsymbol{0}\}$ iff:*

$$\exists \boldsymbol{\lambda} \in (\mathbb{R}^+)^n, \lambda_0 \in \mathbb{R}^+ \text{ such that } \phi(\boldsymbol{x}) \equiv \boldsymbol{\lambda}.(A\boldsymbol{x} + \boldsymbol{a}) + \lambda_0$$

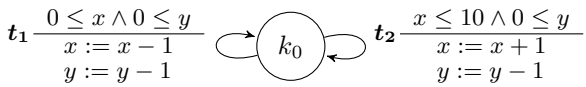
The notation $\equiv$ is a formal equality, which means that $\boldsymbol{x}$ can be eliminated and coefficients identified (from now on, we will use a simple equality sign = for it). In other words:

$$\exists \boldsymbol{\lambda} \in (\mathbb{R}^+)^n, \lambda_0 \in \mathbb{R}^+ \text{ such that } \boldsymbol{c} = \boldsymbol{\lambda}.A \text{ and } c_0 = \boldsymbol{\lambda}.\boldsymbol{a} + \lambda_0$$

This lemma transforms the search for a ranking function (with an infinite number of unknown coefficients) into the solving of a finite set of affine equations. In other words, it provides a *template* for the ranking function to be discovered.
The rest of the section is devoted to the explanation of the two algorithms (SAS10, PLDI15) on the simple case of **programs with one control point** [6] and the search for a monodimensional ranking function of maximal termination power. Both methods will be illustrated by the same running example depicted in Example 9.

[6] This restriction is only for the sake of readability. The two papers contain algorithms to deal with any number of control points.

Example 9 (Running example). *Consider the following automaton, where transitions are specified by* $(\frac{guard}{action})$*:*



Under initial assumptions $x = 5, y = 10$*, an invariant generator (*ASPIC*) gives the following inductive invariant for* $\mathscr{I}$*:*

$$\mathscr{I} = \{0 \leq x+1, x \leq 11, 0 \leq y+1, y \leq x+5, x+y \leq 15\}$$

In other words $\mathscr{I} = \{\boldsymbol{x} \mid \bigwedge_{i=1}^{m} \boldsymbol{a}_i.\boldsymbol{x} - b_i \geq 0\}$*. with*

$$\begin{array}{lccccc} \boldsymbol{a} = & \binom{1}{0} & \binom{-1}{0} & \binom{0}{1} & \binom{1}{-1} & \binom{-1}{-1} \\ \boldsymbol{b} = & -1 & -11 & -1 & -5 & -15 \end{array}$$

A possible monodimensional strict linear ranking function for this automaton is $\rho(x, y) = y + 1$*: this expression is always non-negative on* $\mathscr{I}$*, and both* t_1 *and* t_2 *make this expression strictly decrease.*

3.2.2 SAS10 method

The contributions of [ADFG10] are the following:

- An efficient algorithm to compute ranking functions on arbitrary C numerical programs. Our method, although greedy, is provably complete.
- A new method to derive polynomial upper bounds for the computational complexity (number of transitions) of the source program from the ranking functions.
- The method is implemented as a tool-chain and validated on a collection of test cases from the literature.

Method First, Equation 3.3 is rewritten as:

$$\forall (x, x') \in \tau, x \in \mathscr{I}, \rho(x) - \rho(x') \geq \epsilon_t \text{ with } 0 \leq \epsilon_t \leq 1 \tag{3.4}$$

which express a "weak" positivity constraint, and will allow us to define the "termination power" of a (weak) ranking function as the number of transitions t with $\epsilon_t = 1$ [7]. Of course we want to maximise this number.

In this paper, the LP problem we solve is deducted from the use of the Farkas' Lemma to linearize the constraints coming from equations 3.2 and 3.4:

- On equation 3.2, the ρ function we search for is an affine form, which is positive on the invariant $\mathscr{I}$. As $\mathscr{I} = \{\boldsymbol{x} \mid \bigwedge_{i=1}^{m} \boldsymbol{a}_i.\boldsymbol{x} - b_i \geq 0\}$, there exist constants $\boldsymbol{\alpha}, \alpha_0$ such that for all $\boldsymbol{x}$ the following equality holds:

  $$\rho = \boldsymbol{\lambda} \cdot \boldsymbol{x} + \lambda_0 = \boldsymbol{\alpha}(A\boldsymbol{x} - \boldsymbol{b}) + \alpha_0 \tag{3.5}$$

  by identifying the expressions on all lines of this last equality, we obtain one equation per λ_i, with m unknowns α_i.

- On equation 3.4, the "one-step difference function" $\rho(x) - \rho(x') - 1$ should be positive on each $\mathscr{Q}_t$, thus for each transition t there exists constants $\boldsymbol{\mu}, \mu_0$ such that for all $\boldsymbol{x}, \boldsymbol{x}'$ the following equality holds: $\rho(x) - \rho(x') - 1 = \boldsymbol{\mu_t}(Q_t\boldsymbol{x} - r_t) + \mu_{t0}$. We obtain a number of equations proportional to the number of transitions times the number of constraints appearing in the sets $\mathscr{Q}_t$s.

[7] ϵ_t can be an integer or a rational, for the rest of the section, we will suppose that it is an integer.

- The objective is then to maximise $\sum_t \epsilon_t$, that is, the total number of transitions t where the ranking function is *strictly positive*.

The result of the LP invocation is the values of α and β unknowns, from with we can derive λs, thus ρ, as well as the transitions on which this ranking function is not strictly positive.

Algorithm 4 Monolithic Monodimensional Termination (SAS10)

Input: A control point with τ its set of transitions, $\mathscr{I}$ invariant
Output: Yes+ρ/Abort

1: Apply Farkas on $\mathscr{I}$ to compute the positivity constraints.
2: Apply Farkas on each $t \in \mathscr{T}$ to compute the decreasing constraints.
3: Construct the LP instance, call the PIP solver.
4: From the result, compute λs and ρ and **return** ρ

Example 9 (continuing from p. 32)**.** *On our running example, recall that* $\mathscr{I} = (A.\boldsymbol{x} - \boldsymbol{b} \geq 0)$ *with:*

$$A = \begin{pmatrix} 1 & -1 & 0 & 1 & -1 \\ 0 & 0 & 1 & -1 & -1 \end{pmatrix}^T \text{ and } b = \begin{pmatrix} -1 & -11 & -1 & -5 & -15 \end{pmatrix}^T.$$

- *Equations3.5 expressing positivity are thus obtained by identifying terms of the following equality:*

$$\begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \lambda_0 = \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \\ \alpha_5 \end{pmatrix} \left(\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 1 & -1 \\ -1 & -1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} - \begin{pmatrix} -1 \\ -11 \\ -1 \\ -5 \\ -15 \end{pmatrix} \right) + \alpha_0,$$

  i.e.:

$$\lambda_1 x + \lambda_2 y + \lambda_0 = \alpha_1(x+1) + \alpha_2(-x+11) + \alpha_3(y+1) + \alpha_4(x-y+5) + \alpha_5(-x-y+15) + \alpha_0$$

  and finally the system: $\begin{cases} \lambda_1 = \alpha_1 - \alpha_2 + \alpha_4 - \alpha_5 \\ \lambda_2 = \alpha_3 - \alpha_4 - \alpha_5 \\ \lambda_0 = \alpha_1 + 11\alpha_2 + \alpha_3 + 5\alpha_4 + 15\alpha_5 \end{cases}$

- *Now for the equations expressing decrease:*
  - $\mathscr{Q}_{t1}$ *is the polyhedron defined by the set of 11 (affine) constraints* $\{(x, x')$ *such that* $x \in \mathscr{I} \wedge x \geq 0 \wedge y \geq 0 \wedge x' = x - 1 \wedge y' = y - 1\}$ *(5 constraints coming from* $\mathscr{I}$*, 6 from the transition, since equalities are transformed into 2 inequalities). Thus, we obtain the existence of* μ_i^1*,* $i \in [0, 11]$ *such that:*

$$(\lambda_1 x + \lambda_2 y + \lambda_0) - (\lambda_1 x' + \lambda_2 y' + \lambda_0) - \epsilon_1 = \mu_0^1 + \mu_1^1(x+1) + \ldots + \mu_{10}^1(y'-y+1) + \mu_{11}^1(y-y'-1).$$

    The term $\mu_1^1(x+1)$ *comes from the invariant* $\mathscr{I}$ *and the terms* $\mu_{10}^1(y'-y+1)$ *and* $\mu_{10}^1(y-y'-1)$ *come from the transition relation* t_1 *(*$y := y - 1$ *provides the two inequalities* $y' \geq y - 1$ *and* $y' \leq y - 1$*).*
    After identification, we obtain the following set of equations:

$$\begin{cases} \lambda_1 = \mu_1^1 - \mu_2^1 + \mu_4^1 - \mu_5^1 + \mu_6^1 - \mu_8^1 + \mu_9^1 \\ -\lambda_1 = \mu_8^1 - \mu_9 \\ \lambda_2 = \mu_3^1 - \mu_4^1 - \mu_5^1 + \mu_7^1 - \mu_{10}^1 + \mu_{11}^1 \\ -\lambda_2 = \mu_{10}^1 - \mu_{11}^1 \\ \epsilon_1 = \mu_1^1 + 11\mu_2^1 + \mu_3^1 + 5\mu_4^1 + 15\mu_5^1 + \mu_8^1 - \mu_9^1 + \mu_{10}^1 - \mu_{11}^1 \end{cases}$$

– *$\mathcal{Q}_{t2}$ is the polyhedron defined by the set of (affine) constraints* $\{(x, x')$ *such that* $x \in \mathcal{I} \wedge x \leq 10 \wedge y \geq 0 \wedge x' = x + 1 \wedge y' = y - 1\}$. *We obtain a set of 5 equalities with* 11 *new unknowns* μ_i^2.

All these equations, with positivity constraints for all the λ, α, μ *and* ϵ *variables, as well as* $\epsilon_i \leq 1$, *form a set of equations that we solve with a LP solver under the objective :* $Max(\epsilon_1 + \epsilon_2)$. *The LP solver we use (PIP) gives us a solution with* $\epsilon_1 = \epsilon_2 = 1, \lambda_1 = 0, \lambda_2 = 1, \lambda_0 = 1$, *thus we have found our ranking function* $\rho : (x, y) \mapsto y + 1$.

3.2.3 PLDI15 method

The contributions of [GMR15] are the following:

- A reformulation of the problem of finding ranking functions in a more compact Linear Programming instance than in [ADFG10].
- A way to build this instance lazily according to extremal counterexamples (a counterexample is a path where a candidate ranking function increases) inspired by the CEGAR [8] approach.
- The method is implemented as a standalone tool and benchmarked it against existing analysers.

Intuition The intuition we had came from experiments: our LP problems were quite huge comparing to the relative simplicity of the final outputs (in reality, ranking functions are often simple, they do not imply all program variables, in particular). We thus had the intuition that we could use a CEGAR-like approach that could iteratively refine a ranking function candidate. This approach is depicted in Figure 3.3: the algorithm we use takes the form of an iterative loop that stops if it is able to synthetize a ranking function or, like before, if the program may non terminate. To improve a given ranking function candidate, we search in the program if there is a path that makes this ranking function strictly increase (the "counter example"). From this path, we update the system of constraints for the computation for the next candidate.

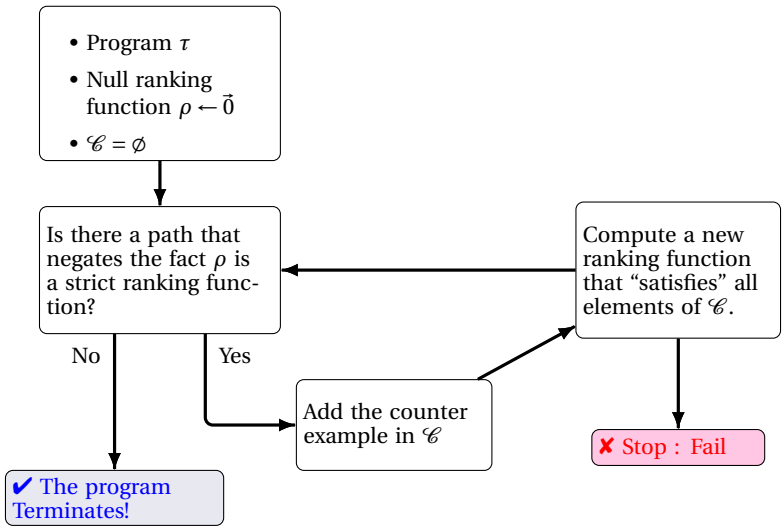


Figure 3.3 – Counter-example guided ranking function generation

[8] Counter-example guided abstraction refinement, [31]

Method As we saw on the running example, the number of unknowns in the LP instance we solve is growing quickly with the number of transitions and the number of constraints of the underlying polyhedra. However, the result we obtain is quite simple, and could have been generated with a lesser set of constraints.
The first difference with the previous paper is that now we consider the whole transition system τ as a big first order formula with disjunctions, thus the notion of "individual transition" is lost. Then, from the two constraints 3.2 and 3.3 we will only apply Farkas on the positivity constraints, which will enable us to construct LP instances with less unknown variables:

- For positivity, Equation 3.5 is still valid, which we simplify into:

$$\exists \alpha_i \text{ s.t. } \boldsymbol{\lambda} = \sum_{i=1}^{m} \alpha_i \boldsymbol{a}_i \tag{3.6}$$

  since once the condition $\boldsymbol{\lambda} = \sum_{i=1}^{m} \gamma_i \boldsymbol{a}_i$ is met then an appropriate choice of λ_0 can always be made.

- For the decreasing constraints, we show that they are equivalent to a *finite number* of constraints obtained by iterating on the generators of a polyhedron:

  Definition 5. *Let us denote by $\mathscr{P}_{\mathscr{I},\tau}$ the set of all reachable* 1*-step differences:*

$$\mathscr{P}_{\mathscr{I},\tau} = \{\boldsymbol{x} - \boldsymbol{x'} \mid \boldsymbol{x} \in \mathscr{I} \wedge (\boldsymbol{x}, \boldsymbol{x'}) \in \tau\}$$

  The closure of its convex hull will be denoted by $\mathscr{P}^H_{\mathscr{I},\tau}$.

  From 3.3 we can derive, using linearity, that $\boldsymbol{\lambda} \cdot \boldsymbol{u} \geq 0$ for all $\boldsymbol{u} \in \mathscr{P}_{\mathscr{I},\tau}$. Remark that this condition is left unchanged if we replace $\mathscr{P}_{\mathscr{I},\tau}$ by $\mathscr{P}^H_{\mathscr{I},\tau}$. Without loss of generality, we can now assume that $\mathscr{P}^H_{\mathscr{I},\tau}$ can be described by a finite number (m) of generators v_i, and the equation becomes $\forall 1 \leq i \leq m,\ \boldsymbol{\lambda} \cdot \boldsymbol{v}_i \geq 0$.

- Now the objective is to maximise the number of v_i such that $\boldsymbol{\lambda} \cdot \boldsymbol{v}_i > 0$.

In order not to explicitly enumerate all the $v_i s$, we define a family of Linear programming instances parametrised by a set V:

Definition 6. *Given $V = \{\boldsymbol{v}_j | 1 \leq j \leq N\}$ a set of generators of (the convex hull of) $\mathscr{P}_{\mathscr{I},\tau}$ and a set of vectors Constraints$(\mathscr{I}) = \{\boldsymbol{a}_i | 1 \leq i \leq m\}$, we denote by LP$(V, \text{Constraints}(\mathscr{I}))$ the following linear programming instance where α_i and ϵ_i are the unknowns:*

$$\begin{cases} \textit{Maximise} \sum_i \epsilon_i \textit{ s.t.} & \\ \alpha_1, \ldots, \alpha_m \geq 0 & \\ 0 \leq \epsilon_j \leq 1 & \textit{for all } 1 \leq j \leq N \\ \sum_{i=1}^{m} \alpha_i (\boldsymbol{v}_j . \boldsymbol{a}_i) \geq \epsilon_j & \textit{for all } 1 \leq j \leq N \end{cases}$$

The result of such an LP problem is *None* if the problem is unfeasible, or a valuation of the α_i and ϵ_i variables maximising the objective function. In the following, we denote as $\boldsymbol{\alpha}$ the vector with α_i components. The main result is the following:

Proposition 2. *Let $V = \{\boldsymbol{v}_j | 1 \leq j \leq N\}$ be a set of generators of the convex hull of $P_{\mathscr{I},\tau}$ and the set of vectors Constraints$(\mathscr{I}) = \{\boldsymbol{a}_i | 1 \leq i \leq m\}$ (constraints of $\mathscr{I}$). Then :*

- *LP$(V, \text{Constraints}(\mathscr{I}))$ is always feasible.*
- *LP$(V, \text{Constraints}(\mathscr{I}))$ gives $\alpha_i s$ such that $\rho(\boldsymbol{x}) = \boldsymbol{\lambda} \cdot \boldsymbol{x} + \lambda_0$ with $\boldsymbol{\lambda} = \sum_{i=1}^{m} \alpha_i \boldsymbol{a}_i$ and $\lambda_0 = \sum_{i=1}^{m} \alpha_i b_i$ is a weak ranking function of maximal power on V.*

Computing this set V (or, equivalently, any finite set between it and $\mathscr{P}_{\mathscr{I},\tau}$) may be expensive (and the cardinal of V may be exponential in the number of constraints). Obviously, we could compute a disjunctive normal form (DNF) for $\tau \wedge \mathscr{I}$, each disjunct denoting a convex polyhedron [14], compute their generators $(\boldsymbol{x}, \boldsymbol{x}')$ and collect all resulting $\boldsymbol{x}-\boldsymbol{x}'$; computing the DNF has exponential cost even if not all disjuncts are needed. The new approach described in this article will only compute extremal points as needed, as opposed to eagerly expanding the DNF.

A Wrong but Intuitive Algorithm Following a quite common technique [31] in the static analysis literature, we propose to compute the set of generators and the ranking function incrementally and at the same time. We propose a candidate (weak) ranking function, and search for generators v_i where this ranking is only weak. If it is not, we won, if not, we add this generator in a set $\mathscr{C}$.
We first propose a simple but somewhat wrong incremental algorithm, which gives the intuition of the correct algorithm detailed in [GMR15]. $\mathscr{C}$ is a set of vectors and $\rho(\boldsymbol{x}) = \boldsymbol{\lambda} \cdot \boldsymbol{x} + \lambda_0$ a candidate ranking function.

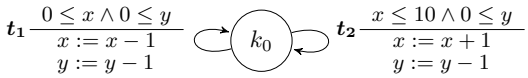
Algorithm 5 Incremental Monodimensional Termination

Input: A control point with τ its set of transitions, $\mathscr{I}$ invariant
Output: Yes+ρ/Abort

1: $\mathscr{C} := \emptyset$, $\rho(\boldsymbol{x}) := \boldsymbol{0}$, thus $\boldsymbol{\lambda} = \boldsymbol{0}$. ▷ Initialisation
2: **while** true **do**
3: Ask an SMT-solver if $\exists \boldsymbol{u}$ s.t. $\mathscr{I} \wedge \tau \wedge \boldsymbol{\lambda} \cdot \boldsymbol{u} \leq 0$.
4: If Unsat, ρ is a *strict* ranking function. **return** ρ
5: If Sat, we get $\boldsymbol{u}$ from the SMT model.
6: Add $\boldsymbol{u}$ to $\mathscr{C}$.
7: Call $LP(\mathscr{C}, Constraints(\mathscr{I}))$ ▷ compute a new (candidate) ranking function
8: **end while**;

Let us illustrate this algorithm on our running example.

Example 9 (continuing from p. 32)**.** *Let us recall the automaton and its transition relation:*



$$\tau = \left\{ (x, y, x', y') \;\middle|\; \begin{array}{c} x \leq 10 \wedge 0 \leq y \wedge x' = x+1 \wedge y' = y-1 \\ \vee \\ 0 \leq x \wedge 0 \leq y \wedge x' = x-1 \wedge y' = y-1 \end{array} \right\}$$

Recall that we are looking for $\rho(\boldsymbol{x}) = \boldsymbol{\lambda} \cdot \boldsymbol{x} + \lambda_0$ with $\boldsymbol{\lambda}$ a positive linear combination of the $\boldsymbol{a}_i$s.
We will follow Algorithm 5 step by step, repeated steps will be denoted by '.

1. *Beginning with $\mathscr{C} = \{\}$ and $\rho(\boldsymbol{x}) = 0$, that is: $\boldsymbol{\lambda} = \boldsymbol{0}$ and $\lambda_0 = 0$. The unknown vector $\boldsymbol{u}$ is always $\binom{x-x'}{y-y'}$.*

First iteration.

2. *$Sat\,(\mathscr{I} \wedge \tau \wedge \boldsymbol{0} \cdot \boldsymbol{u} \leq 0)$?*
   Yes and we have the model $\boldsymbol{u} = \binom{-1}{1}$
3. *$\mathscr{C} \leftarrow \left\{\binom{-1}{1}\right\}$*
4. *Call $LP\,(\mathscr{C}, Constraints(\mathscr{I}))$ =*

$$\left\{ \begin{array}{l} \textit{Maximise } \epsilon_1 \textit{ s.t.} \\ \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5 \geq 0 \\ 0 \leq \epsilon_1 \leq 1 \\ -\alpha_1 + \alpha_2 + \alpha_3 - 2\alpha_4 \geq \epsilon_1 \end{array} \right.$$

   (The last constraint comes from $\boldsymbol{u} \cdot \boldsymbol{a}_i$ for each i.)
   The solver answers $\alpha_2 = 1$, $\alpha_1 = \alpha_3 = \alpha_4 = \alpha_5 = 0$.
   We deduce $\boldsymbol{\lambda} = \boldsymbol{a}_2 = \binom{-1}{0}$ and $\lambda_0 = b_2 = 11$.

Second iteration.

2'. $Sat\left(\mathscr{I} \wedge \tau \wedge \binom{-1}{0} \cdot \boldsymbol{u} \leq 0\right)$?

Yes and we have the model $\boldsymbol{u} = \binom{1}{1}$

3'. $\mathscr{C} \leftarrow \left\{\binom{-1}{1}; \binom{1}{1}\right\}$

4'. *Call LP* $(\mathscr{C}, Constraints(\mathscr{I})) =$

$$\begin{cases} Maximise\ \epsilon_1 + \epsilon_2\ s.t. \\ \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5 \geq 0 \\ 0 \leq \epsilon_1, \epsilon_2 \leq 1 \\ -\alpha_1 + \alpha_2 + \alpha_3 - 2\alpha_4 \geq \epsilon_1 \\ \alpha_1 - \alpha_2 + \alpha_3 - 2\alpha_5 \geq \epsilon_2 \end{cases}$$

The solver answers $\alpha_3 = 1$, $\alpha_1 = \alpha_2 = \alpha_4 = \alpha_5 = 0$. *We deduce* $\boldsymbol{\lambda} = \boldsymbol{a}_3 = \binom{0}{1}$ *and* $\lambda_0 = b_3 = 1$.

Third iteration.

2". $Sat(\mathscr{I} \wedge \tau \wedge y - y' \leq 0)$ *? No, we stop.*

Return. *We have* $\boldsymbol{\lambda} = \binom{0}{1}$ *and* $\lambda_0 = 1$. *We obtain* $\rho(x, y) = y + 1$, *a strict ranking function for* $(\tau, \mathscr{I})$.
Let us point out the fact that the two models $\boldsymbol{u}$ *we obtained correspond to the two transitions* $\boldsymbol{t}_1$ *and* $\boldsymbol{t}_2$.

