IMPLICIT COMPUTATIONAL COMPLEXITY AND COMPILERS

BY

THOMAS RUBIANO

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science in Université Paris 13 (LIPN)
Department of Computer Science University of Copenhagen (DIKU)

Defended publicly the 1st of December 2017 at Villetaneuse, France

Doctoral Committee:

Guillaume Bonfante	Université de Lorraine	Reviewer
Christophe Fouqueré	Université Paris13	Examiner
Laure Gonnord	Université Lyon1	Reviewer
Tobias Grosser	ETH Zürich	Examiner
Stefano Guerrini	Université Paris13	Examiner
Virgile Mogbil	Université Paris13	Director
Torben Mogensen	Københavns Universitet	Examiner
Jean-Yves Moyen	Université Paris13	Co-director
Ulrich Schöpp	Ludwig-Maximilians-Universität München	Examiner
Jakob Grue Simonsen	Københavns Universitet	Director

Abstract

Implicit Computational Complexity and Compilers

Complexity theory helps us predict and control resources, usually time and space, consumed by programs. Static analysis on specific syntactic criterion allows us to categorize some programs. A common approach is to observe the program's data's behavior. For instance, the detection of non-size-increasing programs is based on a simple principle : counting memory allocation and deallocation, particularly in loops. This way, we can detect programs which compute within a constant amount of space. This method can easily be expressed as property on control flow graphs. Because analyses on data's behaviour are syntactic, they can be done at compile time. Because they are only static, those analyses are not always computable or easily computable and approximations are needed. "Size-Change Principle" from C. S. Lee, N. D. Jones et A. M. Ben-Amram presented a method to predict termination by observing resources evolution and a lot of research came from this theory. Until now, these implicit complexity theories were essentially applied on more or less toy languages. This thesis applies implicit computational complexity methods into "real life" programs by manipulating intermediate representation languages in compilers. This give an accurate idea of the actual expressivity of these analyses and show that implicit computational complexity and compilers communities can fuel each other fruitfully. As we show in this thesis, the methods developed are quite generals and open the way to several new applications.

keywords Implicit computational complexity, Static analysis, Compilers

Résumé

Complexité Implicite et compilateurs

La théorie de la complexité s'intéresse à la gestion des ressources, temps ou espace, consommés par un programme lors de son exécution. L'analyse statique nous permet de rechercher certains critères syntaxiques afin de catégoriser des familles de programmes. L'une des approches les plus fructueuses dans le domaine consiste à observer le comportement potentiel des données manipulées. Par exemple, la détection de programmes "non size increasing" se base sur le principe très simple de compter le nombre d'allocations et de dé-allocations de mémoire, en particulier au cours de boucles et on arrive ainsi à détecter les programmes calculant en espace constant. Cette méthode s'exprime très bien comme propriété sur les graphes de flot de contrôle. Comme les méthodes de complexité implicite fonctionnent à l'aide de critères purement syntaxiques, ces analyses peuvent être faites au moment de la compilation. Parce qu'elles ne sont ici que statiques, ces analyses ne sont pas toujours calculables ou facilement calculables, des compromis doivent être faits en s'autorisant des approximations. Dans le sillon du "Size-Change Principle" de C. S. Lee, N. D. Jones et A. M. Ben-Amram, beaucoup de recherches reprennent cette méthode de prédiction de terminaison par observation de l'évolution des ressources. Pour le moment, ces méthodes venant des théories de la complexité implicite ont surtout été appliquées sur des langages plus ou moins jouets. Cette thèse tend à porter ces méthodes sur de "vrais" langages de programmation en s'appliquant au niveau des représentations intermédiaires dans des compilateurs largement utilisés. Elle fournit à la communauté un outil permettant de traiter une grande quantité d'exemples et d'avoir une idée plus précise de l'expressivité réelle de ces analyses. De plus cette thèse crée un pont entre deux communautés, celle de la complexité implicite et celle de la compilation, montrant ainsi que chacune peut apporter à l'autre.

mots clés Complexité implicite, Analyses statiques, Compilateurs

Remerciements

Si on y réfléchit bien, ce travail n'est que le produit d'une multitude d'aides, de dons et d'influences. Le premier don est sûrement celui de ma mère qui a tout fait pour que "nous" - mes sœurs et moi - n'ayons rien à nous soucier d'autre que notre avenir personnel. Ensuite vient mon père qui a su semer en moi un intérêt pour les sciences et plus particulièrement l'informatique. Il y a, plus généralement, le soutien de la famille, je pense aux sista, aux cousins, oncles et tantes toujours présents pour m'encourager.

Je remercie également mes parents scientifiques: mes enseignants de l'institut Galilée qui sont devenus mes collègues par la suite. Il y a Thierry Hamon grâce à qui j'ai pu découvrir ce qu'est un laboratoire de recherche, Sylvie Borne qui m'a donné l'opportunité de m'épanouir via une autre passion: l'enseignement. L'équipe LCR ou "LoVe" qui m'a accueilli chaleureusement dès le premier jour, malgré mon retard conséquent le 3 Novembre 2014... Merci Virgile pour tes encouragements et ton aide indispensable. Tak Jakob for din velkomst og din sympati. Merci Vincent Danjean, Basile Starynkevitch, Lucas Saiu et Lars Kristiansen. Merci Thomas Seiller pour la clarté que tu as apporté à ma recherche. Et finalement Jean-Yves, tu as été mon premier professeur d'informatique en 2008, il y a 9 ans... je te remercie, au nom de la CPES 2010 et de toute la promotion ingénieur 2013, pour ta pédagogie. Tu nous as fait découvrir l'algorithmique par le jeu. Puis, en tant qu'encadrant tu as su maintenir cette flamme de motivation et de confiance en moi tout au long de ces 3 années. Tu as toujours trouvé les bons mots aux bons moments. Tu m'as montré ta vision de la recherche finalement encore comme une énigme à résoudre, un éléphant à découvrir[1], un jeu collaboratif où on ne peut gagner seul: faire progresser les autres est finalement le principal. Également grâce à toi j'ai finalement pu sortir de ma banlieue Parisienne et j'ai découvert Copenhague, une ville chaleureuse (au sens figuré!) à échelle humaine.

[1] Parabole des "aveugles et de l'éléphant"

Il y a aussi ceux qui m'ont accueilli à bras ouverts aux moments les plus difficiles, je pense aux amis de Copenhague: Noy, le Grønjordskollegiet, les amis du Bastard Café (the gamers), les membres du labo (the HCC team) et surtout "the Dude" qui, en plus de me laisser gagner aux échecs de temps à autre, m'a aidé sur tous les plans lors de mon séjour au Danemark. Thanks Dude ☺

Je remercie mes amis de longue date (les poteaux du Lycée toujours prêts à faire des centaines de kilomètre pour moi, la promo ING 2013 toujours chauds pour une petite dernière et l'inoubliable promo PLS 2014) qui m'ont apporté un soutien psychologique et surtout beaucoup de divertissements pendants ces longues années de formation et de recherche scientifique et personnelle.

Big up à la B103 et tous les doctorants de Paris 13! Merci Antoine d'avoir nourri tous les doctorants de ce bureau. Merci Ivan d'avoir retapé ce banc auquel je tiens beaucoup! Merci Pipo pour ta sagesse et ton influence de gluten-free-data-scientist-bobo-punk-apple-addict-archlinuxien-du-16ème. Merci au cosmonaute Luc pour ton aide et tes meilleurs découvertes de ce monde, je pense au warpigs, Gillette et 20 Fingers, Martine Clémenceau, François Valéry et ces autres choses des vrais vivants.

Il y a aussi plus récemment les coloc' de Grenoble (qui ne m'ont pas beaucoup vu sortir de ma chambre le premier mois… rédaction oblige!) qui offrent une superbe ambiance de vie commune. Merci au VimTex Master Hamza pour tes conseils vim et ton accueil au Verimag.

J'aimerais terminer par deux personnes qui ont eu un rôle important dans cette histoire: Lison dont les influences m'ont aidé à faire les bons choix aux bons moments. Alice sans qui je n'aurais sûrement pas pu finir cette thèse à temps… ou du tout?

Contents

Introduction

Context

Let's play chess! Chess is a turn based board game which opposes two opponents who are moving pieces in respect to some rules. At each turn, a player has a finite number of possible moves offering a finite number of new possible moves and so on. The number of possible *sequences* of moves grows exponentially. This means that we need *a lot* of time to analyse all of them and possibly choose the best or maybe just avoid the worst sequences. Morality: "exponential is bad" - as my first algorithmic professor used to say. It is worth being aware that, *at a human scale*, it is impossible to plan every move of a chess game. At least it will prevent us from losing time by trying, even more if it is a blitz game or if the rules are changing daily[2]...

Complexity of programs

The study of complexity theory helps us predict and control resources (usually time and space) consumed by programs. It extends to the search for the minimal amount of resources needed to solve a given problem. It is relatively easy to compute the complexity of an algorithm but there are many different ways to solve a problem, meaning many different programs that compute the same function, each with a potential different complexity. For instance, sorting a list of n elements can be done in time $O(n^2)$ with the *insertion sort* algorithm but in time $O(n \times log(n))$ with a *merge sort* one. Figure 1 gives an idea of the growth speed of both functions.