Remarks Unfortunately, this simple algorithm suffers from two problems which may prevent its termination:

- First, termination is guaranteed only if the models provided by the SMT tests come from a finite set (then the number of iterations is bounded by the cardinality of that set) and also lie in the boundary of the convex hull $\mathscr{P}^H_{\mathscr{I},\tau}$ (if not we could accumulate unoptimally tight constraints). To solve this problem, we impose that the model for the SMT-test should minimise $\boldsymbol{\lambda} \cdot \boldsymbol{u}$, as in "optimisation modulo theory [87, 98]."

- Second, even if the first issue is resolved, the above algorithm terminates only if there is a strict ranking function ($\boldsymbol{\lambda} \cdot \boldsymbol{v} > 0$ for all $\boldsymbol{v} \in \mathscr{V}$). Indeed, termination is ensured by the algorithm never choosing twice the same $\boldsymbol{u} = \boldsymbol{x} - \boldsymbol{x'}$, which is the case if there exists a (weak/quasi) ranking function such that $\boldsymbol{\lambda} \cdot \boldsymbol{u} > 0$. But what if the SMT-solver picks $\boldsymbol{u}$ such that *all* weak ranking functions $\boldsymbol{\lambda}$ satisfy $\boldsymbol{\lambda} \cdot \boldsymbol{u} = 0$? In this case, the algorithm may not terminate, always picking the same $\boldsymbol{u}$. To solve that purpose, we force the SMT solver to always search for $\boldsymbol{u}$s that are *not* linear combinations of the preceding $\boldsymbol{u}$s.

From this remark, we finally construct an algorithm that satisfies the following proposition:

Proposition 3. *Algorithm 1 of [GMR15] always terminates and returns a weak ranking function of maximal termination power.*

3.2.4 Comparison of the two approaches

Despite the fact that they use the same initial ideas and Linear Programming to search for a monodimensional ranking function of maximal ranking function and finally build a multidimensional one with an iterative algorithm, the two preceding articles have some crucial differences:

- In SAS10, the transition system is given as a union of disjoint functional transitions, in PLDI15 it can be any first order formula.

- In SAS10, the Farkas' lemma is applied to both decreasing constraints and positivity constraints, in PLDI15 we only apply Farkas on the positivity constraints.

- For decreasing constraints, in SAS10 we construct a LP instance whose size is proportional to the number of *constraints* of a given transition, for all individual transitions. In PLDI15 we enumerate the *generators* of a bigger set, but lazily. We expected the number of generators actually enumerated to be rather small in practice since "actual programs have simple ranking functions" (build on a limited number of variables, thus can be obtained with a limited number of linear combination of constraints/generators). This has been confirmed by our experiments.
- In SAS10 we only rely on the use of a LP (rational) solver, in PLDI15 we also rely on a Max-SMT solver, which is capable of maximising a linear function on any first order affine formula.
- Experiments show that in PLDI15's implementation our LP instances are one-two orders of magnitude smaller than in SAS10's one (see Section 3.2.6).

3.2.5 From ranking to worst-case

We define the Worst-case Computational Complexity (WCCC) as an upper bound on the number of transitions that are executed, given an initial value of the counter variables.
With this definition, one could over-approximate the WCCC of a terminating program by the total number of reachable states (because a finite trace cannot contain twice the same state), i.e., $\text{WCCC} \leq \sum_k \#\tilde{\mathscr{R}}_k$ or even more conservatively $\text{WCCC} \leq \sum_k \#\tilde{\mathscr{I}}_k$ as $\mathscr{R}_k$ is itself over-approximated by $\mathscr{I}_k$. [9] This is a very rough over-approximation but, even worse, this technique can lead to an infinite WCCC, even for a terminating automaton, if some invariant $\mathscr{I}_k$ is unbounded. Rather, we can use the ranking function itself to prune the invariant sets. Indeed, consider a trace $(k_0, \boldsymbol{x}_0), \ldots, (k_p, \boldsymbol{x}_p)$ in the execution of the automaton. By definition of a ranking function, $\rho(k_{i+1}, \boldsymbol{x}_{i+1}) \prec \rho(k_i, \boldsymbol{x}_i)$. Since $\prec$ is a strict order, it follows by transitivity that all $\rho(k_i, \boldsymbol{x}_i)$ are distinct in $\mathscr{W}$. Hence, the length of the trace is bounded by the cardinal of the co-domain of ρ:

$$\text{WCCC} \leq \#\bigcup_k \rho(k, \tilde{\mathscr{I}}_k) \leq \sum_k \#\rho(k, \tilde{\mathscr{I}}_k) \tag{3.7}$$

The first inequality is more accurate but harder to compute as it involves a union of sets. Instead, we use the second less accurate inequality. In [ADFG10] we propose a simple solution based on a computation of the number of the numerical points of a given $\mathbb{Z}$-polyhedron (intersection of an integral lattice, here $\mathbb{Z}^n$, and a polyhedron, here $\mathscr{I}$). The main tool we use for that purpose are Ehrhart polynomials [32, 105]. The experimental results of Section 3.2.6 show that these evaluations are precise enough for classical algorithms of the literature.

3.2.6 Experimental results

Implementation For SAS10, we implemented a tool suite with C2FSM and ASPIC as front-ends (See Section 2.2.1) The third tool, RANK (3000 lines of C++) , from the integer interpreted automaton and the invariants given by ASPIC, tries to prove the termination of the program by computing (multidimensional affine) ranking functions. In case of success, RANK computes the worst-case computational complexity of the program. Also, in case of failure (denoted by "Don't Know"), RANK tries to exhibit a counterexample that causes non-termination. The linear programs involved in the termination part are solved thanks to the PIP tool. The PIP tool may fail if the generated LP instance is too big.
For PLDI15, we implemented our prototype TERMITE in 3k lines of OCAML [10]. As depicted in Figure 3.5 Termite uses LLVM[11] to compile C code into a Single Static Assignment intermediate representation [43]. PAGAI [65, 67] is used to compute invariants from the LLVM IR. The transition relation

[9] Here, the notation $\tilde{\mathscr{S}}$ means the integral points in a set $\mathscr{S}$, and $\#\tilde{\mathscr{S}}$ denotes the cardinal of $\tilde{\mathscr{S}}$.
[10] http://termite-analyser.github.io/
[11] http://llvm.org/

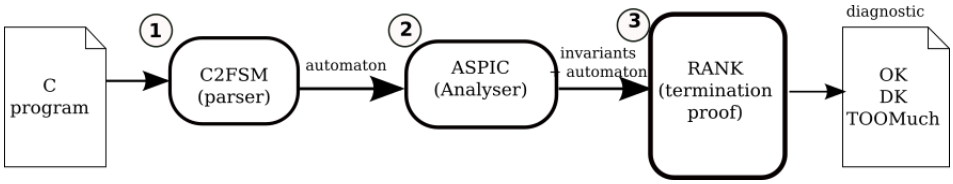


Figure 3.4 – RANK tool-chain. DK is "I do not know". Too much variables indicates that we generated a LP instance that is too big for PIP

is produced from these invariants and the LLVM IR. Then, the core analyser implements the multidimensional, multiple control point algorithm described herein. Z3[12] answers (optimising) SMT and LP queries.

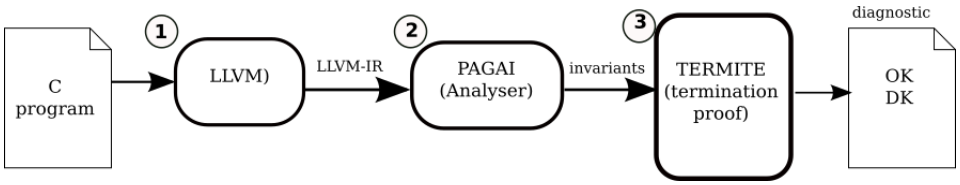


Figure 3.5 – TERMITE tool-chain

Results The experimental results first show that the method is expressive enough to prove termination and compute the WCCC of challenging polynomial programs from the literature. Both tools RANK (SAS10) and TERMITE (PLDI15) perform well on these benchmarks.
We also show that the size of the LP instances are drastically decreased by the use of our incremental technique implemented in TERMITE.
Finally, we compare TERMITE with existing state-of-the art tools, and show that despite the fact the tool is only a prototype, it compares well with other more mature tools.

Expressivity of the tools Our tool chain has been tested on a set of benchmarks from the literature. Most of the examples were collected in [29] from many other papers dealing with termination analysis. They can be found at the following web address: `http://compsys-tools.ens-lyon.fr/wtc/index.html`.
On Tables 3.1 and 3.1, the first two columns identify the test cases. The symbol ♣ indicates a test case we developed to check our algorithm [13]. Columns 3 to 5 give statistics about the (generated) automaton: number of relevant variables, of control points, of transitions. The next column gives the dimension of the ranking function found by our algorithm. The next column gives the timing measurements on a 2 GHz Pentium with 1 GByte of memory running Debian 2.6. The "Analysis" measures include the invariants computation time from the ASPIC file, the computation of the ranking function, and the evaluation of the WCCC. In general, the WCCC is the maximum of several parametric expressions valid on different domains of the program inputs. To make the table simpler, the last column gives only the expression that can reach the maximum value.

Scalability of TERMITE compared to RANK We compared the two tools on an modified version of the WTC test suite. On these benchmarks, due to limitations in its front-end and invariant generator, RANK could only run on 46 files from the WTC test suite, solving 25 of them in total time 76 ms,

[12] `https://github.com/Z3Prover`
[13] These examples have been integrated in the SVCOMP benchmark `https://sv-comp.sosy-lab.org/2017/`

Table 3.1 – Experimental results SAS10 for RANK (WTC benchmark)

Name	Ref	Vars	States	Trans	dim ρ	Time (s)	WCCC
easy1	[29]	3	4	5	1	0.2	43
easy2	[29]	3	3	3	1	0.07	z_0+3
ackermann	[15]	2	7	7	1	0.07	$4m_0+5$
terminate	[33]	3	1	1	1	0.08	$i_0+j_0+k_0+102$
gcd	[25]	2	5	1	1	0.2	x_0+y_0+2
rsd	♣	3	3	4	1	0.2	$\frac{r_o^2}{2}+\frac{5r_0}{2}+5$
nd_loop	♣	2	4	6	1	0.05	22
wcet2	♣	2	3	5	1	0.15	$55-12i_0$
relation1	♣	2	4	4	1	0.14	4
ndecr	♣	2	4	4	1	0.08	i_0+2
perfect	[27]	4	5	12	3	0.35	$2+\frac{3x_0}{2}+\frac{x_0^2}{2}$
cousot16	[38]	2	3	4	1	0.05	106
random2d	[29]	5	10	21	1	1.1	$6N_0+3$
random1d	[29]	3	4	6	2	0.1	$max+3$
wise	♣	2	6	10	2	0.11	$1+\lvert x_0-y_0\rvert$
wcet1	♣	3	6	8	2	0.25	n_0+2
complex	[58]	2	4	11	2	0.28	$1560-9b_0-45a_0$
nestedLoop	[58]	6	5	12	3	1.3	$n_0m_0+2N_0+n_0+3$
exmini	♣	4	3	6	2	0.1	$104+k_0-j_0-x_0$
aaron2	[29]	3	6	10	2	0.2	$2(x_0-y_0)+5$
while2	♣	3	3	4	2	0.1	$3+2N+N^2$
cousot9	[38]	3	4	5	2	0.2	$3+7/2N+5/2N^2$
ax	♣	4	3	6	3	0.16	$n_0^2-n_0+2$
loops	[89]	3	4	5	2	0.15	N^2-N+5
counterex1	♣	4	5	13	3	1.1	x_0+2
determinant	[27]	4	6	7	4	0.15	$\frac{N^3}{3}+\frac{N^2}{2}+\frac{N}{6}+3$
maccarthy91	[34]	4	5	18	2	1.2	$13773/11-1363/110x$
speedpldi2	[58]	4	4	9	2	1.0	$2n_0-m_0+3$
speedpldi3	[58]	4	6	11	3	1.3	$n_0m_0+n_0+4$
speedpldi4	[58]	3	6	11	2	1.1	$2n_0-2m_0+5$
speedFails4	[58]	5	4	10	2	1.1	n_0-x_0+4

Table 3.2 – Experimental results SAS10 for RANK (Sorting programs)

Name	Ref	Vars	States	Trans	dim ρ	Time (s)	WCCC
realselect	♣	7	5	10	3	0.4	$\frac{N^2}{2}+\frac{3N}{2}+1$
insertsort	♣	3	6	7	2	0.22	$\frac{N^2}{2}+\frac{3N}{2}+1$
sipmabubble	[29]	4	10	17	3	0.33	N^2+2N+3
realbubble	♣	6	5	11	3	0.4	N^2+2
realshellsort	♣	8	8	13	4	1.1	$\frac{N^3}{6}-\frac{N}{6}$
realheapsort	♣	10	11	31	3	2.8	$4N^2-11N+9$

Table 3.3 – PLDI15: Comparison of TERMITE and some of the state-of-the-art Termination tools. For each benchmark suite, the total number of benchmarks and the number of benchmarks proved to terminate by each tool are given. Timings, in milliseconds, exclude the front-end for TERMITE and LOOPUS and the invariant generator for TERMITE. (l,c) are the average number of lines and columns of the linear programming instances.

Suite	# benchmarks	Termite			Loopus		AProVE		Ultimate	
		# success	time	(l,c)	# success	time	# success	time	# success	time
PolyBench	30	22	117	(9,3)	30	37	0	2100	0	2750
Sorts	6	5	126	(15,4)	3	67	0	12230	0	2980
TermComp	129	119	12	(2,1)	78	15	111	9757	85	5863
WTC	58	46	64	(5,2)	33	48	36	11740	40	4536

with average linear programming problem (*lines*, *columns*) = (584,229), much higher than TERMITE's (5,2). On some examples however, RANK is able to prove termination that TERMITE is unable to prove. This essentially comes from the combination of its front-end C2FSM and its invariant generator ASPIC that sometimes perform clever graph transformations on the control-flow graph and thus are able to synthesise better invariants than PAGAI (the invariant generator used for TERMITE).

Comparison of TERMITE and state-of-the-art tools In Table 3.3 We have compared TERMITE with LOOPUS[14] [112], APROVE[15] [55], Ultimate Büchi Automizer[16], RANK [ADFG13] (with the C2FSM front-end and ASPIC[17] [GS14] accelerating invariant generator), on examples from POLYBENCH[18], WTC (V2)[19], the termination competition[20] and some sorting algorithms (Table 3.3).
TERMITE solves 33% more examples than LOOPUS, in twice the time. APROVE and ULTIMATE are considerably slower.

All these tools have different approaches to prove termination:

- APROVE uses a complex front-end and relies on term rewriting systems.
- The ULTIMATE technique essentially consist in covering the set of program traces by ω-regular languages of terminating traces ("modules"). The termination argument for each module can be obtained from a variety of approaches; thus their approach truly is a meta-approach since a variety of sub-provers can be used. We in fact think that our technique could be adapted into a sub-prover for their system.
- The algorithm in LOOPUS [112] is similar to ours in some ways; however, it explores all paths in a given loop, and uses heuristics to syntactically derive ranking functions from the transitions of the loop. In our experiments, Loopus is able to quickly conclude for termination in many cases since simple arguments are very often sufficient to prove that a given loop terminates

Difficulties A termination analyser typically consists in 1) a front-end 2) an invariant generator 3) a termination analysis. One therefore compares whole tool-chains instead of only the termination analyses. One difficulty is that some front-ends are more picky than others; for instance, APROVE's front-end crashes on some LLVM opcodes. We have made some reasonable efforts to have our examples accepted by the various tools when we could identify some obvious reason why they could not be processed by the front-end or invariant generator.

[14] http://forsyte.at/software/loopus/
[15] http://aprove.informatik.rwth-aachen.de/
[16] http://ultimate.informatik.uni-freiburg.de/BuchiAutomizer/
[17] http://laure.gonnord.org/pro/aspic/aspic.html
[18] http://www.cs.ucla.edu/~pouchet/software/polybench/
[19] http://compsys-tools.ens-lyon.fr/wtc/index.html
[20] http://termination-portal.org/wiki/Termination_Competition

3.3 Improving scalability

Improving the scalability of a given termination algorithm might not be sufficient, and clearly none of our two analysers is capable to deal with large C programs:

- Because of technical reasons: the robustness of the parsers, the fact that we have to deal with possibly a large number of statement types.
- Because the underlying invariant generators do not scale well: for challenging programs polyhedral invariants are really necessary, but their generation is expensive. Improving the scalability of such techniques is a challenging problem as we saw in the previous chapter.
- Because of more intrinsic reasons: a given program is made of a possibly large number of procedures, and only examining each at its turn might not be sufficient. Moreover for a given procedure, the number of constraints that are generated might be too big for some LP solvers.

In the research report [GAA12], we propose an incremental scalable analysis implemented in a tool called STOP (short paper [AAG12]). This tool is based on two classical techniques of static analysis, slicing, and summaries. The experiments were made above the RANK tool-chain.

Example 10 (Running example: mergesort)**.** *We detail our method on an implementation of the merge sort of an array. The code is taken from [34] and is depicted in Figure 3.6. For sake of readability, we drew boxes around inner-loops, and commented the end of outer loops. For this example, the* RANK *tool-suite fails with* `TWO_MANY_VARIABLES` *error because the size of the underlying linear programming instance given to PIP pis too big.*

3.3.1 Method

A slicing for termination In Algorithm 6 we depict a simple slicing algorithm based on the notion of *definition sites*. It relies on the sub-function DEFINITION_SITES that computes the *reaching definitions* [1] $\mathrm{RD}_{\mathtt{stmt}}(\mathtt{xi})$ of each variable `xi` read by a given statement `stmt`. The result is a set of assignments which can possibly define the value of `xi` read by `stmt`. The property of this algorithm is that it computes a *safe abstraction* of all functions for the termination property.

Algorithm 6 Termination-Specific Slicing Algorithm

Input: A function body
Output: A semantically equivalent (w.r.t.) termination function body

```
function DO_SLICE(function_body)
    while_loop_set = set of while statements
    SLICE(while_loop_set)      ▷ At this point, all the statements belonging to the slice are marked
end function
function SLICE(stmt_set)
    for all stmt ∈ stmt_set do
        if is_not_marked(stmt) then
            mark(stmt)                                          ▷ Add stmt to the slice
            SLICE(DEFINITION_SITES(stmt))
        end if
    end for
end function
```

Example 10 (continuing from p. 42)**.** *The result of the slicing algorithm on the end the motivating example code is depicted in Figure 3.7*

```
1  int main() {
2    int n,i,j,k,l,t,h,m,p,q,r;
3    int up;     /* really boolean */
4    int a[2*n+1]
5    up = 1;  p = 1;
6
7   loop4 : do{ // sorting a
8      h = 1;
9      m = n;
10     if(up == 1){
11       i = 1;
12       j = n;
13       k = n+1;
14       l = 2*n;
15     } else {
16       k = 1;
17       l = n;
18       i = n+1;
19       j = 2*n;
20     }
21   loop5 : do{
22       if(m >= p)
23         q = p;
24       else q = m;
25      m = m-q;
26       if(m >= p)
27         r = p;
28       else r = m;
29      m = m-r;
30
31     loop0 : while(q>0 && r>0) {
32         if(a[i] < a[j]){
33           a[k] = a[i];
34           k = k+h;
35           i = i+1;
36           q = q-1;
37         } else {
38           a[k] = a[j];
39           k = k+h;
40           j = j-1;
41           r = r-1;}
42
43
44     loop1 : while(r > 0){
45         a[k] = a[j];
46         k = k+h;
47         j = j-1;
48         r = r-1;
49 }
50
51
52     loop2 : while(q > 0){
53         a[k] = a[i];
54         k = k+h;
55         i = i+1;
56         q = q-1;
57 }
58
59       h = -h;
60       t = k;
61       k = l;
62       l = t;
63     } while(m > 0);//end of loop5
64
65     up = 1-up;
66     p = 2*p;
67
68   } while(p<n);//end of loop4
69
70 //final copy of the array
71 // in the first half of a
72  if(up == 0){
73     i = 1;
74
75     loop3 : while(i <= n){
76       a[i] = a[i+n];
77       i = i+1;
78     }
79
80     }//end if test up==0
81
82   return 0;
83 }
```

Figure 3.6 – Our motivating example: iterative merge sort

```
1 if(up == 0){
2   i = 1;
3   while(i <= n){
4     a[i] = a[i+n];
5     i = i+1;
6   }
7 }
```

(a) Before slicing

```
1 if (up == 0) {
2   i = 1;
3   while(i <= n){
4     i = (i + 1);
5   }
6 }
```

(b) After slicing

Figure 3.7 – Slicing Algorithm operating on a piece of code (loop3)

A summary-based algorithm for termination Algorithm 7 depicts the modular algorithm we use to incrementally prove termination of a given piece of code.. For the sake of readability we only provide the algorithm for while loops. The rest is straightforward. This algorithm strongly depends on the notion of summary, that we recall here:

Definition 7 (Summary)**.** *Let C be a code and $R_C(x_0, y_0, \ldots, x, y)$ be an over-approximation of the re-*

lation between initial variables $x_0, y_0, \ldots$ and final variables $x, y, \ldots$. Then the following code is called a summary *of C :*

```
x0 = x; y0 = y; ...
x = random(); y = random();...
if(!R_C(x0,y0, ..., x, y, ...)) abort
```

The *summary* of C is obviously an abstraction of C and thus can be used for proving the termination of C.

Example 10 (continuing from p. 42)**.** *In the running example, loop3 can be replaced by:*

```
i0=i;n0=n;
i=random();y=random();
if (i > n) abort
```

Algorithm 7 Scalable Algorithm for proving termination (partial)

Input: A statement, and invariants
Output: OK or Don't Know

1: **function** MODULARTERM(statement,pcinvs)
2: **if** statement = `while cond do S` **then**
3: $res \leftarrow ModularTerm(\mathtt{S}, pcinvs)$
4: **if** res = Don't Know **then**
5: **return** Don't Know
6: **else**
7: $context \leftarrow getContext(statement, pcinvs)$ ▷ provides precise invariants for the sub-automaton to be analysed.
8: $res \leftarrow Rank(context, statement)$ ▷ launch of the whole toolchain depicted in Figure 3.4.
9: **if** res = Don't Know **then**
10: **return** Don't Know
11: **else**
12: Compute a summary of the loop and replace in the code.
13: **return** *OK*
14: **end if**
15: **end if**
16: **end if**
17: **end function**

The main function MODULARTERM takes as input a statement and a structure that is able to give an over approximation of the context of each statement of the program under analysis. These invariants are computed using ASPIC as input/output relations.
MODULARTERM proceeds the abstract syntactic tree with a depth-first traversal, evaluating each sub-tree to the value `OK` if it terminates, or `Don't Know` if the analysis does not succeed to answer. If a sub-tree is evaluated to `Don't Know`, the analysis fails and stops with `Don't Know` output.
Computing summaries can also be applied at the granularity of functions, but we didn't implement it.

3.3.2 Experimental results

We implemented our method as a driver over the RANK tool-chain, written in C++. The total number of LOC is 3000 for the driver itself, and 150 additional lines for statistics. The code intensively uses

Table 3.4 – Experiments detailed in [GAA12] . For each benchmark, we give its number of programs, the average number of lines of codes before (`LoCA`) and after (`LoCB`) slicing, `Regressions` denote the number of regressions (examples that were proved `OK` with `WTC` and for which STOP now answers `Don't Know`). We also give average execution times of each method.

Benchmark	**#Progs**	**LoCB**	**LoCA**	**Regressions**	**WTC(s)**	**SToP(s)**
WTC	50	28	21	3	0.92	2.45
Sorting	6	55	39	0	3.66	3.52

the source-to-source compiler infrastructure Rose ([90]) : functions for parsing, constructing flow graphs, searching in the code structure, pretty printing and instrumenting C codes.
Table 3.4 give an extract of the experiments on we made with our tool STOP (the entire benchmark is available on `http://compsys-tools.ens-lyon.fr/stop`).
The experiments were done on a Dual Core 2Ghz. As we expected, the execution times on middle-sized examples are much larger with STOP than `WTC`, mainly because of the cost of slicing and producing intermediate files. These results could be enhanced by hand by calling `WTC` on larger subprograms (some nested loops can be handled with a unique call to `WTC`). For sorting functions, the method shows it pertinence in terms of timing results and in terms of precision (it handles `sipmamergesort`, which was not handled by RANK before, as the generated LP instance was too big for PIP) in 22 seconds, but we had to manage the very last step by hand by providing a coarser abstraction than the one obtained by ASPIC). Thus STOP is able to deal with larger programs than WTC and seems to scale well. Unfortunately, due to lack of time and manpower we did not make any further experiments, and STOP is no longer maintained.

3.4 A case study: termination for JIT compilers

In the paper [RAPG14], we study the relevance of fast and simple solutions to compute approximations of the number of iterations of loops (*loop trip count*) of imperative real-world programs. The context of this work is the use of these approximations in compiler optimisations: most of the time, the optimisations yield greater benefits for large trip counts, and are either innocuous or detrimental for small ones.
In this particular work, we argue that, although predicting exactly the trip count of a loop is undecidable, most of the time, there is no need to use computationally expensive state-of-the-art methods to compute (an approximation of) it.
We support our position with an actual case study. We show that a fast predictor can be used to speedup the JavaScript JIT compiler of Firefox - one of the most well-engineered runtime environments in use today.

3.4.1 Method

Key observation: real-world loops are simple The motivation of using simple heuristics come from our early work on *abstract acceleration* ([GH06], [GS14]), in which we compute an overapproximation of the transitive closure of loops at low cost. The experiments show that in many of the analysed programs, static analysers get more precise results just because they are able to precisely deal with loops of the form `for (int i = M; i < N; i++)` (which is an example of a *accelerable loop*) in the most precise way.
Following these experiments, we decided to implement a light and fast heuristic to dynamically compute an approximation of the number of executions of loop to be executed, whose main specification is to be as precise as possible in the case of such simple loops.

Fast Trip Count Prediction at runtime We apply our heuristic dynamically. In other words, we instrument a program - or its interpreter - to estimate the trip count immediately before the first iteration of the loops. Our instrumentation inspects the state of the variables used in the stop condition of each loop.
In the sequel, we only consider perfect loops with a single "interval" exit-condition, *i.e.* loops of the form : `while (e1 ⋈ e2) { some computation }`. For this kind of loops, we assume that the trip count will be the absolute difference between e_1 and e_2. For instance, in a loop such as `for (int i = M; i < N; i++)`, we say that its trip count will be `|val(N) - val(i)|` (`val(x)` is the runtime value of `x` when the test is performed).
For each loop of this form, we insert a new instruction before the loop, according to Algorithm 8[21]. Let us point out that this algorithm only performs a single $O(1)$ operation per loop, without actually looking inside the body of the loop.
However, for loops of the form `for (int i = M; i < N; i=i+s)` (s and N invariants in the loops), this heuristic gives an overapproximation of the total number of loops, and the most precise result if $s = 1$. There is no guarantee of precision for the general case, of course, but the experiments will show that this simple-blind instrumentation fits our needs in practice. Because our heuristic is so simple, we can execute it quickly. This perfectly suits the needs of a JIT compiler.

Algorithm 8 Trip Count Instrumentation Heuristic

Input: Loop L
Output: Loop L' with new instructions that estimate its minimum trip count

1: **if** comparison $e_1 < e_2$ controls loop exit **then**
2: Insert instruction $tripcount = |e_1 - e_2|$ before L, giving L'.
3: **end if**

3.4.2 Application: Hot code detection in JIT compilers

Virtual environments that combine interpretation and compilation face a difficult question: when to invoke the JIT compiler [47]? Premature compilation might produce binaries that do not run long enough to amortise the cost of the JIT transformation. On the other hand, late compilation might delay the optimisation of critical parts of the program. As an example, the Firefox browser separates native execution in two parts. After a few rounds of interpretation, the *baseline compiler* translates the program into non-optimised native code. Once this basic native code is deemed hot, it is recompiled, this time by *IonMonkey*, an optimising compiler. Figure 3.8 illustrates this behaviour.

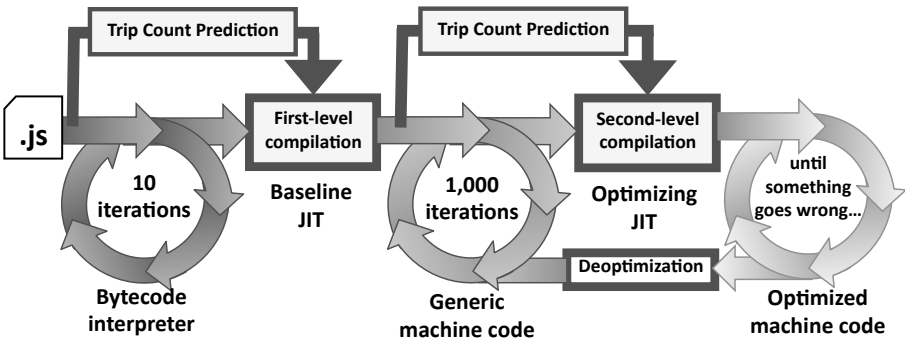


Figure 3.8 – Life cycle of a JavaScript program in the Mozilla runtime: We can perform trip count prediction at two different execution stages.

[21] Obvious adaptations are necessary to handle ≤ and ≥.

Table 3.5 – Hit rate of our simple heuristic for JavaScript (Bottom). N denotes the number of iterations of a loop observed via profiling, so the interval $[N, N]$ gives us the exact predictions. We only produce estimates for interval loops, that account for 71% of the loops in our benchmarks.

JavaScript	$]1, \sqrt{N}]$	$]\sqrt{N}, N/2]$	$]N/2, N]$	$[N, N]$	$[N, 2N[$	$[2N, N^2[$	$[N^2, \infty[$
SunSpider 1.0	0.0%	0.0%	0.0%	89.2%	2.0%	4.7%	4.1%
V8 v6	0.6%	1.7%	0.0%	94.8%	2.3%	0.0%	0.6%
Kraken 1.1	0.8%	0.0%	3.2%	83.9%	1.6%	2.4%	8.1%

SunSpider 1.0	
3d-cube	-1%
3d-morph	-1%
3d-raytrace	1%
access-binary-trees	0%
access-fannkuch	2%
access-nbody	-1%
access-nsieve	2%
bitops-3bit-in-byte	0%
bitops-bits-in-byte	1%
bitops-bitwise-and	3%
bitops-nsieve-bits	3%
ctrlflow-recursive	-1%
crypto-aes	0%
crypto-md5	3%
crypto-sha1	10%
date-format-tofte	2%
date-format-xparb	2%

SunSpider 1.0	
math-cordic	2%
math-partial-sums	-1%
math-spectral-norm	-1%
regexp-dna	0%
string-base64	24%
string-fasta	-7%
string-tagcloud	1%
string-unpack-code	0%
string-validate-input	-5%

V8 version 6.0	
crypto	0%
deltablue	2%
earley-boyer	0%
raytrace	0%
regexp	3%
richards	-1%
splay	1%

Kraken 1.1	
ai-astar	1%
audio-beat-detect	0%
audio-dft	-4%
audio-fft	0%
audio-oscillator	1%
img-gaussian-blur	-3%
imaging-darkroom	-3%
imaging-desaturate	0%
json-parse-financial	1%
json-stringify-tinder	1%
crypto-aes	6%
crypto-ccm	2%
crypto-pbkdf2	7%
crypto-sha256-itrv	6%

Figure 3.9 – Speedup due to trip count predictor for benchmarks distributed with Firefox.

We have deployed our predictor in the interpreter and in the baseline compiler used in Firefox, as we illustrate in Figure 3.8. The current distribution of Firefox calls the baseline compiler after 10 events, and the optimising compiler after 1,000 events. If we predict that a loop will run for more than 10 iterations, we call the baseline compiler immediately, bypassing the warm-up period. Similarly, once in native mode, we call the optimising compiler immediately upon finding a loop that we estimate to run more than 1,000 times.

The experiments depicted in Table 3.5 show that although our method is neither sound or complete, it is capable of precise approximation of the actual trip count of interval loops (approximately 90% of the loops in the JavaScript benchmarks where correcly predicted).

Figure 3.9 shows the speedup that we obtain using our trip count predictor to perform earlier invocation of the JIT compiler.

Each number is the average of 100 runs. We call the baseline compiler immediately once we predict that a loop will iterate 10 times, and we call IonMonkey immediately once we predict that a loop will iterate 1,000 times. Figure 3.9, reveals that our technique has been able to speed up some benchmarks by a substantial factor. We have also detected slowdowns in a few scripts. This negative behaviour happens in benchmarks that iterate for a very short time. In this case, early compilation is not able to pay off the cost of code generation.

All these results advocate the use of simple preprocessing for proving the termination (or counting the number of loops) of real-world programs that may include proper slicing and simple pattern-matching.

3.5 Conclusion

This chapter is a synthesis of my contributions in the area of termination. The combination of powerful static analyses with techniques coming from the compilation community has demonstrated its great potentiality in this particular domain. I believe that there is a continuum to be explored from

the kernel algorithms to their implementation and their application to specific domain benchmarks. In this work we particularly focused on the termination of numerical programs with complex control, for which scalability was already a major concern. Our contributions significantly improved both the applicability and the scalability of termination provers for this kind of programs.
The next step is now to handle complex data structures or pointer variables, which are in our current setting abstracted away by a coarse abstraction. Proving termination or functional correctness of work-list algorithms would be the next challenge. Being able to express and prove properties of pointers and array is a necessary step toward that purpose, that we will examine in the next chapter.

4 Dealing with memory

There are numerous motivations for developing static analyses to deal with memory:

- The previous static analyses on numerical variables are only valid when the variables are stored in different memory locations. If numerical variables and pointer variables coexist in a given program, we must be sure that the numerical variables locations are only accessed by their name. If they also can be accessed via their address, then we must take *aliasing* into account. This advocates for the design of precise alias analyses.
- Some crucial safety properties such as the absence of array-out-of-bound accesses need to be ensured in low level like C where there is no clever mechanism such as typing or size tracking that ensures them. In the case of the C language particularly, there is a need for analyses that are capable of ensuring such safety properties for general purpose big programs. This advocates for the design of *low cost* algorithms that can run in a reasonable amount of time inside production compilers.
- Many compiler optimisations, such as loop transformations, tiling, are very limited in practice. Indeed one particular feature of imperative programming languages remains to be handled satisfactorily by the current state-of-the-art approaches: the disambiguation of pointer intervals. This advocates for the design of *specialised* analyses tailored for some application characteristics (here, array intensive-computations kernels).
- Static analyses for array properties are still suffering from a certain lack of precision. Indeed the main difficulty is to generate universally quantified invariants with arrays, as we have to deal with an unbounded number of cells and their content at the same time. We believe that the design of such static analyses will benefit to the two communities of verification and compilation.

This chapter summarises my contributions to the design of static memory analyses, with a focus on scalability as well as expressivity and applicability.

Summary and outline Section 4.1 describes the model of programs we use to design most of the analyses of the chapter. Then the chapter presents three contributions in the domain of *sparse analyses of pointers*.
Section 4.2 exposes the design of *sparse memory region analysis* that is capable of eliminating useless array bound checks. This technique is based on our previous work on symbolic range analyses of numerical variables (section 2.5).
Section 4.3 then details two static analyses designed to improve the precision of compiler pointer disambiguation.

The last contribution of the chapter, described in Section 4.4, is a new abstraction of programs with arrays. This technique transforms a given program and a property to be proved into a formula without arrays, whose satisfiability entails the property.

4.1 Model of programs

In this chapter, we particularly focus on the design of *scalable analyses* for C programs with pointers. This led us to choose the SSA intermediate representation variant e-SSA (to be able to transfer information from tests, as we already saw in section 2.5), and a syntax/semantics which mimics the LLVM Intermediate Representation [77].
Our analyses will be described on the mini language we describe in Figure 4.1, whose semantics will not be described here (it can be found in [SMO+14] for instance). Branches and jump instructions will not be represented in our CFGs. All the instructions in this mini-language have the property to be atomic in the sens that they generate only one new information for a given analysis. This knowledge appears due to memory allocation (malloc), deallocation (free), pointer arithmetic, intersections and phi-functions. Each of these instructions defines new variables, whose names are associated with information. For instance, the instruction $p_0 = \texttt{free}(p_1)$ copies p_1 to p_0, and binds p_0 to a memory chunk of size 0. Moreover, in our language, there is no more `&x` instruction since in LLVM, when the address of a variable is taken, then this variable is no longer in SSA form, it will be represented in memory (thus it becomes a pointer).

Integer constants	::=	$\{c_1, c_2, \ldots\}$
Integer variables	::=	$\{i_1, i_2, \ldots\}$
Pointer variables	::=	$\{p_1, p_2, \ldots\}$
Instructions (I)	::=	
– Allocate memory	|	$p_0 = \texttt{malloc}(i_0)$
– Free memory	|	$p_0 = \texttt{free}(p_1)$
– Pointer plus int	|	$p_0 = p_1 + i_0$
– Pointer plus const	|	$p_0 = p_1 + c_0$
– Bound intersection	|	$p_0 = p_1 \cap [l, u]$
– Load into pointer	|	$p_0 = *p_1$
– Store from pointer	|	$*p_0 = p_1$
– ϕ-function	|	$p_0 = \phi(p_1 : \ell_1, p_2 : \ell_2)$
– Branch if not zero	|	$\texttt{bnz}(v, \ell)$
– Unconditional jump	|	$\texttt{jump}(\ell)$

Figure 4.1 – The syntax of our language of pointers.

4.2 Validation of memory accesses through symbolic analyses

The C programming language does not prevent out-of-bounds memory accesses. There exist several techniques to secure C programs; however, these methods tend to slow down these programs substantially, because they populate the binary code with runtime checks. To deal with this problem, in [SMO+14] we have designed and tested two static analyses - symbolic region and range analysis (described in Section 2.5) - which we combine to remove the majority of these guards. In addition to the analyses themselves, we bring the following other contributions :

- We describe live range splitting strategies that improve the efficiency and the precision of our analyses.
- We show how to deal with integer overflows, a phenomenon that can compromise the correctness of static algorithms that validate memory accesses.
- A complete tool chain combined with Address Sanitizer, which improves its applicability.

4.2.1 Context : Address Sanitizer

AddressSanitizer [99] is an industrial-quality tool built on top of the LLVM compiler [77]. This tool produces instrumented binaries out of C source code, to either log or prevent any out-of-bounds memory access. AddressSanitizer has a fairly large community of users, having been employed to instrument browsers such as Firefox and Chromium. This instrumentation has a cost: in general it slows down computationally intensive programs by approximately 70%. The analyses we will design in this section will enable us to remove about half of this overhead, keeping all the guarantees that AddressSanitizer provides.
Unlike Java where arrays are associated with their sizes, in C, arrays are not packed together with size information. We have to infer this size automatically. This is the object of the symbolic region analysis.

4.2.2 Symbolic Region Analysis

To solve this problem, we resort to static analysis. We have designed a novel *region analysis*, which binds each pointer to an interval of *valid offsets*. This analysis makes use of the symbolic region analysis we depicted in Section 2.5
Like the symbolic range analysis from Section 2.5.1, the region analysis also associates with each variable an abstract state given by an interval. However, here this abstract state has a very different interpretation. If we say that $W(p) = [\ell, u]$, then we mean that all the addresses between the offsets $p + \ell$ and $p + u$ are valid. Figure 4.2 clarifies the semantic of $W(p)$. The first instruction of Figure 4.2 allocates n words in memory and assigns the newly created region to pointer v_1. Thus, given a stack S, if $b_1 = S(v_1)$ is the value of v_1, then any address between $b_1 + 0$ and $b_2 + n - 1$ is valid. The second instruction increments v_1 and calls the new address v_2. Similarly, if $b_2 = S(v_2)$, then the address $b_2 - 1$ is valid and the address $b_2 - n - 2$ is also valid.
As we will state in Proposition 4, it is always safe to dereference a pointer if it includes the address zero among its valid offsets.

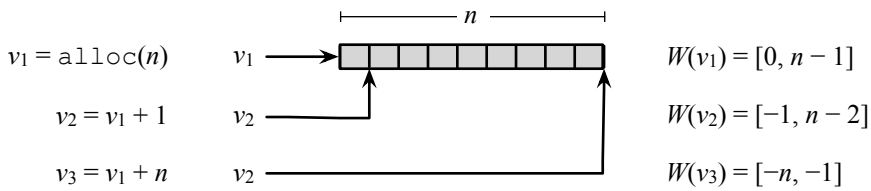


Figure 4.2 – Semantics for "valid offsets."

Splitting Required by Symbolic Region Analysis Our symbolic region analysis takes information from instructions that define pointers and instructions that free memory. We deal with the first source of information via the standard static single assignment form, as we do for the symbolic range analysis. Splitting after `free` is also simple, although this operation requires guidance from alias analysis. If we free the region bound to a pointer p at a program label ℓ, we know that after ℓ every alias of p will point to empty memory space. To make this information clear to our region analysis, we rename every alias p'_k of p to a fresh name p_k". We then initialise each of these new names with the constant zero. In this way, our region analysis will bind these variables to empty array sizes.

The SymRegion abstract domain Our analysis of regions rests on the semi-lattice SymRegion $(S^2, \sqsubseteq, \sqcap, [-\infty, +\infty], \emptyset)$, which is the inverse of the structure used in the symbolic range analysis of Section 2.5. Here, we have a meet operator "$\sqcap$", such that:

$$[a_1, a_2] \sqcap [b_1, b_2] = \begin{cases} \emptyset, & \text{if } a_2 < b_1 \text{ or } b_2 < a_1 \\ [\max(a_1, b_1), \min(a_2, b_2)], & \text{otherwise} \end{cases}$$

$$
\begin{array}{rcl}
v = c, c \in \mathbb{N} & \rightsquigarrow & W(v) = \emptyset \\
v_1 = \texttt{alloc}(v_2) & \rightsquigarrow & W(v_1) = [0, R(v_2)_{\downarrow} - 1] \\
a = b & \rightsquigarrow & W(a) = W(b) \\
\begin{array}{r} \texttt{free}(v); \\ v'_1 = 0; v'_2 = 0 \ldots \\ v_1, v_2, \ldots \in \Pi(v) \end{array} & \rightsquigarrow & W(v'_1) = W(v'_2) = \ldots = \emptyset \\
v = \phi(v_0, v_1) & \rightsquigarrow & W(v) = W(v_0) \sqcap W(v_1) \\
\begin{array}{r} v_1 = v_2 + n \\ \text{with } n \in \mathbb{N} \end{array} & \rightsquigarrow & W(v_1) = \begin{cases} \emptyset \text{ if } W(v_2) = \emptyset, \text{else} \\ [W(v_2)_{\downarrow} - n, W(v_2)_{\uparrow} - n] \end{cases} \\
\begin{array}{r} v_1 = v_2 + v_3 \\ \text{with } v_3 \text{ scalar} \end{array} & \rightsquigarrow & W(v_1) = \begin{cases} \emptyset \text{ if } W(v_2) = \emptyset \\ \text{else } [W(v_2)_{\downarrow} - R(v_3)_{\downarrow}, \\ \qquad W(v_2)_{\uparrow} - R(v_3)_{\uparrow}] \end{cases}
\end{array}
$$

Figure 4.3 – Constraint generation for the symbolic region analysis. $R(x)$ denotes the (symbolic) range of x obtained from Section 2.5

The least element of our semi-lattice is $\emptyset$, the greatest is $[-\infty, +\infty]$.
Therefore, whereas in Section 2.5 we were always expanding ranges, here we are always contracting them. In other words, the symbolic range analysis finds the largest ranges covered by variables, i.e., it is a *may* analysis. On the other hand, the symbolic region analysis finds the narrowest regions that pointers can dereference, i.e., it is a *must* analysis. In the abstract interpretation jargon, we say that we are computing *under-approximations*.

The widening we use is thus a *lower widening*[1], under the terminology of [82]. This operator is defined as follows:

$$
W_1 \nabla W_2 = \begin{cases} \emptyset & \text{if } W_{2\downarrow} > W_{1\downarrow} \text{ or } W_{2\uparrow} < W_{1\uparrow} \\ W_2 & \text{otherwise} \end{cases}
$$

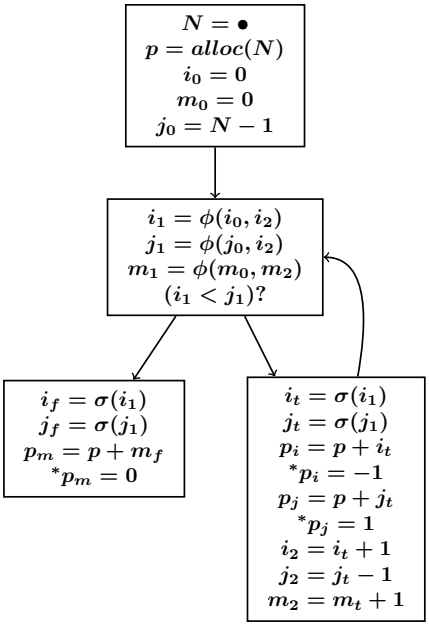
where $[a, b]_{\downarrow} = a$ and $[a, b]_{\uparrow} = b$.

An instance of region analysis is thus a set of constraints that we solve with Kleene iterations with widening points at phi-nodes points. These constraints are extracted from the program's source code, according to the rules in Figure 4.3. The figure naturally distinguish between *scalars* and *pointers*. Pointers are variables that have been initialised with the `alloc` instruction, or that are computed as functions of other pointers. Scalars are all the other variables; hence, they are always bound to the region $\emptyset$. For the last constraint, we must encode the fact that we cannot add a pointer and a variable, thus v_3 must be a scalar. Therefore, if p_1 is defined as $p_1 = p_2 + v$, then its region is given by the admissible region of p_2 added to all the possible values of v. That is why we must consider the interval range of v. Before moving on, we draw the reader's attention to the abstract semantics of the $\texttt{free}(v)$ instruction. As we explained in Section 2.5.1, this instruction leads us to rename every alias of v, and all these new names will be bound to the new region $\emptyset$.

Our region analysis uses a widening operator for ϕ-functions, to ensure that our algorithm terminates in face of pointer arithmetic.

[1] Let us point out the fact that, unlike the proposition of [82], we directly widen to $\emptyset$ when one of the bounds is not stable.

Example 6 (continuing from p. 22)**.** *If we run our region analysis on the program of Figure 4.4a, then we get that* $W(p) = [0, N-1]$, $W(p_i) = [0,1]$, $W(p_j) = [-1,0]$, *and* $W(p_m) = [0, -\infty] = \emptyset$.



(a) Our running example

```
1  unsigned N = read(); // N0>=0
2  int* p = alloc(N);
3  int i = 0;
4  int m = 0;
5  int j = N-1;
6  while (i<j){
7    if (0 <= i || i <= N-1)
8      p[i] = -1;
9    else
10     perror("array out-of-bound access!");
11   if (0 <= j || j <= N-1)
12     p[j] = 1;
13   else
14     perror("array out-of-bound access!");
15   i++;
16   j--;
17   m++;
18 }
19 if (0 <= m || m <= N-1)
20   p[m] = 0;
21 else
22   perror("array out-of-bound access!");
```

(b) Useless bound checks elimination

Figure 4.4 – Array bound check elimination. Highlighted code is useless.