Since there are infinitely many programs solving the same problem (or computing the same function), finding the minimal complexity corresponding to a problem is difficult. In one hand, we can show upper bounds on the complexity of functions: the existence of the insertion sort algorithm shows that the complexity of sorting

[2] 59 quintillion was the number of possible ENIGMA machine settings allied needed to test daily during the World War II

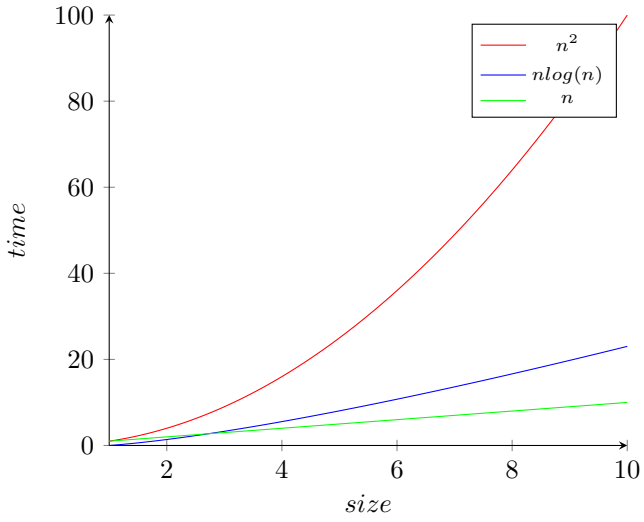


Figure 1: Average run times.

is at most $O(n^2)$. On the other hand, lower bounds on the complexity of functions are usually extremely hard to prove.

Knowing the complexity of a program is obviously interesting for two main reasons:

1. it helps determine efficient and non efficient parts of the program.

2. it helps predict the amount of resources needed regarding the potential inputs.

Those are two very important issues: the first is about detecting weaknesses and potential optimizations, the second is more about security and safety of programs. Safeness is important for embedded systems such as autonomous systems with restricted resources (i.e satellites). It is also important for security: an exaggerated[3] example could be the well known *RSA* cryptographic system which is related to the *integer factorization* problem. This problem is presumed "hard" to solve, indeed no *efficient* algorithm has been found. When we say "efficient" it means able to solve it in polynomial time. But it has never been proved that such an algorithm does not exist. Yes, most of communication systems said "secure" are based on a *presumption* that the lower bound complexity level of this problem is not *polynomial*.

While computing the complexity of a given program is not as hard as finding lower bound for a given problem, a "real life" program is not an easy task, even less when the program is complicated or uses some

[3]It is exaggerated because a proof of P=NP would just mean that no scheme can be provably secure according to the accepted definitions of security, `https://crypto.stackexchange.com/questions/8891/p-np-and-current-cryptographic-systems`

advanced features of programming languages. Complexity is formally defined with abstract computational models (e.g. Turing Machines) but because it is in a too low level language, they are not widely used for actual programs. This is why actual programs are hard to analyse and this is why communities try to develop tools which are able to perform a clever analysis of actual programs, determine their complexity, and maybe even analyse the function computed by the program and suggest some optimizations or drastic rewrites to improve its complexity.

In most of the cases, predicting the amount of resources needed by a program for its execution is nearly impossible. In Real-time systems or cloud computing, the execution environment is so unpredictable that the only viable solution remains simulation, testing and monitoring, with all their drawbacks.

Static analyses are made without executing the program, so we cannot know which path of execution the program will compute exactly (see section 1.2). Here, *Static Complexity Analysis* is the best alternative we have to anticipate performance. The price to pay is precision.

20 years of Implicit Computational Complexity

The branch of research which study the theory and practice of static complexity analysis as described above goes under the name of *Implicit Computational Complexity* (ICC). The field of Implicit Computational Complexity was initiated by the early works of Cobham on Bounded Recursion [Cobham, 1962], and the breakthrough of Bellantoni and Cook on Safe Recursion [Bellantoni and Cook, 1992].

These works designed restricted programming languages in which it is only possible to write programs that run in polynomial time. The characterisation of polynomial time complexity they are based on are implicit in two senses: firstly, there is no explicit reference to an actual computational model or other kind of machinery and the language they use is (arguably) somewhat similar to actual programming languages; secondly, in the case of Safe Recursion, there is no reference to an explicit bound on programs, the syntax is (strongly) constrained but no bound needs to be provided.

That way, the programmer does not need to know about complexity anymore, and the system can nonetheless provide bounds. Of course, the drawback is that the language is too restricted. In many cases, a programmer can very well write a polytime program which is not part of the language. The system thus rejects it and the program needs to be entirely rewritten. Since finding the complexity of any program is an undecidable problem,

we cannot entirely avoid these "false negatives". However, the early ICC languages tend to be too restricted for practical use. In the years since the breakthrough of Safe Recursion, many other ICC systems have been created. A great focus has been put on getting more and more expressivity, that is, being able to accurately analyse as many programs as possible, and, hopefully, as many actual programs in actual programming languages as possible.