These ranges correctly tell us, for instance, that the largest (safely) addressable offset from address p is $N-1$. *The ranges that we find for the program in Figure 4.4a let us remove bound checks for the memory accesses at lines 7 and 11. The region of* p_i *tells us that* $p_i = p[i]$ *and* $p_i - 1$ *are safe addresses. The region of* p_j *indicates that* $p_j = p[j]$ *and* $p_j + 1$ *are also safe addresses. On the other hand, the range of* p_m *tells us that it is not safe to dereference this pointer without a bound check. In this case, we have a false-positive, because we conservatively assume that a memory access is unsafe.*

Correctness of the Symbolic Region Analysis. Proposition 4 states the key property of our symbolic region analysis, which comes from the fact we follow the Abstract Interpretation framework. We have defined the proposition for loads, but it is also true for instructions such as $*v_1 = v_2$, which store the contents of v_2 into the address pointed by v_1.

Proposition 4. *Let P be a program and* $p\,c \in \mathbb{N}$, *such that* $P[p\,c]$ *is* $v_2 = *v_1$. *If* $0 \in W(v_1)$, *then P cannot be stuck at* $p\,c$.

The paper contains other contributions sketched in Figure 4.5:

- The safety of the two previous analyses (Symbolic Ranges, Symbolic Regions) is guaranteed if and only if there is no buffer overflow. We designed a *slicing* algorithm to find variables and expressions that should be prevented from buffer overflows. For that purpose, we use the *Sparse Evaluation Graph* [43], and propagate data dependencies as well as control dependencies. The result of this analysis is given to an instrumentation tool to add additional overflow guards before the problematic instructions.

- We have included an optional tainted flow analysis [97] in our framework. If we are interested in preventing only buffer overflows caused by malicious inputs, then we can restrict our attention to operations that are *influenced* by *values* produced by external functions. We designed

a constraint-based static analysis to identify which memory accesses can be manipulated by malicious users, in such a way that the other accesses do not need to be guarded. More details can be found in [SMO+14].

- We have designed an interprocedural analysis to propagate "less than information" from a given procedure to all the functions it calls. After we analyse the body of a function g, we have enough information to compute a less-than relationship between variables declared in its body. We compute such relationship for every pair of variables that are used as actual parameters of other functions called in the body of g. Let $f(v_1, v_2, \ldots, v_n)$ be an instruction, within the body of g, that calls a certain function f with actual parameters $v_i, 1 \leq i \leq n$. We determine the less-than relation between each $v_i, v_j, 1 \leq i, j \leq n$. Additionally, we also determine minimum lower bounds to all the scalar variables v_i.

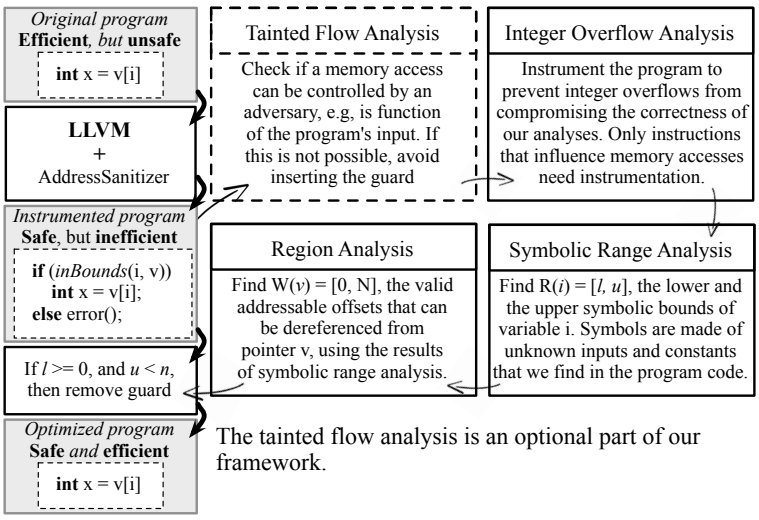


Figure 4.5 – Overview of our pipeline of static analyses.

4.2.3 Validation

Experimental setting We run our experiments in a twelve-core Intel(R) Xeon(R) CPU E5-2620 at 2.00GHz, with 15,360KB of cache, and 16GB of RAM. Neither our compiler, nor our benchmarks, run in parallel. The GreenArrays toolchain is tested over the integer programs available in the SPEC CPU 2006 benchmark suite. The paper [SMO+14] reports the experiments, that show that:

- our approach is practical - all our analyses run in acceptable time;
- it is effective - we can eliminate a reasonable number of bound checks, hence speeding up safe code and reducing its energy consumption; and
- it is competitive with state-of-the-art approaches to bound check elimination.

I only report a few of them here.

Analyses Numbers and Run Times In section 2.5, Figure 2.9, we already show the comparison of the runtime of our symbolic range analysis with existing other techniques.
The overflow analyses cause the insertion of an overflow test on 77% of all the arithmetic instructions, which represent themselves 3.5% of the total of the instructions of the LLVM bytecode.
Figure 4.6 depicts the relative times (in seconds) of all the static analyses of the toolchain for some of the most representative programs of the SPEC benchmark. The less-than tests represent the overhead for our analysis to be interprocedural, the Figure shows here clearly a possibility of improve-

ment. However, the total analysis time shows that our toolchain scales well and the total analysis time is compatible with their use inside production compilers.

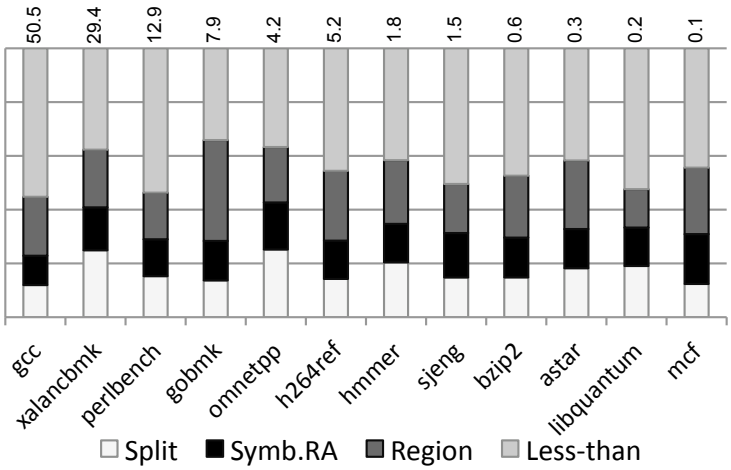


Figure 4.6 – Time taken by static analyses. **Split**: live range splitting and Overflow instrumentation. **Symb.RA**: Symbolic range analysis. **Region**: Region analysis. **Less-than**: Less-than test.

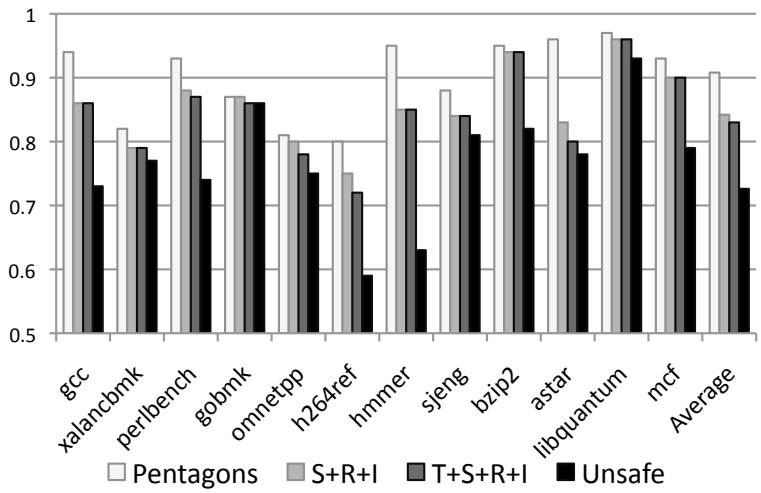


Figure 4.7 – Impact of our analyses: run time of program after bound check elimination, compared to run time of program with all the bound checks. The shorter the bar, the faster the program. **Pentagon**: Logozzo *et al* [79]. **T**: Tainted flow analysis. **I**: Integer overflow analysis. **S**: Symbolic range analysis. **R**: Region analysis. *Average speedup*: Pentagons = 9.1%, S+R+I = 15.8%, T+S+R+I=17.0%, Unsafe = 26.4%.

Impact of the analysis Figure 4.7 shows how much we speed up the binaries produced by AddressSanitizer with our analyses. On average, we achieve a speedup of 17%. This is less than the 48% of bound checks that we eliminate, but these instructions account for only a small part of the entire execution cost. The compulsory cost of a guard consists of two loads, two comparisons, and two branches. By eliminating these guards, we improve also the quality of the code produced by the register allocator, as we increase the average size of basic blocks. We are still 15.2% slower than unsafe programs, i.e., programs without runtime bound checks.

4.3 Static pointer analyses for optimising compilers

Pointer analysis is one of the most fundamental compiler technologies. This analysis lets the compiler distinguish one memory location from others; hence, it provides the necessary information to transform code that manipulates memory. Given this importance, it comes as no surprise that pointer analysis has been one of the most researched topics within the field of compiler construction [68]. This research has contributed to make the present algorithms more precise [63, 110], and faster [64, 100]. Pointer and heap analyses have also been designed for other imperative languages like Java, where the size of allocated regions are statically known, but where some optimizations can be done if an analysis can precisely capture the relationship between method calls [107] [2]. Nevertheless, one particular feature of imperative programming languages remains to be handled satisfactorily by the current state-of-the-art approaches: the disambiguation of pointer intervals: state-of-the-art pointer analyses often fail to disambiguate regions addressed from a common base pointer via different offsets, as explained by Yong and Horwitz [109]. Therefore, we claim that, to reach their full potential, compilers need to be provided with more effective alias analyses.

Example 11 (Running example for range-based pointer disambiguation)**.** *Figure 4.8 shows a pattern typically found in distributed systems implemented in C. Messages are represented as arrays of bytes. In this particular example, messages have two parts: an identifier, which is stored in the beginning of the array, and a payload, which is stored right after. This behaviour is depicted in Figure 4.9.*
The loops in lines 3-6 and 7-10 fill up each of these parts with data If a compiler can prove that the stores at lines 4 and 8 are always independent, then it can perform optimisations that would not be possible otherwise. For instance, it can parallelize the loops, or switch them, or merge them into a single body.
No alias analysis currently available in either GCC or LLVM is able to disambiguate the stores at lines 4 and 8. These analyses are limited because they do not contain range information.

```
1  void prepare(char* p, int N, char* m) {
     char *i, *e, *f;
3    for (i = p, e = p + N; i < e; i += 2) {
       *i = 0;
5      *(i + 1) = 0xFF;
     }
7    for (f = e + strlen(m); i < f; i++) {
       *i = *m;
9      m++;
     }
11 }

13 int main(int argc, char** argv) {
     int Z = atoi(argv[1]);
15   char* b = (char*)malloc(Z);
     char* s = (char*)malloc(strlen(argv[2])
         );
     strcpy(s, argv[2]);
17   prepare(b, Z, s);
     //...
19 return 0;
   }
```

Figure 4.8 – Example of program that builds up messages as sequences of serialised bytes. We are interested in disambiguating the locations accessed at lines 4 and 8.

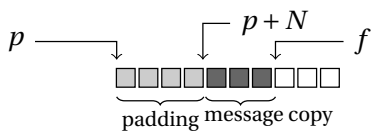


Figure 4.9 – Behaviour of the `prepare` procedure.

Example 12 (Running example for "less-than" based pointer disambiguation)**.** *To motivate the need for a new points-to analysis that is capable of deriving "less than" information, we show its application*

[2] A nice comparison of memory models used for analysing variants of C and Java can be found in the paper [101]

on programs in Figure 4.10 [3]

```
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;
  }
}
```

Figure 4.10 – A typical challenge for pointer disambiguation techniques

The figure displays a C implementation of the well known partition algorithm. We know that memory positions `v[i]` *and* `v[j]` *can never alias within the same loop iteration, since* `i` *and* `j` *are never equal. However, traditional points-to analyses cannot prove this fact.*

Typical implementations of alias analyses, built on top of the work of Andersen [6] or Steensgaard [102], can distinguish pointers that dereference different memory blocks; however, they do not say much about references ranging on the same array. In this work, we propose two new analyses based on the computation of *offset range information* (CGO'16, Section 4.3.1) or *"less than" information* (CGO'17, Section 4.3.2). These two analyses share the common "Single information" strategy.

Pointer disambiguation as queries For our pointer analyses to be useful for client analyses such as tool fusion, code motion, ..., the analyses should be split into:

- An analysis phase which computes abstract values for all variables of the program. We want this analysis phase to be scalable and precise.
- A query interface for other compiler phases. This interface should be simple: given a pair of pointers, it should answer "they do not alias" or "they may alias." We want this query to be as fast as possible, so that the client analyses can make intensive calls to it.

4.3.1 CGO 16 : Symbolic range of pointers

The contribution of this work is the (scalable) combination of pointer analysis with range analysis on the symbolic interval lattice SymBoxes (described in Section 2.5) in the form of a constraint systems whose solution is a set of abstract states of the form $\mathsf{loc}_i + R_{ij}$ for each pointer p_j.
We use two different strategies to disambiguate pointers: the global and the local test. Our global pointer analysis goes over the entire code of the program, associating variables that have the pointer type with elements of an abstract domain that we will define soon. The local analysis, on the other hand, works only for small regions of the program text.

[3] The other motivating example of the [MPR+17] paper was actually not solved with the published analysis, for expressivity reasons due to the LLVM representation. We found a workaround which was one of the main motivation of the algorithms described in the journal paper [MPMQPG17].

The notion of abstract location Our analysis binds variable names to sets of *locations* and *ranges*. We denote the set of locations in a program by $\mathscr{L}oc$: $\mathscr{L}oc = \{\mathsf{loc}_0, \mathsf{loc}_1 \ldots, \mathsf{loc}_{n-1}\}$ where n is the number of allocation sites. In our representation, i.e., Figure 4.1, new locations are created by *malloc* operations.

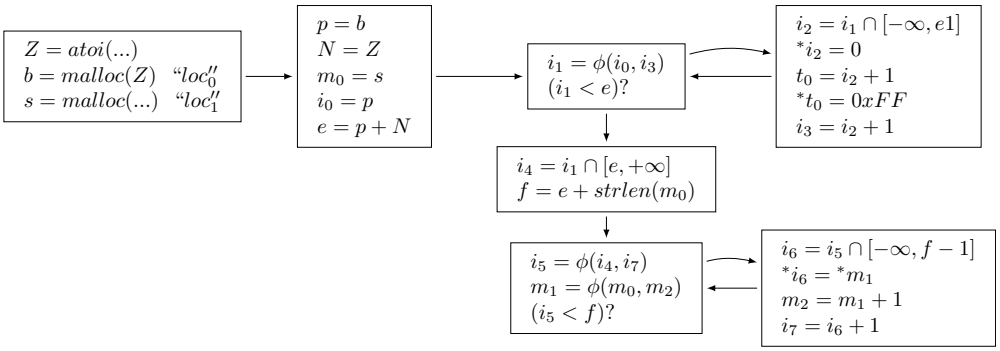


Figure 4.11 – CFG of the motivating example of Figure 4.8

Example 11 (continuing from p. 56)**.** *Figure 4.11 shows the control flow graph of the program seen in Figure 4.8. The two allocations at lines 14 and 15 are associated respectively with* loc_0 *and* loc_1.

Collecting information for pointers We already saw in Section 2.5 that we want the collected abstract information to be *invariant* in the live range of all variables. For pointers, new information appear at memory allocation (malloc), deallocation (free), pointer arithmetic, intersections and ϕ-functions. Each of these instructions defines new variables, whose names are associated with information. For instance, the instruction $p_0 = \texttt{free}(p_1)$ copies p_1 to p_0, and binds p_0 to a memory chunk of size 0.

MemLocs , an Abstract Domain of Pointer Locations. We associate pointers with tuples of size n: $(\mathsf{SymBoxes} \uplus \bot)^n$; n being the number of program sites where memory is allocated (the cardinal of $\mathscr{L}oc$) and $\uplus$ is the disjoint union.
Let $@(\mathsf{loc}_i)$ denotes the actual address value returned by the ith malloc of the program. By construction, all actual addresses are supposed to be offsets of a given $@(\mathsf{loc}_i)$. The abstract value $\mathrm{GR}(p) = (p_0, \ldots, p_{n-1})$ represents (an abstract version) of the set of memory locations that pointer variable p can address throughout the execution of a program:

Definition 8 (Abstraction)**.** *A set of actual addresses,* $S = \{s \mid \exists i \in \mathbb{N}, d \in \mathbb{N}, s = @(loc_i) + d\}$ *is abstracted by* $\alpha(S) = (p_0, p_1 \ldots, p_{n-1})$ *where :*

- $p_i = \bot$ *if there is no address in S which is an offset of* $@(loc_i)$
- $p_i = \alpha_{SymBoxes}(\{d \in \mathbb{Z} \mid s = @(loc_i) + d \in S\})$, *otherwise. The offsets from a given pointer are abstracted all-together in the SymBoxes lattice.*

The goal of our Global Range (GR)analysis is to compute such an abstract value for each pointer of the program. Some elements in a tuple $\mathrm{GR}(p)$ are bound to the undefined location, e.g., $\bot$. These elements are not interesting to us, as they do not encode any useful information. Thus, to avoid keeping track of them, we rely on the concept of *support*, which we state in Definition 9.

Definition 9 (Support)**.** *We denote by* $supp_{GR}(p)$ *the set of indexes for which* p_i *is not* $\bot$ *:*

$$supp_{GR}(p) = \{i \mid p_i \neq \bot\}.$$

For sake of readability, let us denote for instance $\mathrm{GR}(p) = (\bot, [l_1, u_1], \bot, [l_3, u_3], \bot)$, by the set $\mathrm{GR}(p) = \{\mathsf{loc}_1 + [l_1, u_1], \mathsf{loc}_3 + [l_3, u_3]\}$. In the concrete world, this notation will mean that pointer p can address any memory location from $@(\mathsf{loc}_1) + l_1$ to $@(\mathsf{loc}_1) + u_1$, and from $@(\mathsf{loc}_3) + l_3$ to $@(\mathsf{loc}_3) + u_3$.
For instance, consider that $l_1 = 3$, $u_1 = 5$, $l_3 = 3$ and $u_3 = 8$. $\mathrm{GR}(p) = \{\mathsf{loc}_1 + [3,5], \mathsf{loc}_3 + [3,8]\}$ is then depicted in Figure 4.12.

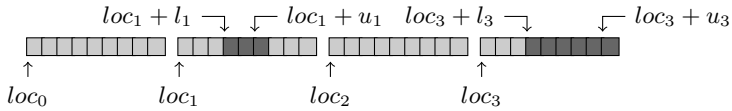


Figure 4.12 – The concrete semantics of $\mathrm{GR}(p) = \{\mathsf{loc}_1 + [3,5], \mathsf{loc}_3 + [3,8]\}$. Dark grey cells denote possible (concrete) values of p.

Now for the abstract operations: $(\bot, \ldots, \bot)$ is the least element of our lattice, and $([-\infty, \infty], \ldots, [-\infty, \infty])$ the greatest one.
Given the two abstract values $\mathrm{GR}(p^1) = (p^1_0, \ldots p^1_{n-1})$ and $\mathrm{GR}(p^2) = (p^2_0, \ldots p^2_{n-1})$, the union $\mathrm{GR}(p^1) \sqcup \mathrm{GR}(p^2)$ is the tuple $(q_0, \ldots, q_{n-1})$ where:

$$q_i = \begin{cases} \bot & \text{if } p^1_i = p^2_i = \bot \\ p_1 \sqcup p_2 & \text{else} \end{cases}$$

and $\mathrm{GR}(p^1) \sqsubseteq \mathrm{GR}(p^2)$ if an only if all involved (symbolic) intervals of p^1 are included in the ones of p^2: $\forall i \in [0..(n-1)], p^1_i \sqsubseteq p^2_i$ (considering $\bot \sqsubseteq R$ and $\bot \sqcup R = R$ for all non-empty intervals R).
We how have to define the widening operator for our abstract domain:

Definition 10. *Given $GR(p)$ and $GR(p')$ with $GR(p) \sqsubseteq GR(p')$, we define the widening operator:*

$$GR(p) \nabla GR(p') = (p_0 \nabla p'_0, \ldots, p_{n-1} \nabla p'_{n-1}),$$

where ∇ denotes the widening on SymBoxes , extended with $\bot \nabla \bot = \bot$ and $\bot \nabla [l, u] = [l, u]$.

We call the new abstract domain we have just defined MemLocs .

Example 11 (continuing from p. 56)**.** *For the example depicted in Figure 4.11 where we only have two malloc sites denoted by loc_0 and loc_1, we will obtain the following results: $GR(p) = GR(b) = \{loc_0 + [0,0]\}$, $GR(m_0) = GR(s) = \{loc_1 + [0,0]\}$, $GR(e) = loc_0 + [N, N]$, $GR(m_1) = loc_1 + [1, +\infty]$, $GR(i_7) = loc_0 + [N + strlen(m_0), N + strlen(m_0) + 1]$.*

Constraint generation for the MemLocs abstract domain (GR) The abstract semantics of each instruction in our core language is given by Figure 4.13. Figure 4.13 defines a system of equations whose fixed point gives us an approximation on the locations that each pointer may dereference.

Definition 11 (Concretization)**.** *Given $GR(p)$ an abstract value (a set of "abstract addresses for p"), denoted by $GR(p) = (p_0, \ldots, p_{n-1})$, we define its concretization as follows:*

$$\gamma(GR(p)) = \bigcup_{i \in supp_{GR}(p)} \{@(loc_i) + o, o \in p_i\}$$

The concretization function of this abstract value is thus a set of (concrete) addresses, obtained by shifting a set of base addresses by a certain value in SymBoxes .
Without any surprise, (α, γ) is a Galois connection. We solve the constraint system by Kleene iteration over the system of constraints, with one step of descending iteration.

$$j : p = \text{ malloc } (v) \text{ with } v \text{ scalar} \Rightarrow \mathrm{GR}(p) = (\bot, \ldots, \underbrace{[0,0]}_{j^{th} component}, \ldots)$$

$$p = \text{ free } (v) \text{ with } v \text{ scalar} \Rightarrow \mathrm{GR}(p) = (\bot, \ldots, \bot)$$

$$v = v_1 \Rightarrow \mathrm{GR}(v) = \mathrm{GR}(v_1)$$

$$q = p + c \text{ with } c \text{ scalar variable} \Rightarrow \begin{cases} \mathrm{GR}(q) = (q_0, \ldots, q_{n-1}) \text{ with} \\ q_i = \begin{cases} \bot \text{ if } p_i = \bot \\ p_i + R(c) \text{ else} \end{cases} \end{cases}$$

$$q = \phi(p^1, p^2) \Rightarrow \mathrm{GR}(q) = \mathrm{GR}(p^1) \sqcup \mathrm{GR}(p^2)$$

$$q = p^1 \cap [-\infty, p^2] \Rightarrow \begin{cases} \mathrm{GR}(q) = (q_0, \ldots, q_{n-1}) \text{ with} \\ q_i = \begin{cases} \bot \text{ if } (p_i^1 = \bot \text{ or } p_i^2 = \bot) \\ p_i^1 \sqcap [-\infty, p_{i\uparrow}^2] \text{ else} \end{cases} \end{cases}$$

$$q = p^1 \cap [p^2, +\infty] \Rightarrow \begin{cases} \mathrm{GR}(q) = (q_0, \ldots, q_{n-1}) \text{ with} \\ q_i = \begin{cases} \bot \text{ if } (p_i^1 = \bot \text{ or } p_i^2 = \bot) \\ p_i^1 \sqcap [p_{i\downarrow}^2, +\infty] \text{ else} \end{cases} \end{cases}$$

$$q = *p \Rightarrow \mathrm{GR}(q) = ([-\infty, \infty], \ldots, [-\infty, \infty])$$

$$*q = p \Rightarrow \text{Nothing}$$

Figure 4.13 – Constraint generation for GR with $\mathrm{GR}(p) = (p_0, \ldots, p_{n-1})$ given p in the right hand side of rules

Answering GR Queries Our queries are based on the following result, that is an immediate consequence of the fact our analysis is an abstract interpretation:

Proposition 5 (Correctness)**.** *Let p and p' be two pointers in a given program then :*

$$\begin{array}{ll} \textit{if} & supp_{GR}(p) \cap supp_{GR}(p') = \emptyset \\ \textit{or} & \forall i \in supp_{GR}(p) \cap supp_{GR}(p'), p_i \sqcap p'_i = \emptyset \end{array}$$

then $\gamma(GR(p)) \cap \gamma(GR(p')) = \emptyset$.

In other words, if the abstract values of two different pointers of the program have a null intersection, then the two *concrete pointers* do not alias. This result is directly implied by the abstract interpretation framework. Thanks to this result, we implement the query $Q_{\mathrm{GR}}(p, p')$ as:

- If $\mathrm{GR}(p)$ and $\mathrm{GR}(p')$ have an empty intersection, then "they do not alias."
- Else "they may alias."

Example 11 (continuing from p. 56)**.** *In the example, since $GR(i_7) = loc_0 + [N + strlen(m_0), N + strlen(m_0) + 1]$ and $GR(i_2) = loc_0 + [0, N-1]$, i_7 and i_2 never alias.*

About local analysis The global pointer analysis is not path sensitive. As a consequence, this analysis cannot, for instance, distinguish the effects of different iterations of a loop upon the actual value of a pointer, or the effects of different branches of a conditional test on that very pointer. To solve this issue, we propose to combine the global analysis with a local analysis (called LR) where we create information at merge nodes (a new abstract location is created). The detail of this analysis can be found in [PMB+16].

4.3.2 CGO 17 : Pointer Disambiguation via Strict Inequalities

The contribution of the paper [MPR+17] is the (scalable) combination of pointer analysis with a "Pentagon-like" analysis which is capable to derive "less than" information for numerical variables as well as pointer variables. This analysis also relies on the symbolic interval lattice SymBoxes described in Section 2.5. We report the main ideas of the analysis, the complete formalisation can be found in the paper.

An additional splitting strategy The objective of the analysis is to derive less-than information (Less Than Analysis (CGO'17) (LT)) a program into SSA form, for numerical variables as well as pointer variables.
To derive such information that still have to *stay invariant on live ranges*, we propose a new splitting strategy depicted in Figure 4.14. The idea is to derive from an instruction $x_1 = x_2 + n$ and a pre-computed range for n that implies that $n < 0$, the information $x_1 < x_2$. However, this information is only valid after the instruction, thus we invent a new name for x_2, namely x_3 (Figure 4.14b). We let $x_1 = x_2 + n \parallel \langle x_3 = x_2 \rangle$ denote a composition of two statements, $x_1 = x_2 + n$ and $x_3 = x_2$. The second instruction splits the live range of x_2. Both statements happen in parallel. Thus, $x_1 = x_2 + n \parallel \langle x_3 = x_2 \rangle$ does not represent an actual assembly instruction; it is only used for notation convenience. Similarly, when transforming conditional tests, we let $\langle x_{1t} = x_1, x_{2t} = x_2 \rangle$ denote two copies that happen in parallel: $x_{1t} = x_1$, and $x_{2t} = x_2$. Whenever there is no risk of ambiguity, we write simply $\langle x_{1t}, x_{2t} \rangle$. As usual, parallel copies and ϕ-functions are removed before code generation, after the analyses that require them have already run.

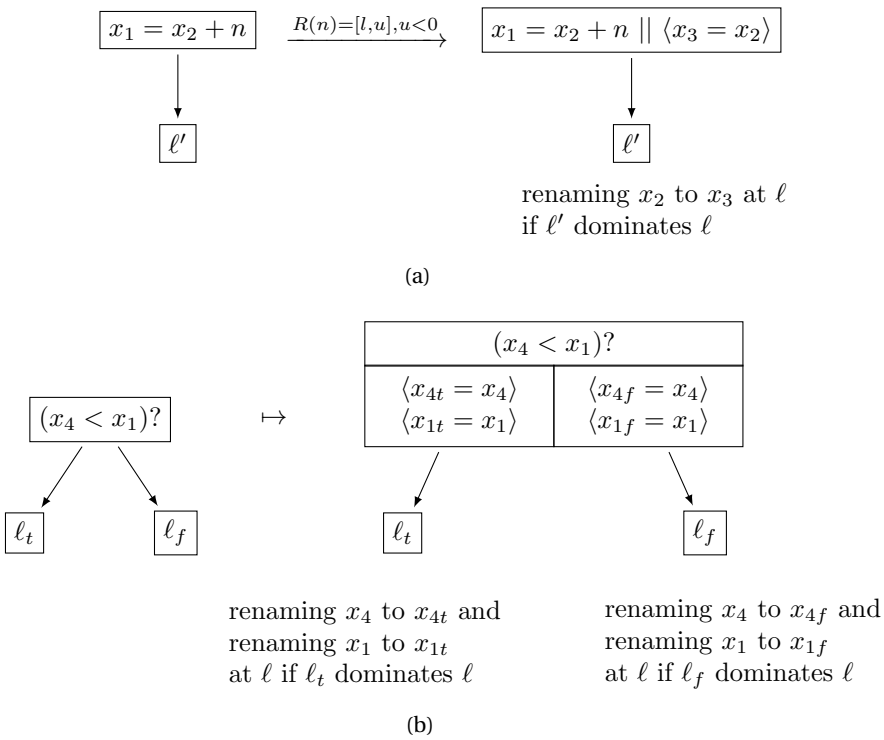


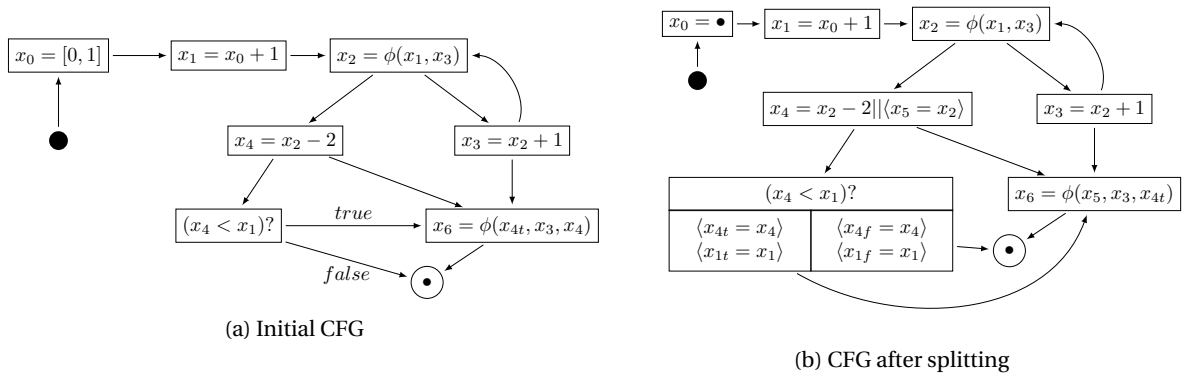
Figure 4.14 – Splitting strategy for the less-than analysis.

Example 13 (Splitting strategy)**.** *Figure 4.15 illustrates the splitting strategy on a toy example.*

Constraint generation, solving Once we have a suitable program representation, we use the rules in Figure 4.16 to generate constraints. These constraints determine the less-than set of variables. Constraint generation is $O(|\mathcal{V}|)$, where $\mathcal{V}$ is the set of variables in the target program.

Constraints are solved via a worklist algorithm. We initialise $\mathsf{LT}(x)$ to $\mathcal{V}$, for every variable x. During the resolution process, elements are removed from each LT, until a fixed point is achieved. We prove that this process terminates and its correction.

Example 13 (continuing from p. 61)**.** *The rules in Figure 4.16 produce the following constraints for the program in Figure 4.15a* $LT(x_0) = \emptyset$, $LT(x_1) = \{x_0\} \cup LT(x_0)$, $LT(x_2) = LT(x_1) \cap LT(x_3)$, $LT(x_3) =$



(a) Initial CFG

(b) CFG after splitting

Figure 4.15 – Illustration of the splitting strategy on a toy example

$$x = \bullet \rightsquigarrow_1 \mathsf{LT}(x) = \emptyset$$

$$x_1 = x_2 + n \rightsquigarrow_2 \mathsf{LT}(x_1) = \{x_2\} \cup \mathsf{LT}(x_2)$$

$$x_1 = x_2 - n \parallel \langle x_3 = x_2 \rangle \rightsquigarrow_3 \begin{cases} \mathsf{LT}(x_3) = \{x_1\} \cup \mathsf{LT}(x_2) \\ \mathsf{LT}(x_1) = \emptyset \end{cases}$$

$$x = \phi(x_1, \ldots, x_n) \rightsquigarrow_4 \mathsf{LT}(x) = \mathsf{LT}(x_1) \cap \ldots \cap \mathsf{LT}(x_n)$$

$$(x_1 < x_2)? \begin{cases} \ell_t : \langle x_{1t}, x_{2t} \rangle \\ \ell_f : \langle x_{1f}, x_{2f} \rangle \end{cases} \rightsquigarrow_5 \begin{cases} \mathsf{LT}(x_{2t}) = \\ \quad \{x_{1t}\} \cup \mathsf{LT}(x_1) \\ \mathsf{LT}(x_{1f}) = \mathsf{LT}(x_2) \\ \mathsf{LT}(x_{1t}) = \mathsf{LT}(x_1) \\ \mathsf{LT}(x_{2f}) = \mathsf{LT}(x_2) \end{cases}$$

Figure 4.16 – Constraint generation rules. n is a variable such that $R(n) = [l, u], l > 0$.

$\{x_2\} \cup LT(x_2)$, $LT(x_4) = \emptyset$, $LT(x_5) = \{x_4\} \cup LT(x_3)$, $LT(x_{1t}) = \{x_{4t}\} \cap LT(x_{4t})$, $LT(x_{1f}) = LT(x_1)$, $LT(x_{4f}) = LT(x_1)$, $LT(x_{4t}) = LT(x_4)$, $LT(x_6) = LT(x_3) \cap LT(x_{4f}) \cap LT(x_5)$.
To solve them, we initialise every LT set to $\{x_0, x_1, x_2, x_3, x_4, x_6, x_{1f}, x_{1t}, x_{4f}, x_{4t}\}$*, i.e., the set of program variables.*
Chaotic iterations on those constraints achieves the following fixed point: $LT(x_0) = LT(x_4) = LT(x_{4t}) = \emptyset$*;* $LT(x_1) = LT(x_2) = LT(x_{4f}) = LT(x_{1f}) = LT(x_6) = \{x_0\}$*;* $LT(x_3) = \{x_0, x_2\}$*;* $LT(x_5) = \{x_0, x_4\}$*; and* $LT(x_{1t}) = \{x_{4t}\}$.

Answering Queries The less-than check that we have discussed in this section lets us compare pointers directly, if they are bound to a less-than relation, or indirectly, if they are derived from a common base. This observation gives the query $Q_{LT}(p_1, p_2)$:

- If $p_1 \in \mathsf{LT}(p_2)$ or $p_2 \in \mathsf{LT}(p_1)$, then "they do not alias."
- Else if $p_1 = p + x_1$ and $p_2 = p + x_2$ with the same base pointer p and $x_1 \in \mathsf{LT}(x_2)$ or $x_2 \in \mathsf{LT}(x_1)$, then "they do not alias."
- Else "they may alias."

Example 12 (continuing from p. 56)**.** *On the example of Figure 4.10, we are able to disambiguate the two accesses to* $v + i$ *and* $v + j$ *of the* `partition` *procedure, since we are able to infer* $j \in LT(i)$ *at lines 9-11.*

The C standard refers to arithmetic types and pointer types collectively as scalar types [70]{§6.2.5.21}. Notice that the less-than analysis that we have discussed thus far works seamlessly for scalars; thus, it also builds relations between pointers. For instance, the common idiom "for(int* pi = p; pi < pe; pi++);" gives us that pi < pe inside the loop.

4.3.3 Experimental Evaluation

Implementation We have implemented our range analysis in the LLVM compiler, version 3.5. An interprocedural version of the "less-than" analysis has been implemented in LLVM version 3.7.
In this section, we show a few numbers that we have obtained with these two implementations. All our experiments have been run in a standard machine, Intel i7-4770K, with 8GB of memory, running a Linux Desktop system.
Our goal with these experiments is to show:

- that our alias analyses are more precise than other alternatives of practical runtime;
- that it scales up to large programs.

It is worth underling that these two evaluations have been reproduced by an independent committee "Artefact Evaluation Committee" [4] at CGO'16 and CGO'17.

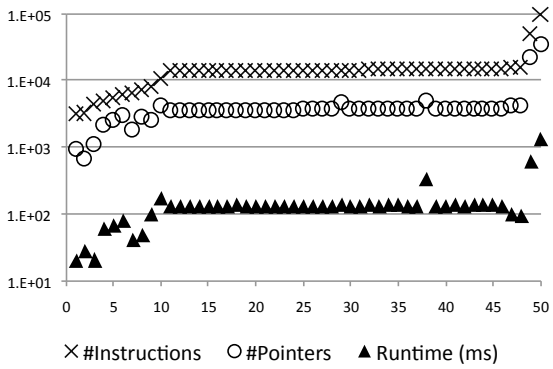
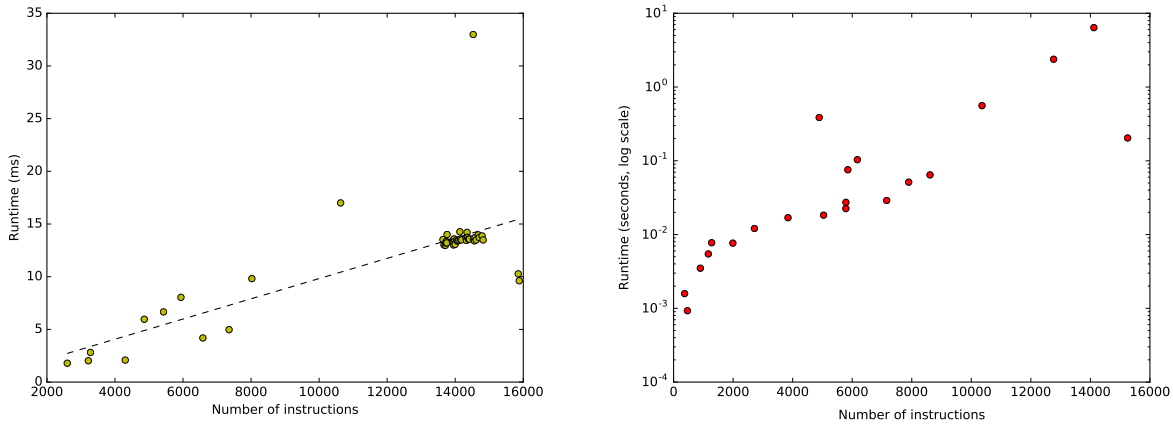


Figure 4.17 – Runtime of CGO'16 analysis for the 50 largest benchmarks in the LLVM test suite. Each point on the X-axis represents a different benchmark. Benchmarks are ordered by size. This experiment took less than 10 seconds.

Runtime The chart in Figure 4.17 shows how our CGO 16 analysis scales when applied on programs of different sizes. We have used the 50 largest programs in the LLVM benchmark suite. These programs gave us a total of 800,720 instructions in the LLVM intermediate representation, and a total of 241,658 different pointer variables. We analysed all these 50 programs in 8.36 seconds. We can – effectively – analyse 100,000 instructions in about one second. In this case, we are counting only the time to map variables to values in SymBoxes . We do not count the time to query each pair of pointers, because usually compiler optimisations perform these queries selectively, for instance, only for pairs of pointers within a loop. Also, we do not count the time to run the out-of-the-box implementation of range analysis. The chart provides strong visual indication of the linear behaviour of our algorithm, the linear correlation between time and number of instructions being 0.98.

Charts in Figure 4.18 compares the two approaches for middle-sized benchmarks (from 2000 to 15000 instructions). While the CGO'17 implementation is faster in average for these benchmarks, figure 4.18b demonstrates an exponential behaviour. We believe that some of our implementation choices could clearly be improved in further versions of this implementation. However, for middle-sized benchmarks, the solving time is already less than 10 ms, which is already a satisfactory result.

[4] See http://ctuning.org/ae/cgo2016.html and http://ctuning.org/ae/cgo2017.html



(a) Offset range information (CGO'16)

(b) Less than information(CGO'17)

Figure 4.18 – Runtime of our CGO'16 (classic scale) and CGO'17 (log scale) analyses. The experiments were launched on 20 middle-sized benchmarks from LLVM `Multisources`. This experiment took less than 20 seconds for CGO16 and CGO17

Pointer analyses evaluation in LLVM To measure the precision of our methods, we compare them against the techniques already in place in the LLVM compiler. Our metric is the percentage of *pointer queries* disambiguated. To generate queries, we resort to LLVM's `aa-eval` pass, which tries to disambiguate every pair of pointers in the program. Our main competitor will be LLVM's basic disambiguation technique, the `basic-aa` algorithm. This analysis uses several heuristics to disambiguate pointers, relying mostly on the fact that pointers derived from different allocation sites cannot alias in well-formed programs [5]. We also compare the results obtained by the the "scalar-evolution-based" (SCEV) alias analysis. This analysis tries to infer closed-form expressions to the induction variables used in loops. With this information, SCEV can track the ranges of indices which dereference array `a` within the loop. Contrary to our analysis, SCEV is only effective to disambiguate pointers accessed within loops and indexed by variables in the expected closed-form.

Precision of the CGO'16 method To evaluate the precision of the CGO'16 method, we have chosen three benchmarks that have been used in previous work that compares pointer analyses: Prolangs [95], PtrDist [111] and MallocBench [57]. Figure 4.19 reports the the results in terms of percentage of pairs of pointers that are reported not to alias compared with the total number of pointer pairs. We first notice that in general all the pointer analyses in LLVM disambiguate a relatively low number of pointers. This happens because many pointers are passed as arguments of functions, and, not knowing if these functions will be called from outside the program, the analyses must, conservatively, assume that these parameters may alias. Second, we notice that our pointer analysis is one order of magnitude more precise than the scalar-evolution based implementation available in LLVM. Finally, we notice that we are able to disambiguate more queries than the *basic* analysis. Furthermore, our results complements it in non-trivial ways. In total, we tried to disambiguate 3.093 million pairs of pointers. Our analysis found out that 1.29 million pairs reference non-overlapping regions. The *basic* analysis has been able to distinguish 953 thousand pairs. By combining these two analyses, we extended this number to 1.439 thousand pairs of pointers. SCEV could not increase this number any further.

[5] it is worth to mention that the LLVM 3.7 version contains other alias analyses, whose results we shall not use, because they have been able to resolve a very low number of queries in our experiments. Most probably there will be an increasing number of aliases analyses implemented in LLVM in a soon future, making these lines obsolete.

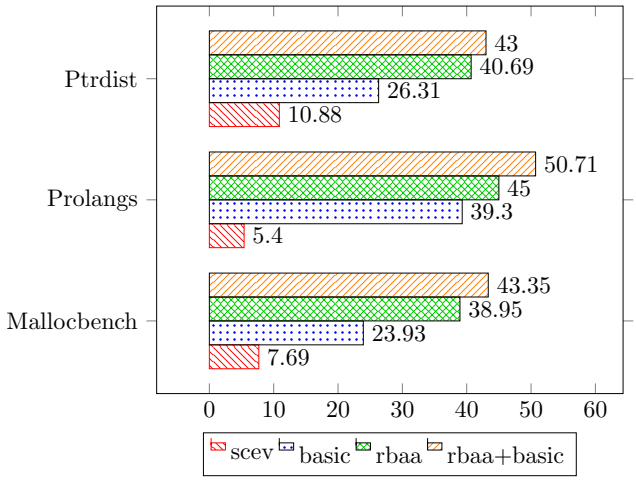


Figure 4.19 – Precision of the offset range analysis compared and combined to other LLVM analyses

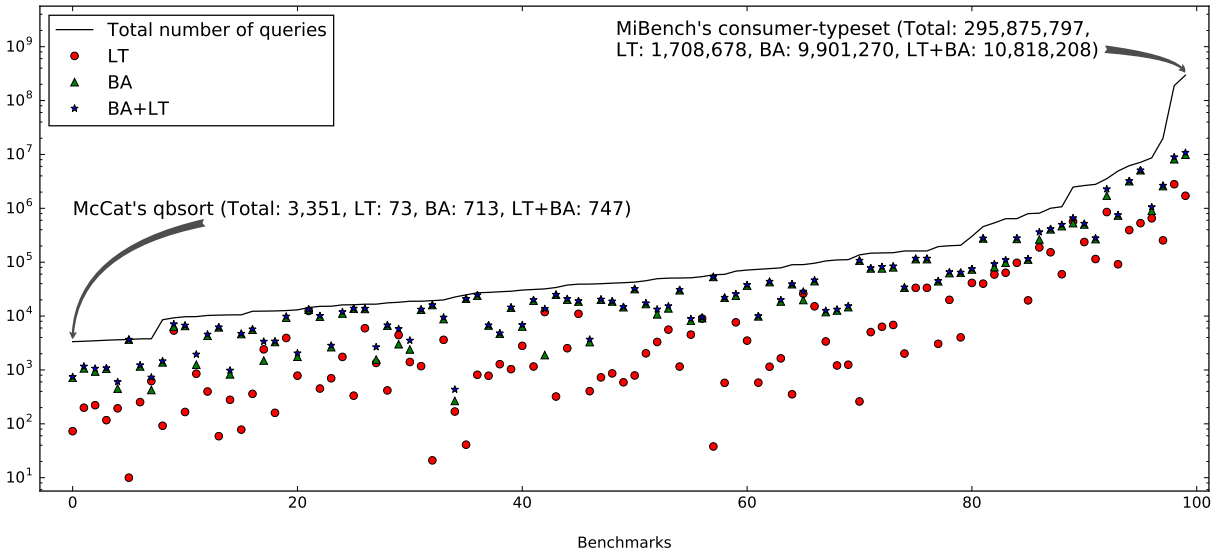


Figure 4.20 – Effectiveness of our alias analysis (**LT**), when compared to LLVM's basic alias analysis on the 100 largest benchmarks in the LLVM test suite. Each point in the X-axis represents one benchmark. The Y-axis represents total number of queries (one query per pair of pointers), and number of queries in which each algorithm got a "no-alias" response (the higher the better).

Precision of the CGO'17 method Figure 4.20 shows the results of the three alias analyses when applied on the 100 largest benchmarks in the LLVM test suite. Our method rarely disambiguates more pairs of pointers than **BA**. Such result is expected: most of the queries consist of pairs of pointers derived from different memory allocation sites, which **BA** disambiguates, and we do not analyse. Figure 4.21 compares our analysis (LT), the basic analysis (BA) of LLVM and a variant of Andersen's analysis (CF) Our numbers have been obtained in LLVM 3.7, whereas **CF**'s has been produced via LLVM 4.0 [6] We emphasise that both versions of this compiler produce exactly the same number of alias queries, and, more importantly, **BA** outputs exactly the same answers in both cases. This experiment reveals that there is no clear winner in this alias analysis context. **BA+LT** is more than 20% more precise than **BA+CF** in three benchmarks: `lbm`, `milc` and `gobmk`. **BA+CF**, in turn, is three times more precise in `omnetpp`. The main conclusions that we draw from this comparison are the following:

[6] The complete methodology can be found in the [MPR+17] paper.