More generally, ICC aims at finding syntactic criteria on programs that guarantee some semantic property. Since then, many different directions have been studied in ICC. The main ideas revolve around following data flow (the *Tiering* of Leivant and Marion [Leivant and Marion, 1995], the *Size Change Termination* of Lee, Jones and Ben-Amram [Lee et al., 2001], the *Non-Size Increasing* programs of Hofmann [Hofmann, 1999a], ...), performing a static check on values (the *Quasi-interpretations* of Bonfante, Marion and Moyen [Bonfante et al., 2011], the mwp-polynomials of Kristiansen and Jones [Kristiansen and Jones, 2009], ...) or enforcing a strict type checking (variations on Girard's Linear Logic [Girard, 1987] such as Baillot and Terui's DLAL [Baillot and Terui, 2009]). Schöpp introduced a more restricted Bounded Linear Logic: the Stratified Bounded Affine Logic [Schopp, 2007]. Hofmann and Jost [Hofmann and Jost, 2003] furnish upper bounds on the heap usage in functional programming by accepting some restrictions.

Applications

Logical methods have been applied to static complexity analysis with considerable success. Today, we have implicit characterizations of several remarkable complexity classes, both of practical interest (constant space, L, NC, P) and of theoretical interest (PSPACE, ELEMENTARY). However, in spite of the variety of tools used to obtain these characterizations (structural proof theory, linear logic, type systems, quasi-interpretations...), the range of languages considered is still quite limited, and essentially confined to first-order term rewriting systems and functional languages (which, in turn, all boil down to some form of typed-calculus). Moreover, even if we restrict ourselves within these programming paradigms, there is a problem of expressiveness.

We may resume the situation with Figure 2. On the vertical axis, we have a higher and higher degree of expressiveness, i.e., we are closer and closer to intensional completeness. On the horizontal axis, we have richer and richer programming paradigms: at the low end we have first-order term rewriting, followed by

higher-order rewriting and functional programming (λ-calculus), then we may think of functional languages with imperative features (memory allocation, side effects, etc.), then non-deterministic and/or probabilistic languages, and we may put languages exhibiting parallelism and concurrent features at the high end. The dots in the picture could represent our present achievements: a satisfactory level of expressiveness is obtained only for the poorest paradigms (first-order term rewriting), and as we move along the horizontal axis, we either have poor expressiveness or no result at all. The dashed curve represents the decidability barrier: it is natural to picture it as a hyperbolic slope, because intensional completeness becomes harder as richness increases.

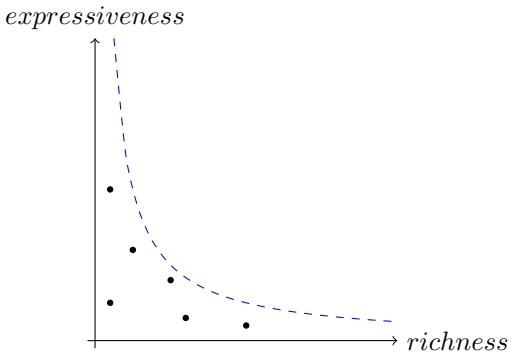


Figure 2: Expressiveness versus richness.

On "toy" languages

As said previously, most of the mentioned results usually concern what we will naively call "toy" languages (because not widely used on real programs) such as Term Rewriting Systems, λ-calculus or the LOOP language.

Term Rewrite Systems

([Avanzini, 2013] and others) use Term Rewrite Systems (TRS for short) machine model which is close to first order functional programs. A term rewrite system (which build a program) consists of a collection of directed equations, so called rewrite rules. Computation in this model is performed by successively applying equations from left to right.

Term rewriting forms a Turing complete model of computation, hence fundamental questions as for example termination of a given TRS, are undecidable in general. But TRS is a powerful abstract model for cost analysis:

```
fibo(n) ->
   if n <= 1
      then n
      else fibo(n-1)+fibo(n-2)
```

Figure 3: Fibonacci function in TRS.

it is easier to guess the relation between input's size and the number of rewrite steps performed. This is why a lot of work on TRS concern Termination analysis which provides upper bound on the maximal length of reductions.

I can now quickly introduce the main ideas of Size-Change Principle (SCP). SCP studies termination on TRSs with the following principle: if the reduction of a term is infinite, an infinity of functions call is made. If we study all the possible sequences or chains of functions and show that none of them can be infinite then we can show termination of this TRS. Resource Control Graph is one of the tool used for "SCP like" methods.