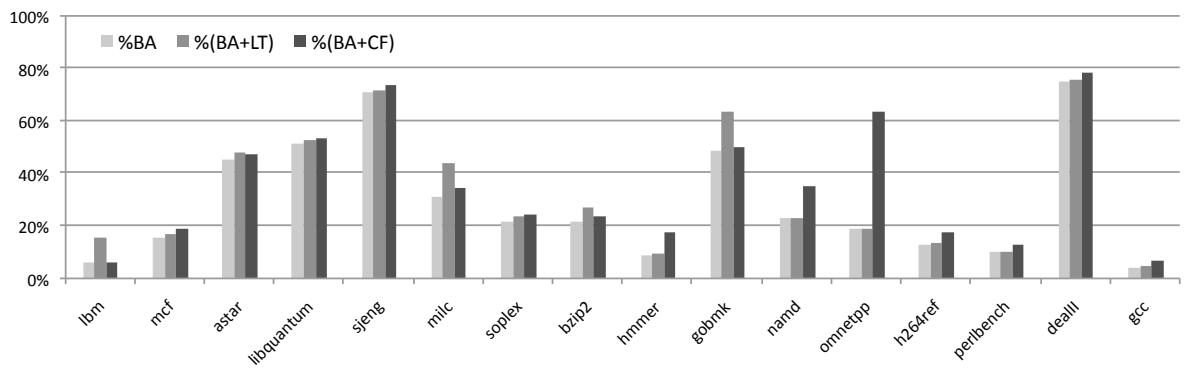


Figure 4.21 – How two different alias analysis (**LT** and **CF**) increase the capacity of LLVM's basic alias analysis (**BA**) to disambiguate pointers. The Y-axis shows the percentage of no-alias responses. The higher the bar, the better.

- These analysis are complementary.
- Mainstream compilers still miss opportunities to disambiguate alias queries.

4.3.4 Impact on further analyses and optimisations

Applicability of the CGO'17 method: program dependence graph construction. We show how our new alias analysis improves the construction of a client analysis, the Program Dependence Graph (PDG) construction, a classic data structure introduced by Ferrante *et al.* [51]. We use the implementation of PDGs available in the FlowTracker system [92], which has a distribution for LLVM 3.7. The PDG is a graph whose vertices represent program variables and memory locations, and the edges represent dependences between these entities. An instruction such as $a[i] = b$ creates a data dependence edge from b to the memory node $a[i]$. The more memory nodes the PDG contains, the more precise it is, because if two locations alias, they fall into the same node. In the absence of any alias information, the PDG contains at most one memory node; perfect alias information yields one memory node for each independent location in the program. The full methodology of these experiments can be found in the [MPR+17] paper. The results depicted in Figure 4.22 show that our analysis combined with BA clearly improves the size of the resulting PDG.

Impact on LLVM memory optimisations We also studied the impact of our analyses on further optimisations in the context of Maroua Maalej's Phd thesis. A more detailed study can be found in her manuscript. In this paragraph, we show how (a modified and extended version of) our CGO'17 analysis [MPMQPG17] does not only outperform the state-of-the-art pointer LLVM analyses in terms of the total number of pointer pairs that are proved to be non aliasing, but also the additional information is relevant for further analyses and optimisations, and in particular LICM, as we show in Example 14.

Example 14 (Loop invariant code motion (LICM))**.** *In Figure 4.23 we depict a toy example in which the underlined statement could be moved from inside the loop to its* pre-header *at line* 5 *(N is supposed to be positive). This optimisation is valid if the statement is* invariant in the loop*, and also the loop is guaranteed to execute at least once. The first condition is true since the two pointers p_2 and p_1 do not alias. An improved alias analysis thus benefits to the LICM pass.*

Figure 4.24 shows that our analysis improves the performance of the LICM optimisation by a factor of 1.77. This factor does not include the programs for which our analysis combined with the other LLVM passes reports errors.

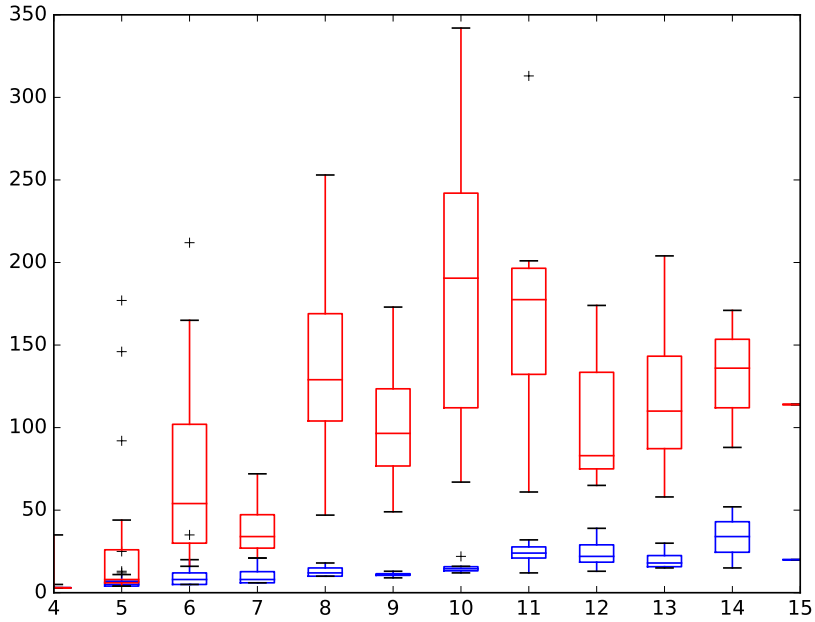


Figure 4.22 – Precision of dependence graph. All benchmarks with the same number of memory locations (X-axis) are considered together. For each set, we depict the repartition of the number of nodes of the generated Program Dependence Graphs, with Basic Alias (in blue), and Basic Alias combined with LT (in red). The more memory nodes the PDG contains, the more precise it is.

```
int* p = malloc (2*N*sizeof(int))
int *p1, *p2, *a
*p = 8; *a = 10
p1 = p + N
p2 = p + 2 × N
while p2 > p1 do
    *a = *p
    *p2 = 4
    p2- -
end while
```

Figure 4.23 – LICM hoisting load optimisation.

These experiments demonstrate that there is still a need for more precise analyses. Moreover, our experience in the understanding of the LLVM optimisation phases shows that their design should be rethought in order to benefit more from generic analyses passes like ours: the tests they perform in order to perform a given optimisation are often too restrictive.

4.4 Static analyses of programs manipulating arrays

As we saw previously , verifying safety properties of numerical programs is already a hard job, and it is even harder if the program acts upon arrays or other forms of maps. Contrarily to the preceding analyses where the properties to prove were only about *sets of addresses*, here we want to express and find *relational properties between a given (set of) address(es) and its (their) content*, which is much harder.

The main difficulty is to find a way to express and decide properties that may depend on the content

Program	#Inst	#callHoistedSunk		#loadHoistedSunk	
		O3	O3sraa	O3	O3sraa
cdecl	4859	*	*	5	6
football	14529	*	X	22	X
simulator	7698	*	*	10	16
assembler	6661	*	*	*	*
loader	2263	*	*	*	*
gnugo	5449	*	*	10	12
unix-tbl	8101	1	X	61	X
agrep	15382	4	X	53	X
fixoutput	369	*	*	1	5
compiler	3515	*	*	*	*
bison	15645	1	1	165	179
archie-client	5939	*	*	*	*
TimberWolfMC	98792	2	7	1287	1447
allroots	574	*	*	*	*
unix-smail	5435	4	4	3	3
plot2fig	3217	1	1	3	3
bc	10632	*	*	18	19
yacr2	6583	*	*	144	190
ks	1368	*	*	8	11
anagram	993	1	1	4	4
ft	1646	*	*	4	4
cfrac	7353	1	1	5	6
espresso	50751	1	1	301	398
gs	55281	*	X	20	X

Figure 4.24 – Optimisations (call or load motion) performed by LICM with O3 alone, and O3 enhanced with our analysis `sraa` (CGO'17), on the programs of the LLVM testsuite. The number of instructions is the number of LLVM internal representation instructions just after the parsing phase. A star means 0, a cross denotes that we were not able to run the analysis, due to implementation issues.

of an *unbounded* number of memory addresses, or "cells." Transposing the approaches for verifying programs operating upon Boolean and integer values (e.g. abstract interpretation, counterexample-guided abstraction refinement...), is not immediate.
In the SAS paper [MG16], together with D. Monniaux, we propose an alternative approach to ad-hoc array abstract domains. Instead, we generate an abstraction as a scalar problem and feed it to a preexisting SMT solver, with tunable precision. Our transformed problem is expressed using Horn clauses, a common format with clear and unambiguous logical semantics for verification problems.

4.4.1 Models of programs

The syntax of the (CFG of the) programs we consider is depicted in Figure 4.25. This is a simple *language* with numerical and array variables. Any statement in the program (*control flow graph*) will be: i) either an array read to a fresh variable, v=a[i]; the variables of the program are $(\boldsymbol{x}, i, v)$ where $\boldsymbol{x}$ is a vector of arbitrarily many variables; ii) either an array write, a[i]=v; (where v and i are variables) the variables of the program are $(\boldsymbol{x}, i, v)$ before and after the statement; iii) or a scalar operation, including assignments and guards over scalar variables. More complex statements can be transformed to a sequence of such statements, by introducing temporary variables if needed: for instance, $a[i] = a[j]$ is transformed into $temp = a[j]$; $a[i] = temp$.

4.4.2 Program Verification with Horn Clauses

As we already saw in Chapter 2, Section 2.1.1, the classical approach to program verification consists in computing *inductive invariants* on control-flow graphs. In this work, we propose to express these invariants to be found as *predicates* on the program variables, and the control-flow graph itself as a whole Horn formula.

Integer constants	::=	$\{c_1, c_2, \ldots\}$
Integer variables	::=	$\{x_1, x_2, \ldots\}$
Array variables	::=	$\{a_1, a_2, \ldots\}$
Instructions (I)	::=	
– Array creation	|	$new(a)$
– Array read	|	$x_1 = a[x_1]$
– Array write	|	$a[x] = c$
– Numerical assignments	|	$x = numexpr$
– Branch if not zero	|	$\texttt{bnz}(x, \ell)$
– Unconditional jump	|	$\texttt{jump}(\ell)$

Figure 4.25 – The syntax of language with arrays

Definition 12. *A Constrained Horn Clause (Constrained Horn Clause (CFC)) is a formula of the form:*

$$\bigwedge_i P_i(\boldsymbol{x}) \wedge f(\boldsymbol{x}, \boldsymbol{x}') \implies P'(\boldsymbol{x}')$$

where for all i:

- *P_i and P' are predicates.*
- *$\boldsymbol{x}, \boldsymbol{x}'$ are variables that are local to the clause.*
- *f is a formula over a given theory (here, integer arithmetic with or without arrays).*

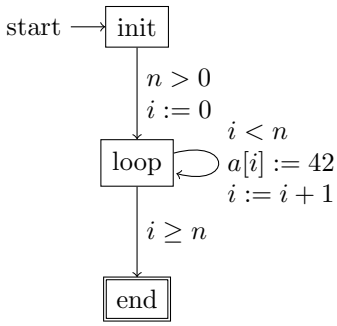
A Horn formula is a set of CHCs.

The property to be proved is also expressed in term of a formula, and the verification problems consists in checking if the program semantics implies the property. This is illustrated in example 15. Various tools can solve systems of Horn clauses (depending on the underlying theory, of course), that is, can synthesise suitable predicates $I^\sharp_\ell$, which constitute *inductive invariants*. In this article, we tried Z3[7] with the PDR fixed point solver [69], Z3 with the SPACER solver [74, 75],[8] and ELDARICA[94].[9] Since program verification is undecidable, such tools, in general, may fail to terminate, or may return "unknown."

Example 15 (Motivating example)**.** *Consider the program of Figure 4.26 where a given array of unknown size is filled with the same constant value 42. We would like to prove that this program truly fills array a[] with value 42.*

```
void array_fill1(int n, int a[n])
{
  for(int i=0; i<n; i++)
      a[i]=42;
  /* assert ∀0 ≤ k < n, a[k] = 42 */
}
```

(a) Listing for array fill



(b) Its CFG,

Figure 4.26 – The array fill example

The flat encoding into Horn clauses assigns a predicate (set of states) to each of the control nodes (Fig. 4.26b), and turns each transition into a Horn rule with variables ranging in $Arr(A, B)$, the type of

[7] https://github.com/Z3Prover hash 7f6ef0b6c0813f2e9e8f993d45722c0e5b99e152; due to various problems we preferred not to use results from later versions.

[8] https://bitbucket.org/spacer/code hash 7e1f9af01b796750d9097b331bb66b752ea0ee3c

[9] https://github.com/uuverifiers/eldarica/releases/tag/v1.1-rc

arrays of B indexed by A [76, Ch. 7]:

$$\forall n \in \mathbb{Z}\ \forall a \in Arr(\mathbb{Z},\mathbb{Z})\ \ n > 0 \implies loop(n,0,a) \tag{4.1}$$

$$\forall n,i \in \mathbb{Z}\ \forall a \in Arr(\mathbb{Z},\mathbb{Z})\ \ i < n \wedge loop(n,i,a) \implies loop(n,i+1,store(a,i,42)) \tag{4.2}$$

$$\forall n,i \in \mathbb{Z}\ \forall a \in Arr(\mathbb{Z},\mathbb{Z})\ \ i \geq n \wedge loop(n,i,a) \implies end(n,i,a) \tag{4.3}$$

$$\forall x,n,i \in \mathbb{Z}\ \forall a \in Arr(\mathbb{Z},\mathbb{Z})\ \ 0 \leq x < n \wedge end(n,i,a) \implies select(a,x) = 42 \tag{4.4}$$

where $store(a,i,v)$ is array a where the value at index i has been replaced by v and $select(a,x)$ denotes $a[x]$. Equation 4.4 encodes the property to prove. The encoding of the problem is then given to solvers: if a given solver returns "sat", it means that it has been able to synthesise invariants *for each unknown predicate, namely* init, state *and* end.
*None of the tools we have tried (*Z3, SPACER, ELDARICA*) has been able to solve this system, presumably because they cannot infer universally quantified invariants over arrays.*[10] *Indeed, here the loop invariant needed is $0 \leq i \leq n \wedge (\forall k\ 0 \leq k < i \implies a[k] = 42)$. While $0 \leq i \leq n$ is inferred by a variety of approaches, the rest is tougher.*

Most software model checkers attempt constructing invariants from *Craig interpolants* obtained from refutations [30] of the accessibility of error states in local [69] or global [81] unfoldings of the problem. However, interpolation over array properties is difficult, especially since the goal is not to provide any interpolant, but interpolants that generalise well to invariants [2, 3]. This contribution instead introduces a way to derive universally quantified invariants from the analysis of a system of Horn clauses on scalar variables (without array variables).

4.4.3 An encoding of programs with arrays

To use the power of Horn solvers, we soundly abstract problems with arrays to problems without arrays. In the Horn clauses for Ex. 15, we attached to each program point p_ℓ a predicate I_ℓ over $\mathbb{Z} \times \mathbb{Z} \times Arr(\mathbb{Z},\mathbb{Z})$ when the program variables are two integers i,n and one integer-value, integer-indexed array a. [11] In any solution of the system of clauses, if the valuation (i,n,a) is reachable at program point p_ℓ, then $I_\ell(i,n,a)$ holds. Instead, in the case of Ex. 15, we will consider a predicate $I^\sharp_\ell$ over $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$ (the array $key \to value$ has been replaced by a pair $(key, value)$) such that $I^\sharp_\ell(i,n,k,a_k)$ [12] holds for each reachable state (i,n,a) satisfying $a[k] = a_k$.
This is the same Galois connection [41] as some earlier works [84] [40, Sec. 2.1]; yet, as we shall see, our abstract transformers are more precise. For simplicity, in the sequel, we will consider that the vector of values of all variables (except arrays) of the program lies in $\mathbb{Z}^d$, the array indexes as well as the content of each cell also being of type $\mathbb{Z}$.

Definition 13. *The "one distinguished cell"* abstraction *of $I \subseteq \mathbb{Z}^d \times Arr(\mathbb{Z},\mathbb{Z})$ is $\alpha(I) = \{(\boldsymbol{x}, i, a[i]) \mid \boldsymbol{x} \in \mathbb{Z}^d, i \in \mathbb{Z}\}$. The* concretization *of $I^\sharp \subseteq \mathbb{Z}^d \times (\mathbb{Z} \times \mathbb{Z})$ is $\gamma(I^\sharp) = \{(\boldsymbol{x}, a) \mid \forall i \in \mathbb{Z}\ (\boldsymbol{x}, i, a[i]) \in I^\sharp\}$. In other words, during the abstraction, the array variable a will be replaced by a couple $(i, a_i) \in \mathbb{Z}^2$, and predicates on a will be transformed into predicates on (i, a_i).*

We depict in Figure 4.27 the abstract semantics of the statements of our mini-language. All the variables inside predicates are universally quantified.

- For scalar only statements, the abstract transformers are straightforward, they express the numerical semantics of each transition $\ell \to \ell'$ on scalar values, the array being unchanged. For instance, for a program with two variables x, y and a unique array a, a transition from control points 1 to 2 of the form $x = y + 1$ is abstracted into $I^\sharp_1(x,y,i,a_i) \wedge x' = y+1 \wedge y' = y \implies I^\sharp_2(x',y',i,a_i)$.

[10] Some of these tools can however infer some simpler array invariants.
[11] For instance, $I_{loop} = loop(n,i,a)$, $I_{end} = end(n,i,a)$.
[12] also denoted by $I^\sharp_\ell((i,n),(k,a_k))$ for sake of readability.

$$
\begin{array}{rcl}
new(a) & \rightsquigarrow & I^\sharp_\ell(\boldsymbol{x}) \implies I^\sharp_{\ell'}(\boldsymbol{x}, k, a_k) \\
x_0 = c, c \in \mathbb{N} & \rightsquigarrow & I^\sharp_\ell(\boldsymbol{x}, x_0, k, a_k) \implies I^\sharp_{\ell'}(\boldsymbol{x}, c, k, a_k) \\
x_0 = numexpr & \rightsquigarrow & I^\sharp_\ell(\boldsymbol{x}, k, a_k) \wedge R(\boldsymbol{x}, \boldsymbol{x}') \implies I^\sharp_{\ell'}(\boldsymbol{x}', k, a_k) \\
x_0 \neq x_1 \text{(test example)} & \rightsquigarrow & I^\sharp_\ell(\boldsymbol{x}, k, a_k) \wedge x_0 \neq x_1 \implies I^\sharp_{\ell'}(\boldsymbol{x}, k, a_k) \\
x_0 = a[i] \text{(read)} & \rightsquigarrow & \begin{cases} k \neq i \wedge I^\sharp_\ell\big((\boldsymbol{x}, i, x_0), (k, a_k)\big) \wedge I^\sharp_\ell\big((\boldsymbol{x}, i, x_0), (i, a_i)\big) \implies I^\sharp_{\ell'}\big((\boldsymbol{x}, i, a_i), (k, a_k)\big) \\ I^\sharp_\ell\big((\boldsymbol{x}, i, x_0), (i, a_i)\big) \implies I^\sharp_{\ell'}\big((\boldsymbol{x}, i, a_i), (i, a_i)\big) \end{cases} \\
a[i] = x_0 \text{(write)} & \rightsquigarrow & \begin{cases} I^\sharp_\ell\big((\boldsymbol{x}, i, v), (k, a_k)\big) \wedge i \neq k \implies I^\sharp_{\ell'}\big((\boldsymbol{x}, i, v), (k, a_k)\big) \\ I^\sharp_\ell\big((\boldsymbol{x}, i, v), (i, a_i)\big) \implies I^\sharp_{\ell'}\big((\boldsymbol{x}, i, v), (i, v)\big) \end{cases} \\
\Phi(\boldsymbol{x}, k, a[k]) \text{(property at} \ell) & \rightsquigarrow & I^\sharp_\ell(\boldsymbol{x}, k, a_k) \implies \Phi(\boldsymbol{x}, k, a_k).
\end{array}
$$

Figure 4.27 – Horn Clauses Generation (Abstract transformers). All statements are considered to be from the control point ℓ to ℓ', $\boldsymbol{x}$ designs the vector of all scalar variables. Properties are also abstracted.

- For a write statement $a[i] = v$ the semantics is expressed as two formulas, depending on whether the distinguished cell is i or not.
- For a read statement, the second formula expresses that a_i is assigned to variable v. The (first) nonlinear rule may be more difficult to comprehend. The intuition is that, to have both $a_i = a[i]$ and $a_k = a[k]$ at the read instruction with a given valuation $(\boldsymbol{x}, i)$ of the other variables, both $a_i = a[i]$ and $a_k = a[k]$ had to be reachable with the same valuation. We use two separate rules for $k = i$ and $k \neq i$ for better precision. A single rule $I^\sharp_1\big((\boldsymbol{x}, i, v), (k, a_k)\big) \wedge I^\sharp_1\big((\boldsymbol{x}, i, v), (i, a_i)\big) \implies I^\sharp_2\big((\boldsymbol{x}, i, a_i), (k, a_k)\big)$ would not enforce that if $i = k$ then $a_i = a_k$ in the consequent.

Applying this transformation on a programs thus gives a set of Horn Clauses that describes the abstract semantics of the program. This transformation is sound and incomplete. Algorithm 9 describes how to use it to prove a given safety property Φ of a program. Let us recall that the property to prove is also abstracted according to 4.27.

Algorithm 9 Proving array properties on a CFG

Input: CFG + Φ a property to prove **Output:** Yes/I do not know

1: To each control point p_ℓ, with vector of scalar variables $\boldsymbol{x}_\ell$, associate a predicate $I^\sharp_\ell(\boldsymbol{x}_\ell, k, a_k)$
2: For each transition of the program, generate Horn rules according to Figure 4.27.
3: Also Generate Horn queries from desired properties.
4: Call a solver.
5: **if** it answers SAT **then**
6: **return** Yes
7: **else**
8: **return** I do not know
9: **end if**

Example 15 (continuing from p. 69)**.** *Let us now apply Algorithm 9 where the predicates I_k are simply*

denoted by the label of the control point i.

$$0 \leq k < n \implies loop(n, 0, k, a_k) \tag{4.5}$$

$$0 \leq k < n \wedge i < n \wedge loop(n, i, k, a_k) \implies write(n, i, k, a_k) \tag{4.6}$$

$$0 \leq k < n \wedge i \neq k \wedge write(n, i, k, a_k) \implies incr(n, i, k, a_k) \tag{4.7}$$

$$write(n, i, i, a_i) \implies incr(n, i, i, 42) \tag{4.8}$$

$$0 \leq k < n \wedge incr(n, i, k, a_k) \implies loop(n, i+1, k, a_k) \tag{4.9}$$

$$0 \leq k < n \wedge i \geq n \wedge loop(n, i, k, a_k) \implies end(n, i, k, a_k) \tag{4.10}$$

Finally, we add the postcondition:

$$0 \leq k < n \wedge end(n, i, k, a_k) \Rightarrow a_k = 42 \tag{4.11}$$

A solution to the resulting system of Horn clauses is found by Z3, thus the property is proved.

4.4.4 Extensions

In [MG16] we provide several extensions to the previous transformation:

- **Maps.** Maps can be considered as arrays with cells of a given type. Nothing in our abstraction uses the fact that the cells suppose that our indices are integers, thus our transformations is also valid for maps.
- **2D arrays.** 2D arrays can be classically expressed as 2-indices maps. Our abstraction can thus be easily adapted to deal with them.
- **Expressing sortedness and relations between k-cells.** The Galois connection of Def. 13 expresses relations of the form $\forall k \in \mathbb{Z}\ \varphi(\boldsymbol{x}, k, a[k])$ where $\boldsymbol{x}$ are variables from the program, a a map and k an index into the map a; in other words, relations between each array element individually and the rest of the variables. It cannot express properties such as sortedness, which link *two* array elements: $\forall k_1, k_2 \in \mathbb{Z}\ k_1 < k_2 \implies a[k_1] \leq a[k_2]$. For such properties, we need two "distinguished cells", with indices k_1 and k_2. In the paper we provide adaptations of the abstraction for the case of multiple cells.
- **Multisets.** Our abstraction for maps may be used to abstract (multi)sets, which permits to express for instance that a given array is a permutation of the input.

4.4.5 Experiments

Implementation We implemented our prototype VAPHOR in 2k lines of OCAML. VAPHOR takes as input a mini-Java program (a variation of WHILE with array accesses, and assertions) and produces a SMTLIB2 file[13]. The core analyser implements the translation for one and two-dimensional arrays described in Section 4.4.3.
A second prototype was re-factored later by J. Braine, which first computes a SMTLIB2 file within the theory of Arrays, then computes its array abstraction and then produces a SMTLIB2 without arrays. A demo page can be found at the following URL: `http://laure.gonnord.org/pro/demopage/vaphor/`

Experiments We have tested our analyser on several examples from the literature, including the array benchmark proposed in [46] also used in [18] (Table 4.1); and other classical array algorithms including *selection sort*, *bubble sort* and *insertion sort* (Table 4.2). We compared our approach to existing Horn clause solvers capable of dealing with arrays. All these files are provided in the demo page.

[13] `http://smtlib.cs.uiowa.edu/`

Table 4.1 – Comparison on the array benchmarks of [46]. (Average) timing are in seconds, CPU time. Abstraction with $N = 1$. "sat" means the property was proved, "unsat" that it could not be proved. "hints" means that some invariants had to be manually supplied to the solver (e.g. even/odd conditions). A star means that we used another version of the solver. Timeout was 5 mn unless otherwise noted. The machine has 32 i3-3110M cores, 64 GiB RAM, C/C++ solvers were compiled with gcc 4.8.4, the JVM is OpenJDK 1.7.0-85.

Benchmark	Z3/PDR		Z3/Spacer		Eldarica		Comment
	Res	Time	Res	Time	Res	Time	
Correct problems, "sat" expected							
append	sat	2.11	sat	0.85	sat	22.61	
copy	sat	4.66	sat	0.44	timeout(300s)		
find	sat	0.20	sat	0.14	sat	12.93	
findnonnull	sat	0.50	sat	0.34	sat	12.04	
initcte	sat	0.16	sat	0.26	sat	13.28	
init2i	sat	0.31	sat	0.16	sat	14.67	
partialcopy	sat	1.88	sat	0.34	timeout(300s)		
reverse	sat	40.70	sat	2.19	timeout(300s)		
strcpy	sat	0.92	sat	0.37	sat*	66.69	
strlen	sat	0.24	sat	0.22	sat	36.69	
swapncopy	sat	71.16	timeout(300s)		timeout(300s)		
memcpy	sat	3.54	sat	0.39	timeout(300s)		
initeven	sat	1.32	sat	0.71	timeout(300s)		"hints"
mergeinterleave	sat	39.49	sat	4.61	timeout	322.39	"hints"
Incorrect problems, "unsat" expected							
copyodd_buggy	unsat	0.08	unsat	0.04	unsat	7.42	
initeven_buggy	unsat	0.06	unsat	0.06	unsat	6.28	
reverse_buggy	unsat	1.88	unsat	1.28	unsat	58.96	
swapncopy_buggy	unsat	3.13	unsat	0.74	unsat	27.54	
mergeinterleave_buggy	unsat	1.16	unsat	0.56	unsat	31.22	

Table 4.2 – Other array-manipulating programs, including various sorting algorithms. a star means that we used another version of the solver, R1 means `random_seed`=1. The ~~striked out~~ result is likely a bug in Z3; the alternative is a bug in Spacer, since the same system cannot be satisfiable and unsatisfiable at the same time.

Benchmark	N	Z3/PDR		Z3/Spacer		Eldarica		Comment
		Res	Time	Res	Time	Res	Time	
bin_search_check	1	sat	0.71	sat	0.34	Crash		
find_mini_check	1	sat	4.22	sat	0.82	sat	110.58	
revrefill1D_check_buggy	1	unsat	0.03	unsat	0.07	unsat	9.21	
array_init_2D	1	sat	0.46	sat	0.22	sat	12.76	
array_sort_2D	1	sat	0.78	sat	0.30	sat	26.68	
selection_sort (sortedness)	2	sat*	99.04	timeout(300s)		timeout(300s)		
selection_sort (sortedness)	2	~~unsat~~	83	sat	48	timeout	334	manual translation
selection_sort (permutation)	1	timeout	600	sat	9.24	timeout	336	manual translation
bubble_sort_simplified	2	sat	5.98	sat	2.77	sat	158.70	
insertion_sort	2	sat(R1)	53.83	timeout(300s)		timeout(300s)		

Limitations Our tool does not currently implement the reasoning over array contents (multiset of values). Experiments for these were thus conducted by manually applying the transformations described in this article in order to obtain a system of Horn clauses. For this reason, because applying rules manually is tedious and error-prone, the only sorting algorithm for which we have checked that the multiset of the output is equal to the multiset of the inputs is selection sort.
Some examples from Dillig et al. [46] involve invariants with even/odd constraints. The Horn solvers we tried do not seem to be able to infer invariants involving divisibility predicates unless these predicates were given by the user. For these cases we added these even/odd properties as additional invariants to prove.

Efficiency caveats Our tool does not currently simplify the system of Horn clauses that it produces. We have observed that, in some cases, manually simplifying the clauses (removing useless variables, inlining single antecedents by substitution...) dramatically reduces solving times. Also, precomputing some simple scalar invariants on the Horn clauses (e.g. $0 \leq k < i$ for a loop from k to $i-1$) and asserting them as assertions to prove in the Horn system sometimes reduces solving time.

We have observed that the execution time of a Horn solver may dramatically change depending on minor changes in the input, pseudo-random number generator seed, or version of the solver. For instance, the same version of Z3 solves the same system of Horn clauses (proving the correctness of selection sort) in 3m 40s or 3h 52m depending on whether the random seed is 1 or 0.[14]
Furthermore, we have run into numerous problems with solvers, including one example that, on successive versions of the same solver, produced "sat" then "unknown" and finally "unsat", as well as crashes.
For all these reasons, we believe that solving times should not be regarded too closely. The purpose of our experimental evaluation is not to benchmark solvers relative to each other, but to show that our abstraction, even though it is incomplete, is powerful enough to lead to fully automated proofs of functional correctness of nontrivial array manipulations, including sorting algorithms. Tools for solving Horn clauses are still in their infancy and we thus expect performance and reliability to increase dramatically.

4.5 Conclusion

In this chapter I summarised the contributions made in the domain of memory analyses, from scalable static analyses for pointer regions to a very expressive program abstraction to prove functional properties of programs with arrays.
The first contributions on memory abstract domains show that the *sparse* dataflow framework is the appropriate technique to design pointer analyses that scale. They also illustrate the need for *specialised* analyses to handle more or less restricted classes of programs/classes of properties. These analyses must be designed by carefully looking at the structural properties of the program to analyse.
The last contribution shows that we can design expressive abstractions that are capable of generating array invariants with a relatively low solving cost. We think that this work deserves to be pursued in the direction of more expressive data-structures and also revisited in the compilation community for instance to generate information for further code optimisation.

[14] We suspect that different choices in SAT lead to different proofs of unsatisfiability, thus different interpolants and different refinements in the PDR algorithm.

5 Conclusion and perspectives

The contributions made in this document addressed the design of efficient and expressive static verifiers or compilers. The approach we consider is the study of each class of properties (numerical properties, termination, memory properties) through various static approaches that explore the limits of expressivity and scalability. The algorithms are implemented and validated through examples of the literature or state-of-the art benchmarks. This choice was motivated by the numerous applications we can find in software validation and compilation/code optimisation.

Since I joined the LIP lab in Lyon (France) in 2013, I have been developing a new vision which fosters the combination of abstract interpretation algorithms with compilation-inspired techniques and tools. Compilation is both an application domain and an inspiration for the design of new solutions. I also benefit from the proximity of the ever growing French Community of Compilation, Code Generation, Analysis [1] that I co-animate with Florian Brandner and Fabrice Rastello.

5.1 Summary of the manuscript

This manuscript developed the contributions made in the domain of static analysis. Each chapter exposed a different application domain and the techniques we developed for enhancing precision, scalability and applicability.

Chapter 2 presented the design of numerical static analyses for numerical programs. The first contribution of the chapter is a novel iteration strategy for abstract interpreters that compute numerical invariants ([MG11]). The "path focusing" algorithm enhanced abstract interpretation with a more "semantic-directed" computation which results in more accurate numerical invariants. The second part of the chapter exposes our work on the design of a novel abstraction for synchronous languages[GG11], [FGG12], having in mind the performance and the readability of the code produced by synchronous compilers. The third contribution of the chapter is a novel abstract domain for parametric intervals [SMO+14] whose goal is to provide efficient information on the range of variables inside SSA (Single Static Assignment)-based compilers.

Chapter 3 exposed a set of contributions that were made in the domain of static analyses for proving termination of sequential programs. Working on the relationship between program scheduling and termination, we first proposed a (quasi-complete) algorithm [ADFG10] to compute multidimensional ranking functions from program invariants. This algorithm was implemented in a standalone tool called RANK, around which we tried to design heuristic strategies ([AAG12], [GAA12]) to scale better. Then we proposed an enhancement of the algorithm [GMR15] based on algebraic properties of polyhedra. Finally, in [RAPG14], we demonstrated that proving termination and computing worst-cases can sometimes be done with relatively simple strategies for general purpose programs.

[1] http://compilfr.ens-lyon.fr/

Chapter 4 presented more recent contributions focusing more on *memory* properties and optimisation. The first contribution [SMO+14] is the design of a novel array-out-of-bound detection toolchain inside the LLVM compiler. This work relied on precise yet scalable set of static analyses specially *tailored* for the application. The chapter then exposed two *sparse* static analyses for pointers ([PMB+16], [MPR+17]) that were designed with a particular thorough effort on splitting strategies to improve both scalability and precision. The chapter then described a more fundamental work on array properties [MG16] that can be seen as one of the foundations of my future work, especially from an expressivity perspective.

5.2 Future research topics

My contributions strongly promote the cross-fertilisation of abstract-interpretation techniques with compiler techniques (scheduling, thorough work on intermediate representations). I believe that the link between the two communities should be reinforced in the following directions:

- Code optimisations are potential clients for code analyses. The tiny link between them is still currently mainly unexplored, and code optimisations do not often clearly state which kind of information they rely on. The properties to find would most probably be *simple* from the static analysis community, however there will remain two challenges: the design of scalable analyses to infer them, and their efficient use for code optimisation. The Phd thesis of Maroua Maalej is only the first step toward a better understanding of this link.

- Code analysis techniques are potentially a source of inspiration in the design of more expressive compilation techniques. We believe that the use of abstractions is still at its early age inside compilers, especially in the domain of aggressive compilation of HPC kernels. The design of expressive code analyses for code (scheduling) optimisation will be enabled by a thorough study of the notion of *dependencies* with various techniques from the static analysis and the rewriting communities, in order to deal with approximation and complex data structures.

Below I give three major subtopics that I would like to address in the years to come, which concern the synergic exploitation of static analyses and compilation techniques. It will largely benefit from the results and experience already presented in the document.

5.2.1 Scalable static analyses for compilers

This research project is an ongoing research project in the more general context of the PROSPIEL Inria associate team led by Sylvain Collange, Inria Rennes, for the period 2015-2017.

The long term objective of this work is to enhance the capabilities of state-of-the-art compilers, so as for the optimisations phases to benefit from *accurate* information concerning the program and its properties. In the particular context of the project static analyses are used in combination with runtime checks to improve compiler optimisation opportunities [106].
The means we choose is to start from the knowledge we recently gained during the Phd of Maroua Maalej and mostly described in Chapter 4. The contribution of this PhD thesis is double: the demonstration that current state-of-the art compiler optimisations do not reach their maximal power since the knowledge they get from the preceding analyses is not as precise as it could be ; and also three novel *sparse* thus scalable pointer analyses capable of capturing *pointer arithmetic* properties.

The particular memory analyses we designed all share the common point that they were designed to address the particular problem induced by pointer arithmetic in production compilers: all optimisations done in a given compiler operate on *intermediate* representations where the data structures (arrays, lists, structs) accesses are all expressed in terms of loads and stores from base pointers. Any clever optimisation (loop code motion, loop fusions, array compaction, ...) thus strongly relies on

a precise handling of the semantics of all these operations. From that perspective, we only made a first step while designing our analyses: a huge amount of work remains to be done in order to redesign optimisations that are capable to be generic enough to benefit from any gain in the previous analyses steps.

In future work we plan to continue the work into the direction of scaling analyses for compilers. In particular, we have the objective to design specialised analyses but with an explicit notion of cost/-precision compromise, in the spirit of the paper [88] that tries to formalise the cost/precision compromise of interprocedural analyses with respect to a "context sensitivity parameter". Another challenge is to find a generic way of adapting existing (relational) abstract domains in the Single Static Information framework so as to improve their scalability.

Collaborators on this topic Sylvain Collange (Inria Rennes), Fabrice Rastello (Inria Rhône Alpes), David Monniaux (Verimag), Fernando Pereira (University of Mineas Gerais, Brasil).

5.2.2 Complex data structures, from expressive analyses to efficient scheduling

This research project is the subject of a ANR JCJC (Young researcher funding) project that was accepted in July 2017. I am the principal investigator of this project.

The long term objective of this work is to give a general way to reason and manipulate programs with general control flow and complex data structures, so as to be able to extract parallelism information and schedule them properly.
The means we choose is to start with the current state of the polyhedral model (Section 3.2.1), which has proved its success in the aggressive compilation and code generation of computation-intensive kernels[2]. We want to enhance the framework theoretically and algorithmically in order to be able to deal with more general programs. As a first step, we first investigate general control flow and focus on specific data structures, such as lists and trees.

This research project takes inspiration from the termination work we described in Chapter 3, where the programs had complex control, but the complex data structures were abstracted away by a coarse abstraction. For instance, write operations were ignored, and read operations were assumed to return random values. Our array abstractions of Section 4.4 are a first step toward the understanding of more complex data structures, however we cannot rely on complex solvers for compilation.

In the context of the ANR submission we studied the relationship between termination and the parallel complexity of programs operating on abstract data types such as lists and trees. In the preliminary paper [AFG16], we revisited the ideas that were at the origin of the SAS'10 paper ([ADFG10]) by replacing the termination algorithm for numerical programs by a termination algorithm from the rewriting community (termination of rewriting systems are powerful to deal with abstract data types). From a termination proof we are capable to infer the "parallel complexity" of the program under analysis, which gives an upper bound on the complexity of the program if it were executed on multiprocessors platforms.

In future work we plan to continue the work around this relationship, especially by designing abstractions that will scale enough to capture the behaviour of bigger size programs. These abstractions should also be precise enough to capture all the *computation dependences* in complex programs/data structures. We may draw inspiration of our work on array properties ([MG16]). The challenge will also to be able to deal with the approximation induced by the abstractions, especially when we have to generate code, so as to avoid too many useless computations, or worse, optimisations which would cause the final optimised program to be incorrect.

[2] http://polyhedral.info/

Collaborators on this topic Christophe Alias (Inria Rhône Alpes), Carsten Fuhs (University of Birbeck, UK), Lionel Morel (Insa Lyon), David Monniaux (Verimag, Grenoble), Tomofumi Yuki (Inria Rennes).

5.2.3 Dataflow model: from semantics to efficient parallel compilation

This research project is part of a current Inria joint-team proposal at LIP.

The long term objective of this work is to propose a novel *intermediate* representation to compile to and from. The novel representation should be expressive enough to be able to deal with all kinds of parallelism, and simple enough to be able to reason on it and provide *scalable* analyses on it.

The means we choose is the study of the dataflow model for programs: the dataflow formalism expresses a computation on an infinite number of values, that can be viewed as successive values of a variable during time. A dataflow program is structured as a set of *communicating processes* that communicate values through *communicating buffers*. Examples of dataflow languages include the synchronous languages Lustre and Signal for which we already produced static analyses (Section 2.4), as well as SigmaC [8]; the DPN representation [4] (data-aware process network, in the context of hardware compilation) is an example of a dataflow intermediate representation for a parallelizing compiler.

The dataflow model, which expresses at the same time data parallelism and task parallelism, is in our opinion one of the best models for analysis, verification and synthesis of parallel systems. This model will be our favourite representation for our programs. Indeed, it shares the "equational" description of computation and data with the polyhedral model, and the static single assignment representation inside compilers.

We plan to work on various application domains, from algebra HPC *kernels* to more complex applications like *software radio* or *deep learning* applications where the main challenge is to be able to transfer big amount of data between computation units. The work on these various domains will be driven by the idea that exploring various topics is a way to converge on unifying representations and algorithms even for specific applications.

All these applications led to the same research goal: finding a way to integrate computations, data, scheduling, distribution in a common analysis and compilation framework. We will study each of these problems from the intermediate representation point of view, like in Section 5.2.2. Abstractions, partial evaluation, and other formal tools will be used in order to revisit the already plethoric literature on Kahn process network variants [9, 73] where static scheduling is very often the main purpose. In this research project, we specifically want to take into account the code of the tasks as well as there producer/consumer characteristics, in a general expressive framework from the program to the code generation.

Collaborators on this topic Christophe Alias (Inria Rhône Alpes), Lionel Morel (Insa Lyon), Matthieu Moy (Verimag, Grenoble).

A Main Symbols and Acronyms

$\boldsymbol{x}$ Vector of program's variables. 7

$\mathscr{K}$ Set of program's control points. 7

$\mathscr{T}$ Set of program's transitions. 7

AIA affine (integer) interpreted automaton. 7

$\mathscr{R}_k$ Valuation of the program's variables at control point k. 8

$\tau_{k,k'}$ Concrete semantics of the transition $k \rightarrow k'$. 8

$\tau^\sharp$ Abstract semantics of the transition τ. 8

$\tau = \cup_{t \in \mathscr{T}} \tau_t$ Transition relation for the whole program. 31

$\mathscr{I}_k$ Inductive invariant at control point k. 10

$\triangledown$ Widening operator. 9

CFG Control Flow Graph. 10

SSA Single Static Assignment. 13

$(\hat{\ }, \tilde{\ })$ Clock/Value abstraction for SIGNAL programs. 19

SSI Static Single Information. 22

eSSA Extended Static Single Assignment Form. 23

SymBoxes Symbolic Range abstract domain. 24

$R(x)$ (possibly symbolic) Interval for scalar x. 24

$(R_\downarrow, R_\uparrow)$ Lower/Upper bound of the interval R. 24

ρ Ranking function. 28

$\mathscr{Q}_t$ Polyhedral transition relation induced by the transition t. 31

$\mathscr{P}_{\mathscr{I},\tau}$ Set of all "reachable one-step differences" of the program. 35

$LP(V, \textbf{\textit{Constraints}}(\mathscr{I}))$ LP ranking instance with generators V and invariant $\mathscr{I}$. 35

WCCC Worst-case Computational Complexity. 38

$W(p)$ Valid offsets from base p. 51

SymRegion Symbolic Region abstract domain. 51

$\mathscr{L}oc$ Set of all allocation sites of the program. 58

MemLocs Memory Locations abstract domain. 58

GR Global Range Analysis (CGO'16). 58

LT Less Than Analysis (CGO'17). 61

CFC Constrained Horn Clause. 69

$I_\ell^\sharp$ Abstract invariant predicate at control point ℓ (SAS'16). 69

B Bibliography

References to other publications

B.1 Bibliography

[1] A. Aho, R. Sethi, et J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986-2006.

[2] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, et N. Sharygina. An extension of lazy abstraction with interpolation for programs with arrays. *Formal Methods in Systems Design*, 45(1): 63–109, 2014. doi: 10.1007/s10703-014-0209-9.

[3] Francesco Alberti et David Monniaux. Polyhedra to the rescue of array interpolants. In *Symposium on applied computing (Software Verification & Testing)*, pages 1745–1750. ACM, 2015. doi: 10.1145/2695664.2695784. (PDF).

[4] Christophe Alias et Alexandru Plesco. Data-aware Process Networks. Research Report RR-8735, Inria - Research Centre Grenoble – Rhône-Alpes, June 2015. (Web). (PDF).

[5] Scott Ananian. The static single information form. Master's thesis, MIT, September 1999. (PDF).

[6] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. Thèse de doctorat, DIKU, University of Copenhagen, 1994.

[7] Mihail Asavoae, Claire Maiza, et Pascal Raymond. Program semantics in model-based WCET analysis: A state of the art perspective. In *13th International Workshop on Worst-Case Execution Time Analysis, WCET 2013, July 9, 2013, Paris, France*, pages 32–41, 2013. doi: 10.4230/OASIcs.WCET.2013.32. (Web).

[8] Pascal Aubry, Pierre-Edouard Beaucamps, Frédéric Blanc, Bruno Bodin, Sergiu Carpov, Loïc Cudennec, Vincent David, Philippe Doré, Paul Dubrulle, Benoît Dupont De Dinechin, François Galea, Thierry Goubier, Michel Harrand, Samuel Jones, Jean-Denis Lesage, Stéphane Louise, Nicolas Morey Chaisemartin, Thanh Hai Nguyen, Xavier Raynaud, et Renaud Sirdey. Extended Cyclostatic Dataflow Program Compilation and Execution for an Integrated Manycore Processor. In *Alchemy 2013 - Architecture, Languages, Compilation and Hardware support for Emerging ManYcore systems*, volume 18 de *Proceedings of the International Conference on Computational Science, ICCS 2013*, pages 1624–1633, Barcelona, Spain, June 2013. doi: 10.1016/j.procs.2013.05.330. (Web). (PDF).

[9] Pascal Aubry, Pierre-Edouard Beaucamps, Frédéric Blanc, Bruno Bodin, Sergiu Carpov, Loïc Cudennec, Vincent David, Philippe Doré, Paul Dubrulle, Benoît Dupont De Dinechin, François Galea, Thierry Goubier, Michel Harrand, Samuel Jones, Jean-Denis Lesage, Stéphane Louise, Nicolas Morey Chaisemartin, Thanh Hai Nguyen, Xavier Raynaud, et Renaud Sirdey. Extended Cyclostatic Dataflow Program Compilation and Execution for an Integrated Manycore Processor. In *Alchemy 2013 - Architecture, Languages, Compilation and Hardware support for Emerging ManYcore systems*, volume 18 de *Proceedings of the International Conference on Computational Science, ICCS 2013*, pages 1624–1633, Barcelona, Spain, June 2013. doi: 10.1016/j.procs.2013.05.330. (Web). (PDF).

[10] R. Bagnara, P. M. Hill, E. Ricci, et E. Zaffanella. Precise widening operators for convex polyhedra. *Science of Computer Programming*, 58(1–2):28–56, 2005. (PDF).

[11] Roberto Bagnara, Patricia M. Hill, et Enea Zaffanella. The Parma Polyhedra Library, version 0.9. (Web).