Resource Control Graphs *Resource Control Graphs* (RCG) is a somewhat generic framework in which to express several different ICC analysis. This framework will be reused and customised all along this thesis. This analysis is strongly based on annotated control graphs for the program, and the kind of annotations, as well as the way to combine them, allows to study various properties of programs. Each state of the program is abstracted by its size, and each instruction is abstracted by the effects it has on the state size whenever it is executed. The abstractions of instruction effects are then used as weights on the arcs of a program's Control Flow Graph.

Let's give a simple example, left-hand side of Figure 4 is a *reverse* function written with a stack machine. If we add a weight, corresponding here to the space used by the program with the `pop` and `push` calls, to the Control Flow Graph we obtain the Resource Control Graph shown on the right.

Termination is proved by finding decreases in a well-founded order on state-size, in line with other termination analyses, resulting in proofs similar in spirit to those produced by Size Change Termination analysis. However, the size of states may also be used to measure the amount of space consumed by the program at each point of execution. This leads to an alternative characterisation of the Non Size Increasing programs, section 1.4. This new tool, which is one of our contribution, is able to encompass several existing analyses and similarities with other studies suggesting that even more analyses might be expressable in this framework, thus giving hopes for a generic tool for studying programs.

```
0: if l = [] then goto 4;
1: h:=pop(l);
2: push(h,acc);
3: goto 0;
4: end;
```

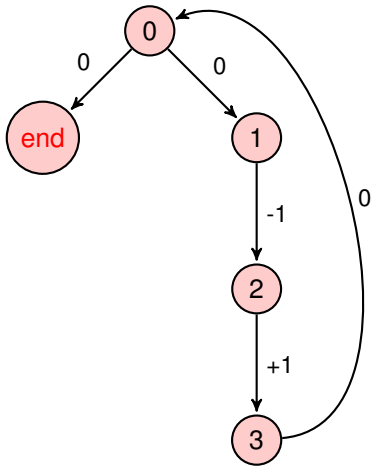


Figure 4: RCG overview (explained in).

Many such systems have been designed for various programming paradigms, making it extremely difficult to compare them efficiently. Yet, they often seem to rely on the same techniques adapted in various ways to cope with the idiosyncrasies of the programming paradigms. RCG is an attempt to take a step back and use a common language to describe ICC as a single beast.

Among powerful methods to prove termination like in [Hofbauer and Lautemann, 1989, Bonfante et al., 2001, Moser, 2009] we can find [Moyen, 2001] who presents ICAR, a tool able to analyses the implicit complexity of TRS using RCG. Termination is checked using "Path Ordering" and complexity upper bound can be determined with Quasi-Interpretations.

Quasi-Interpretations Quasi-interpretations (QI) can be considered as a static analysis methodology for inferring asymptotic resource bounds for first-order functional programs written as constructor term rewriting systems. Used with termination orderings, they allow to define various criteria to guarantee either space or time complexity bounds [Bonfante et al., 2004, Bonfante et al., 2005a, Bonfante et al., 2005b, Baillot et al., 2006, Bonfante et al., 2011].

Comparatively to *Safe Recursion*, the main advances are of two kinds. Firstly, instead of designing a restricted language, QI work as an analysis on a Turing complete language. Programmers do not need to struggle to write their programs any more. Obviously, the system will thus not be able to analyse everything, but it is

still possible to run a program which does not have a QI. Thus, some parts of the program can be correctly analysed while some are still without guarantees. It is then up to the programmer to either try and rewrite those parts (and only those parts), or run the program as is (for example, because an external, handcrafted proof is provided). Secondly, the analysis is implicit in an additional sense: the implicit complexity of the program. In several cases, the given program may run in exponential time but QI nonetheless allow to detect that it would run in polynomial time after some transformations. Thus, instead of analysing the complexity of the program itself(explicit complexity), it tries to analyse the complexity of the function (implicit complexity). This is obviously not perfect (in many cases, the implicit complexity is not detected) but still works on a huge class of programs, namely all the programs that would be polynomial time if some Dynamic Programming technique were used. Quasi-Interpretations have been reused in several other works and are now a well-known tool in the ICC community.

λ-calculus

The ICC community uses λ-calculus for its types disciplines. [Baillot and Terui, 2009] presented a polymorphic type system for lambda calculus ensuring that well-typed programs can be executed in polynomial time called *dual light affine logic* (DLAL). DLAL ensures good properties on lambda-terms: a well-typed term admits a polynomial bound on the length of any of its beta reduction sequences.

Similarly to SCT, [Abel and Altenkirch, 2002] introduced a λ-calculus language on which they implemented a syntactical analysis able to ensure that recursive calls appear only with arguments structurally smaller than the input parameters.

[Laird et al., 2013] compare typed λ-calculus programs regarding to how they compute something (how many steps, in how many ways or with what probability).

By the Curry-Howard correspondence, proofs can be seen as programs. Coupling with *Linear Logic* it provides interesting approaches on resources analysis. [Hofmann and Jost, 2003] wish to extend this setting to verification of quantitative properties, such as time or space complexity bounds.

Others

[Hofmann and Schopp, 2009] use pointers programs as abstraction for LOGSPACE-algorithms and show that pointers programs with counting can decide undirected s-t-reachability problem.

Even if such languages do have a strong utility in Theoretical Computer Science, they are not daily used by programmers. On the other hand, actual languages use much more constructions (*e.g.* objects, pattern matching, exceptions, etc...) which make analyses complicated. Thus, even after 25 years of ICC, it is not possible today to apply its techniques on actual programs. We start filling the gap.

On "real" languages

Here we naively dub "real" languages, those which are widely used. Most of them are designed to combine rich programming paradigms and to give the maximum freedom to the developers. Often, the whole is coated by a comfortable syntactic sugar. The following is a really non-exhaustive list of what have been done in the field of complexity analysis over widely used languages. Most of them search for *WCET* or worst-case execution time and try to approximate number of iterations a loop can perform and provide constraints around the program flow. To compare with ICC theories, where number of iteration of a loop is often decidedly unpredictable and where all cases are considered because it is "negligible", here it is really more refined.

First, we can distinguish two kind of projects: those which need user annotations (loop bounds or ranking function) and those which not. Here we will focus on the second kind.

[Alias et al., 2010] prove termination of *flowchart programs* by observing decreasing behaviours of computed *ranking functions* from program states. Their tool, dubbed *RANK* [Alias et al., 2013], translates C programs into integer interpreted automaton. This tool tries to provide counterexamples in the case the program may not terminate. In the other case, it computes the *worst-case computational complexity* of the program. In [Gonnord et al., 2015] they improved this work and implemented a prototype called *Termite* which uses LLVM to provide SSA intermediate representation (subsection 1.1.2) and a static analyzer working over the LLVM infrastructure: *PAGAI* [Henry et al., 2012] to compute invariants in a way to produce transition relations.

[Giesl et al., 2017, Spoto et al., 2010, Falke et al., 2011] translate *Java Bytecode* or C programs into TRSs or constraint logic programs and then use techniques like above to prove termination. The model checker Terminator [Cook et al., 2006] is used to provide termination proof or counterexamples too.

In [Ströder et al., 2014], an abstract domain is provided for tracking allocated memory at compile time on an intermediate representation. By focusing on memory safety, it prevents undefined behaviour including non-termination. They use Symbolic Execution Graphs to approximate all possible runs and extract an *Integer Transition System* on what it is possible to prove termination.

[Albert et al., 2008b] presented COSTA which mainly provides cost of Java bytecode programs as a function of their input data size at compile time.

This is the oldest work I mention here and it is only 9 years old. Most of them deal with compiler but none was really officially integrated in a common one. Termination and complexity analyses are still considered inefficient to be part of common compilers but those works claim that it not should be.

Motivations

Often, programs comes without warranty. You already read it many times in most of program's license, here in the `GNU GPLv3`:

```
15. Disclaimer of Warranty.

  THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY
APPLICABLE LAW.  EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT
HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY
OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM
IS WITH YOU.  SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF
ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
```

This is not surprising. Because most of real-world programming languages are Turing-complete, most of interesting properties such as termination are undecidable. One way to bring reliability on your program is to test it. Making it exercise manually or automatically can help us to figure out if it satisfies or not some specified properties in some cases or inputs. But, as said by Edsger W.Dijkstra: "Testing can show the presence of errors, but not their absence."

Tests do not *prove* anything therefore do not bring warranty. *Static Analysis* try to do so but often with approximations (see section 1.2). From those approximations we get bounds which sometimes are sufficient to insure the targeted property.