[12] Clark Barrett, Pascal Fontaine, et Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB), 2016. (Web).

[13] Christian Bauer, Alexander Frink, et Richard Kreckel. Introduction to the GiNaC framework for symbolic computation within the C++ programming language. *J. Symb. Comput.*, 33(1):1–12, 2002. (PDF).

[14] Amir M. Ben-Amram et Samir Genaim. Ranking functions for linear-constraint loops. *J. ACM*, 61(4):26:1–26:55, July 2014. doi: 10.1145/2629488. (PDF).

[15] Amir M. Ben-Amram et Chin Soon Lee. Program termination analysis in polynomial time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(1):5, 2007.

[16] Frédéric Besson, Thomas Jensen, et Jean-Pierre Talpin. Polyhedral analysis for synchronous languages. In *Proceedings of the 6th International Symposium on Static Analysis, volume 1694 of Lecture Notes in Computer Science*, pages 51–68. Springer-Verlag, September 1999. (PDF).

[17] A. Biere, A. Biere, M. Heule, H. van Maaren, et T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.

[18] Nikolaj Bjørner, Ken McMillan, et Andrey Rybalchenko. On solving universally quantified Horn clauses. In *Static Analysis Symposium (SAS)*, pages 105–125, 2013. doi: 10.1145/2695664.2695784. (PDF).

[19] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, et Xavier Rival. A static analyzer for large safety-critical software. In *Programming Language Design and Implementation (PLDI)*, pages 196–207. ACM, 2003. doi: 10.1145/781131.781153. (PDF).

[20] William Blume et Rudolf Eigenmann. Symbolic range propagation. In *Proceedings of the 9th International Parallel Processing Symposium (IPPS)*, pages 357–363, 1995.

[21] Rastislav Bodik, Rajiv Gupta, et Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *Programming Language Design and Implementation (PLDI)*, pages 321–333. ACM, 2000.

[22] Bernard Boigelot. Symbolic methods for exploring infinite state spaces. Phd thesis, Université de Liège, 1999. (PDF).

[23] Cari Borrás. Overexposure of radiation therapy patients in Panama: problem recognition and follow-up measures. *Pan-American J. of public health*, 20(2/3):173–187, September 2006. (Web). (PDF).

[24] Marius Bozga, Codruta Girlea, et Radu Iosif. Iterating Octagons. In Anna Kowalewski, Stefan; Philippou, éd. sci., *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5505 de *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, March 2009. doi: 10.1007/978-3-642-00768-2. (Web). (PDF).

[25] Aaron A. Bradley, Zohar Manna, et Henny B. Sipma. The polyranking principle. In *32nd International Colloquium on Automata, Languages and Programming (ICALP)*, volume 3580 de *Lecture Notes in Computer Science*, pages 1349–1361. Springer Verlag, July 2005. (PDF).

[26] Aaron R. Bradley, Zohar Manna, et Henny B. Sipma. Linear ranking with reachability. In Kousha Etessami et Sriram K. Rajamani, éds. sci., *17th International Conference on Computer Aided Verification (CAV)*, volume 3576 de *Lecture Notes in Computer Science*, pages 491–504. Springer Verlag, July 2005. (PDF).

[27] Aaron R. Bradley, Zohar Manna, et Henny B. Sipma. Termination analysis of integer linear loops. In *16th International Conference on Concurrency Theory (CONCUR)*, volume 3653 de *Lecture Notes in Computer Science*, pages 488–502. Springer Verlag, August 2005. (PDF).

[28] Guillaume P. Brat, Doron Drusinsky, Dimitra Giannakopoulou, Allen Goldberg, Klaus Havelund, Michael R. Lowry, Corina S. Pasareanu, Arnaud Venet, Willem Visser, et Richard Washington. Experimental evaluation of verification and validation tools on martian rover software. *Formal Methods in System Design*, 25(2-3):167–198, 2004. doi: 10.1023/B:FORM.0000040027.28662.a4. (PDF).

[29] Aziem Chawdhary, Byron Cook, Sumit Gulwani, Mooly Sagiv, et Hongseok Yang. Ranking abstractions. In *17th European Symposium on Programming (ESOP'08)*, volume 4960 de *Lecture Notes in Computer Science*, pages 81–92, Budapest, April 2008. Springer Verlag.

[30] Jürgen Christ. *Interpolation Modulo Theories*. Thèse de doctorat, University of Freiburg, 2015. (PDF).

[31] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, et Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson et Aravinda Prasad Sistla, éds. sci., *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings*, pages 154–169, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. doi: 10.1007/10722167_15. (PDF).

[32] Philippe Clauss. Handling memory cache policy with integer points counting. In *Parallel Processing, 3rd International Euro-Par Conference*, volume 1300 de *Lecture Notes in Computer Science*, pages 285–293, Passau, August 1997. Springer Verlag. (PDF).

[33] Michael Colón et Henny Sipma. Synthesis of linear ranking functions. In *7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 de *Lecture Notes in Computer Science*, pages 67–81. Springer Verlag, 2001.

[34] Michael A. Colón et Henny B. Sipma. Practical methods for proving program termination. In *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 de *Lecture Notes in Computer Science*, pages 442–454. Springer Verlag, January 2002. (PDF).

[35] Hubert Comon et Yan Jurski. Multiple counters automata, safety analysis and Presburger arithmetic. In *CAV'98*, Vancouver (B.C.), 1998. LNCS 1427, Springer Verlag. (PDF).

[36] A. Costan, S. Gaubert, E. Goubault, M. Martel, et S. Putot. *A Policy Iteration Algorithm for Computing Fixed Points in Static Analys is of Programs*, pages 462–475. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. doi: 10.1007/11513988_46. (PDF).

[37] P. Cousot et R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, Los Angeles, January 1977. (PDF).

[38] Patrick Cousot. Proving program invariance and termination by parametric abstraction, Lagrangian relaxation, and semidefinite programming. In *6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 de *Lecture Notes in Computer Science*, pages 1–24, Paris, January 2005. Springer Verlag. (PDF).

[39] Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes*. State doctorate thesis, Université scientifique et médicale de Grenoble and Institut National Polytechnique de Grenoble, 1978. (Web).

[40] Patrick Cousot et Radhia Cousot. Invited talk: Higher order abstract interpretation. In *IEEE International Conference on Computer Languages*, pages 95–112. IEEE, 1994.

[41] Patrick Cousot et Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, pages 511–547, August 1992. doi: 10.1093/logcom/2.4.511. (PDF).

[42] Patrick Cousot et Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages (POPL)*, pages 84–96. ACM, 1978. doi: 10.1145/512760.512770. (PDF).

[43] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, et Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991. doi: 10.1145/115372.115320. (PDF).

[44] Alain Darte. Understanding loops: The influence of the decomposition of Karp, Miller, and Winograd. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign*, MEMOCODE '10, pages 139–148, Grenoble, France, July 2010. IEEE Computer Society. Invited paper and keynote talk.

[45] Alain Darte. Optimal parallelism detection in nested loops. In David Padua, éd. sci., *Encyclopedia of Parallel Programming*. Springer, 2011.

[46] Işıl Dillig, Thomas Dillig, et Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In *European Conference on Programming Languages and Systems (ESOP)*, pages 246–266, 2010. doi: 10.1007/978-3-642-11957-6_14. (PDF).

[47] E. Duesterwald et V. Bala. Software profiling for hot path prediction: Less is more. In *ASPLOS*, pages 202–211. ACM, 2000.

[48] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–51, 1991. (PDF).

[49] Paul Feautrier. Some efficient solutions to the affine scheduling problem, II, multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[50] Paul Feautrier et Christian Lengauer. The polyhedron model. In David Padua, éd. sci., *Encyclopedia of Parallel Programming*. Springer, 2011.

[51] J. Ferrante, J. Ottenstein, et D. Warren. The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319–349, 1987. (PDF).

[52] A. Finkel et J. Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *Foundations of Software Technology and Theoritical Computer Science (FSTTCS'2002)*, pages 145–156, Kanpur, India, December 2002. Springer. (PDF).

[53] Robert W. Floyd. Assigning meaning to programs. In J. T. Schwartz, éd. sci., *Symposium on Applied Mathematics*, volume 19, pages 19–32. A.M.S., 1967. (PDF).

[54] Abdoulaye Gamatié. *Designing Embedded Systems with the* SIGNAL *Programming Language: Synchronous, Reactive Specification*. Springer, New York, 2009.

[55] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, et René Thiemann. Proving termination of programs automatically with AProVE. In Stéphane Demri, Deepak Kapur, et Christoph Weidenbach, éds. sci., *Automated Reasoning (IJCAR)*, volume 8562 de *LNCS*, pages 184–191. Springer, 2014. doi: 10.1007/978-3-319-08587-6_13. (PDF).

[56] Denis Gopan et Thomas Reps. Lookahead widening. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 452–466. Springer Berlin Heidelberg, 2006. (PDF).

[57] Dirk Grunwald, Benjamin Zorn, et Robert Henderson. Improving the cache locality of memory allocation. In *In PLDI*, pages 177–186. ACM, 1993.

[58] Sumit Gulwani, Sagar Jain, et Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*, pages 375–385, Dublin, 2009. ACM.

[59] Sumit Gulwani, Krishna K. Mehra, et Trishul Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *36th ACM Symposium on Principles of Programming Languages (POPL'09)*, pages 127–139, Savannah, January 2009.

[60] Nicolas Halbwachs. Delay analysis in synchronous programs. In Costas Courcoubetis, éd. sci., *Computer Aided Verification (CAV)*, volume 697 de *LNCS*, pages 333–346. Springer, 1993. doi: 10.1007/3-540-56922-7_28. (PDF).

[61] Nicolas Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. State doctorate thesis, Université scientifique et médicale de Grenoble and Institut National Polytechnique de Grenoble, 1979. (Web). (PDF).

[62] Nicolas Halbwachs, Fabienne Lagnier, et Christophe Ratel. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, sep 1992.

[63] Ben Hardekopf et Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*, pages 290–299. ACM, 2007. (PDF).

[64] Ben Hardekopf et Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *CGO*, pages 265–280, 2011.

[65] Julien Henry, David Monniaux, et Matthieu Moy. PAGAI: a path sensitive static analyzer. In *Tools for Automatic Program AnalysiS (TAPAS 2012)*, page 3, Deauville, France, September 2012. (Web). (PDF).

[66] Julien Henry, Mihail Asavoae, David Monniaux, et Claire Maiza. How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. In *SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2014, LCTES '14, Edinburgh, United Kingdom - June 12 - 13, 2014*, pages 43–52, 2014. doi: 10.1145/2597809.2597817. (Web). (PDF).

[67] Julien Henry, David Monniaux, et Matthieu Moy. The Pagai static analyser, 2014. (Web).

[68] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE*, pages 54–61. ACM, 2001.

[69] K. Hoder et N. Bjørner. Generalized property directed reachability. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 7317 de *LNCS*, pages 157–171. Springer, 2012. doi: 10.1007/978-3-642-31612-8_13.

[70] ISO-Standard. 9899 - The C programming language, 2011.

[71] Daniel Jackson. A direct path to dependable software. *Commun. ACM*, 52(4):78–88, April 2009. doi: 10.1145/1498765.1498787. (Web).

[72] B. Jeannet, N. Halbwachs, et P. Raymond. Dynamic partitioning in analyses of numerical properties. In A. Cortesi et G. Filé, éds. sci., *Static Analysis Symposium, SAS'99*, Venice (Italy), September 1999. LNCS 1694, Springer Verlag. (PDF).

[73] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress 74*, pages 471–475, 1974.

[74] A. Komuravelli, A. Gurfinkel, S. Chaki, et E. M. Clarke. Automatic Abstraction in SMT-Based Unbounded Software Model Checking. In *CAV*, volume 8044, pages 846–862. Springer, 2013. doi: 10.1007/978-3-642-39799-8_59. (PDF).

[75] A. Komuravelli, A. Gurfinkel, et S. Chaki. Smt-based model checking for recursive programs. In Armin Biere et Roderick Bloem, éds. sci., *CAV*, volume 8559 de *LNCS*, pages 17–34. Springer, 2014. doi: 10.1007/978-3-319-08867-9_2. (PDF).

[76] Daniel Kroening et Ofer Strichman. *Decision procedures*. Springer, 2008.

[77] Chris Lattner et Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society. (PDF).

[78] P. Le Guernic, J.-P. Talpin, et J.-C. Le Lann. Polychrony for System Design. *Journal for Circuits, Systems and Computers*, 12(3):261–304, April 2003. (PDF).

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

[80] Zohar Manna. *Mathematical Theory of Computing*. MacGraw-Hill, 1974.

[81] K.L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136, 2006. doi: 10.1007/11817963_14. (PDF).

[82] Antoine Miné. Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions. *Science of Computer Programming*, 2013. (PDF).

[83] David Monniaux. A survey of satisfiability modulo theory. In *Computer Algebra in Scientific Computing*, volume 9890 de *Lecture Notes in Computer Science*. Springer Verlag, September 2016. (PDF).

[84] David Monniaux et Francesco Alberti. A simple abstraction of arrays and maps by program translation. In Sandrine Blazy et Thomas Jensen, éds. sci., *Static Analysis Symposium (SAS)*, pages 217–234, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. doi: 10.1007/978-3-662-48288-9_13. (PDF).

[85] G C. Necula, S McPeak, S Rahul, et W Westley. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228. Springer-Verlag, 2002. (PDF). LNCS 2304.

[86] Flemming Nielson, Hanne Riis Nielson, et Chris Hankin. *Principles of program analysis*. Springer, 2005.

[87] Robert Nieuwenhuis et Albert Oliveras. On SAT modulo theories and optimization problems. In *Theory and Applications of Satisfiability Testing - SAT 2006: 9th International Conference, Seattle, WA, USA, August 12-15, 2006. Proceedings*, pages 156–169, 2006. doi: 10.1007/11814948_18. (PDF).

[88] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, et Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. In *Proceedings of Programming Language Design and Implementation*, page 49. ACM, 2014. (PDF).

[89] Andreas Podelski et Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In Bernhard Steffen et Giorgio Levi, éds. sci., *Verification, Model Checking, and Abstract Interpretation (VMCAI'03)*, volume 2937 de *Lecture Notes in Computer Science*, pages 239–251. Springer Verlag, 2004. (PDF).

[90] D. J. Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000. (PDF).

[91] Xavier Rival et Laurent Mauborgne. The trace partitioning abstract domain. *Transactions on Programming Languages and Systems (TOPLAS)*, 29(5):26, 2007. doi: 10.1145/1275497.1275501. (PDF).

[92] Bruno Rodrigues, Fernando Magno Quintão Pereira, et Diego F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *Compiler Construction*, pages 110–120. ACM, 2016. (PDF).

[93] Raphael Ernani Rodrigues, Victor Hugo Sperle Campos, et Fernando Magno Quint ao Pereira. A fast and low-overhead technique to secure programs again integer overflows. In *International Symposium on Code Generation and Optimization (CGO)*, 2013. (PDF).

[94] P. Rümmer, H. Hojjat, et V. Kuncak. Disjunctive interpolants for Horn-clause verification. In Natasha Sharygina et Helmut Veith, éds. sci., *Computer-aided verification (CAV)*, volume 8044 de *LNCS*, pages 347–363. Springer, 2013. doi: 10.1007/978-3-642-39799-8_24. (PDF).

[95] Barbara G. Ryder, William A. Landi, Philip A. Stocks, Sean Zhang, et Rita Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Trans. Program. Lang. Syst.*, 23(2):105–186, 2001.

[96] A. Schrijver. *Theory of linear and integer programming*. Wiley, NewYork, 1986.

[97] Edward J. Schwartz, Thanassis Avgerinos, et David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society. (PDF).

[98] Roberto Sebastiani et Silvia Tomasi. Optimization in SMT with $\mathcal{LA}(\mathbb{Q})$ cost functions. In *Proceedings of the 6th international joint conference on Automated Reasoning*, (IJCAR'12), pages 484–498. Springer, 2012. doi: 10.1007/978-3-642-31365-3_38. (PDF).

[99] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, et Dmitry Vyukov. AddressSanitizer: a fast address sanity checker. In *USENIX ATC*, pages 28–28. USENIX Association, 2012. (PDF).

[100] Lei Shang, Xinwei Xie, et Jingling Xue. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 264–274, New York, NY, USA, 2012. ACM. (PDF).

[101] Pascal Sotin, Bertrand Jeannet, et Xavier Rival. Concrete Memory Models for Shape Analysis. In *NSAD'2010 - Second International Workshop on Numerical and Symbolic Abstract Domains*, volume 267, pages 139–150, Perpignan, France, September 2010. Elsevier. (Web). (PDF).

[102] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM. (PDF).

[103] Chengnian Sun, Vu Le, Qirun Zhang, et Zhendong Su. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 294–305, New York, NY, USA, 2016. ACM. doi: 10.1145/2931037.2931074. (Web).

[104] Andre L. C. Tavares, Benoit Boissinot, Fernando M. Q. Pereira, et Fabrice Rastello. Parameterized construction of program representations for sparse dataflow analyses. In *Compiler Construction*, pages 2–21. Springer, 2014. (PDF).

[105] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, et Maurice Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 48(1):37–66, March 2007.

[106] Philippe Virouleau, François Broquedis, Thierry Gautier, et Fabrice Rastello. Using data dependencies to improve task-based scheduling strategies on NUMA architectures. In *Euro-Par 2016*, Euro-Par 2016, Grenoble, France, August 2016. (Web). (PDF).

[107] Frédéric Vivien et Martin Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, Proceedings of Programming Language Design and Implementation (PLDI), pages 35–46, New York, NY, USA, 2001. ACM. (PDF).

[108] R Wilson, R French, C Wilson, S Amarasinghe, J Anderson, S Tjiang, S-W Liao, C-W Tseng, M Hall, M Lam, et J Henessy. The SUIF compiler system: a parallelizing and optimizing research compiler. Research Report CSL-TR-94-620, Stanford University, Computer Research Laboratory, 1994. (PDF).

[109] Suan Hsi Yong et Susan Horwitz. Pointer-range analysis. In Roberto Giacobazzi, éd. sci., *Static Analysis: 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004. Proceedings*, pages 133–148, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. (PDF).

[110] Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, et Zhendong Su. Efficient subcubic alias analysis for C. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 829–845, New York, NY, USA, 2014. ACM. (PDF).

[111] Qin Zhao, Rodric Rabbah, et Weng-Fai Wong. Dynamic memory optimization using pool allocation and prefetching. *SIGARCH Comput. Archit. News*, 33(5):27–32, 2005. (PDF).

[112] Florian Zuleger, Sumit Gulwani, Moritz Sinn, et Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In *Proceedings of the 18th international conference on Static analysis*, SAS'11, pages 280–297, Berlin, Heidelberg, 2011. Springer-Verlag. (PDF).

Personal publications, sorted by type

B.2 Journal Papers

[FGG12] Paul Feautrier, Abdoulaye Gamatié, and Laure Gonnord. Enhancing the Compilation of Synchronous Dataflow Programs with a Combined Numerical-Boolean Abstraction. *CSI Journal of Computing*, 1(4):8:86–8:99, 2012. (PDF).

[GB09] Laure Gonnord and Jean-Philippe Babau. Qinna : a component-based framework for runtime safe resource adaptation of embedded systems. *Scalable Computing : Practice and Experience (SCPE)*, 10(3):253–264, 2009.

[GS14] Laure Gonnord and Peter Schrammel. Abstract Acceleration in Linear Relation Analysis. *Science of Computer Programming*, pages 125–153, 2014. (PDF).

[HMG06] Nicolas Halbwachs, David Merchat, and Laure Gonnord. Some ways to reduce the space dimension in polyhedra computations. *Formal Methods in System Design*, 29(1):79–95, 2006. (PDF).

[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. (PDF). Accepted in Oct 2017, final published version of https://hal.inria.fr/hal-01429777v2.

B.3 Conferences

[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. Springer. (PDF).

[GB08] L. Gonnord and J.-P. Babau. Runtime resource assurance and adaptation with Qinna framework: a case study. In *Proceedings of the Multiconference on Computer Science and Information Technology, Real Time Software, RTS'08*, pages 617–624, Wisla, Poland, October 2008. IEEE CS Press, CA.

[GB09] L. Gonnord and J.-P. Babau. Quantity of Resource Properties Expression and Runtime Assurance for Embedded Systems. In *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications, AICCSA'09*, pages 428–435, Rabbat, Morocco, May 2009.

[GG11] Abdoulaye Gamatié and Laure Gonnord. Static analysis of synchronous programs in signal for efficient design of multi-clocked embedded systems. In *Proceedings of the Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES 2011)*, Chicago, USA, April 2011.

[GH06] Laure Gonnord and Nicolas Halbwachs. Combining widening and acceleration in linear relation analysis. In *Proceedings of the 17th International Static Analysis Symposium, SAS'06*, Seoul, Korea, August 2006. Springer. (PDF).

[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. (PDF).

[MG11] David Monniaux and Laure Gonnord. Using bounded model checking to focus fixpoint iterations. In *Proceedings of the 18th International Static Analysis Symposium, SAS'11*, Venice, Italy, September 2011. Springer. (PDF).

[MG16] David Monniaux and Laure Gonnord. Cell morphing: from array programs to array-free Horn clauses. In Xavier Rival, editor, *23rd Static Analysis Symposium (SAS 2016)*, Static Analysis Symposium, Edimbourg, United Kingdom, September 2016. (PDF).

[MPR+17] Maroua Maalej, Vitor Paisante, Pedro Ramos, Laure Gonnord, and Fernando Pereira. Pointer Disambiguation via Strict Inequalities. In *Code Generation and Optimisation*, Austin, United States, February 2017. (PDF).

[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 Optmization*, pages 791–809, Barcelon, Spain, March 2016. (PDF).

[RGC11] Vlad Rusu, Laure Gonnord, and Benoît Combemale. A generic tool for formally tracing executions back to a dsml's operational semantics. In *Proceedings of the Seventh European Conference on Modelling Foundations and Applications (ECMFA 2011)*, Birmingham, UK, June 2011. (PDF).

[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. (PDF).

B.4 Workshops and others

[AAG12] Guillaume Andrieu, Christophe Alias, and Laure Gonnord. SToP: Scalable termination analysis of (c) programs (tool presentation). In *International Workshop on Tools for Automatic Program Analysis (TAPAS'12)*, Deauville, France, September 2012.

[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. (Web).

[AFG16] Christophe Alias, Carsten Fuhs, and Laure Gonnord. Estimation of Parallel Complexity with Rewriting Techniques. In *Workshop on Termination*, Workshop on Termination, Obergurgl, Austria, September 2016. (PDF).

[DG07] Laure Danthony-Gonnord. *Accélération abstraite pour l'amélioration de la précision en Analyse des Relations Linéaires*. Thèse de doctorat, Université Joseph Fourier, Grenoble, October 2007. (PDF).

[FG10] P. Feautrier and L. Gonnord. Accelerated Invariant Generation for C Programs with Aspic and C2fsm. In *Workshop on Tools for Automatic Program AnalysiS, TAPAS'10*, Perpignan, France, September 2010. (PDF).

[GHR04] L. Gonnord, N. Halbwachs, and P. Raymond. From discrete duration calculus to symbolic automata. In *3rd International Workshop on Synchronous Languages, Applications, and Programs, SLAP'04*, Barcelona, Spain, March 2004. (PDF).

[RAPG14] Raphael Ernani Rodrigues, Péricles Alves, Fernando Pereira, and Laure Gonnord. Real-world loops are easy to predict : a case study. In *Workshop on Software Termination*, Vienne, Austria, July 2014. (PDF).

B.5 Reports

[GAA12] Laure Gonnord, Guillaume Andrieu, and Christophe Alias. Modular termination of C programs. Research Report RR-8166, INRIA, December 2012. (PDF).