Expanding Logical Ideas for Complexity Analysis

Expanding Logical Ideas for Complexity Analysis (ELICA) is an ANR research project in computer science which started in February 2015. The aim of the ELICA project is to develop logical methods for static complexity analysis, improve their expressiveness and extend their application to non-deterministic and concurrent programming paradigms. Logic-based tools, in particular of proof theoretic origin, have proved to be of capital importance for static complexity analysis. Criteria guaranteeing complexity bounds have been formulated (some by prospective participants to the ELICA project) using structural restrictions of the λ-calculus, subsystems of linear logic in which cut-elimination has a limited complexity, type systems for functional languages, and semantic methods (polynomial interpretations and denotational semantics). These methods do not give precise bounds on the complexity of programs, but they have the advantage of being modular and of producing easily verifiable certificates. However, these methods currently apply only to a restricted range of programming languages, mostly of functional nature. Even in the functional case, the criteria are unsatisfactory in terms of expressiveness, i.e., with respect to the number of practical programs to which they can be applied. The ELICA project fixes as objective the development of existant logical methods and the introduction of new ones, in order to:

- improve the expressiveness of static complexity analysis criteria for first-order functional languages, but above all extend the scope of such criteria to higher-order complexity (in particular, type-2 complexity), which is still largely unexplored;
- increase the practical impact of these methods by defining complexity criteria for languages with imperative features, and relate them to the criteria for the verification of security properties (such as non-interference);
- apply these methods to non-deterministic and probabilistic languages, to analyze the complexity of programs implementing randomized algorithms, with the long-term goal of employing such methods for the analysis of cryptographic and security protocols;
- extend the scope of these methods to the analysis of concurrent systems. Here, the problem is above all of foundational nature, because there currently is no general definition of computational complexity for concurrent processes. ELICA therefore propose to first develop a theory of computational complexity of

interactive behaviors (generalizing problems and functions), capable of giving a formal content to notions such as answer time to a request, memory space, number of active processes, etc. Subsequently, ELICA studies the application of logical methods to this theory.

Push ICC theories into the "real world"

We've seen that ICC deals with syntactic criterion but mainly on toy languages. As mentioned in the ELICA's motivations, it is time to bring those theories into "real life" programming languages.

This would offer the possibility to enlarge the spectrum of programs able to be analysed and maybe offer statistics able to give an accurate idea of the real expressivity of the methods developed in ICC. Since ICC methods use redefined (simpler) programming language (often Turing-incomplete) in order to have sound and complete answers, here we need to deal with Turing-complete languages and often relax the correctness notion: a sound "yes" answer is sometimes sufficient. But how much can we juggle with correctness notion and simplification of the language in real life programs?

Dream of "Complexity-certifying Compiler"

Real-world languages come with real-world compilers. Widely used compilers are usually focused on optimisations. Indeed, the goal is to produce an efficient code in order to have a fast program.

First, an analysis needs to be efficient and useful to be embedded in "real-world" compilers. But, regarding to most developers, safety is not often relevant for every days programs. It is not critical if your mail manager application crashes, it restarts and that's all. Then security questions can come over and here again a solution much more easier than complex analysis is to use sandboxing and virtual environments. But we can observe a rise of interest in certification of programs. One of the most promising project is *Compcert*, a formally verified optimizing compiler of C programming language which target different architecture. It is specified, programmed and proved in *Coq* a theorem prover written in *Ocaml*. The main particularity is that it is proven that the translations or transformations of the code during compilation preserve the semantic of the program. Furthermore, the performance is close to what can be generated by GCC at the `-O1` level. But Compcert is still too young and maybe too complex to have a huge community. This is why we will not consider Compcert as a "widely used" compiler.

ICC mostly provides analyses without much optimisation of the code. However, analyses and optimisations are not so far apart... An analysis can be used to fuel further optimisations. Typically, building the Control Flow Graph of a program is an analysis that is used for many optimisations afterwards. Providing proven bounds on the time or space usage of a program is also a *security* property. If the program provably uses a fixed amount of memory, then it will not try to perform an attack by heap or stack overflow, this warns us if the program could try to exhaust system resources. Restricting the syntax in order to enforce (some) security is similar to what Facebook does with the restricted FBJS. Since analysis is complex but verification is (usually) easy, one can imagine a compiler that will provide a *certificate* for some property on the compiled code, in a *Proof Carrying Code* paradigm [Necula, 1997]. The certificate could be checked, for example, before uploading an application to an application store for mobile devices to guarantee some safety to the user, or, at the other end, before downloading the application to the device to check if it has sufficient capacity to run it. The other goal here is to vulgarize certification and enlarge the number of programs analysed.

Next, some ICC analyses are known to also embed some program transformation in them. Notably, the *Quasi-Interpretations* method guarantees that the programs run in polynomial time *if some sort of Dynamic programming is used.* Thus, a program admitting a QI can run in exponential time but the analysis says that it will run in polynomial time after some (known) transformation. Bringing such analysis in compiler will indicate which part of the code should be transformed by which method.

Contributions

New approach for ICC research

The analysis presented in this thesis can be seen as proof of concept in order to show that ICC methods can be performed on real programming languages. This implies to work in compilers because it is where it is well-suited to be performed. We also show that, in addition to certificates of good behaviour, our analysis can bring optimizations.

Widely used compilers are communal and well designed therefore modular. This encourage contributions and it brings more and more specialists able to add their own bricks into the project. It is an opportunity for this community to bring their tools into real world languages.

More sociologically, it creates a bridge between two communities. Yes, ICC and compilers can work together and can fuel each other fruitfully. This is just a step opening the way for future works.

Bring a new framework for Data Flow Analysis

In section 2.2, we introduce a new kind of Data Flow Analysis which generalises different notions borrowed from ICC theories (Size-Change Principle, MWP-bounds). We show how this DFA framework can be customized and used for complexity analysis. But we also present a possible instance of this analysis designed in order to find quasi-invariants in loop and then propose a transformation to peel them.

For ICC

This framework could be reused in the future by the ICC community because it is designed as described in most of the methods derived from Size-Changes ideas. Globally, the framework works over a semi-ring which is the heart of the analysis. This semi-ring can be changed as desired in a way to perform the analysis we want.

For Compilers

This work may provide new optimizations through this analysis or at least bring new ideas coming from ICC. Furthermore it will popularizes a branch of computer science viewed as theoretical one till now by most of computer scientists. It shows that it is easy to implement your own customized complexity analysis using data flows.

In the same way as compilers, this framework could be the first brick of a bigger one which could bring contributions from a new community: an intersection between ICC and compiler ones.

Outline

The thesis is organized as follows:

Chapter 1 introduces the basic notions of compilation and static analysis for presenting our first implemented analysis dealing with LLVM Intermediate Representation. This analysis is a proof of concept which detect Non-Size-Increasing programs. It was the first step for me into a widely used compiler: LLVM

Chapter 2 presents our Data-Flow Approach for complexity analysis, describes formally a Data-Flow Analysis framework and tries to show how this tool could be used for all measurable properties of complexity.

Chapter 3 gives another use of this framework: the detection of loop quasi-invariants chunks. A prototype was developed which proposes to peel such chunks then optimize the transformed Intermediate Representation.

Source code. As an environment-friendly thesis, we do not provide in these pages the source code of our implementations. However, it is as much a part of this work and can be consulted on their respective github pages[4].

[4] `https://github.com/ThomasRuby`

Chapter 1

From Compilers to Implicit Computational Complexity

1.1 Compilers

Naively, a compiler translates a human-readable source code into a non-user-friendly assembly code for machines. It takes the opportunity to analyze and optimize the compiled program. All those analyses and transformations are done on more or less different *Intermediate Representations* of the program.

Part of this thesis tries to apprehend how suitable those *Intermediate Representations* (IR) are for static complexity analysis in a way to embed our work directly in compilers and cover a large number of widely used languages. Compilers come with a lot of tools working at different compilation times. They are designed to sequentially make analyses and transformations called passes.

1.1.1 Architecture and designs

The main purpose of compilers is to produce executable code for machines. Their second goal is to produce efficient programs in terms of execution time and space memory used. During the process of translation, the program takes different forms which are often easier to analyse and transform. Compilers are generally composed of three parts Figure 1.1.

- The first part (the green one, on the left) is called *front-end* and is specific to a source language. They are divided into three phases:
  - The lexical analysis (lexing) figures out the "words" or basic bricks of the program (e.g. *keyword*, *identifier*, *symbol name*)
  - The syntax analysis (parsing) provides a syntax tree (*Abstract Syntax Tree*, AST) following rules of a grammar (*Context-Free Grammars*) which defines the language's syntax.

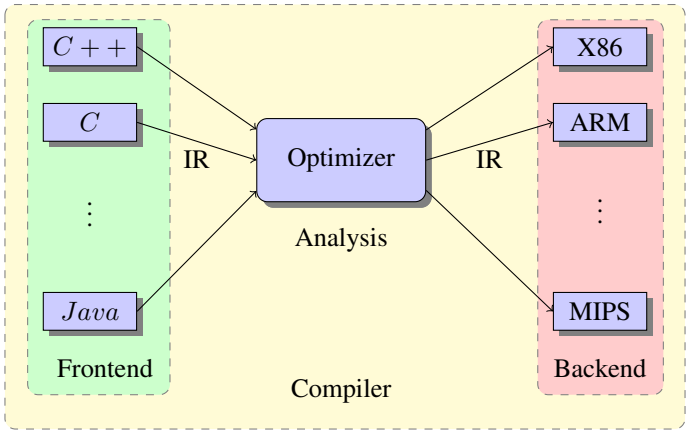


Figure 1.1: Compiler Design

  - The semantic analysis adds semantic information to the AST and builds the *symbol table* (a map which binds each symbol to a value). At this level we can find some semantic checks like *type checking*.

  From those analyses comes an intermediate representation. This translation simplifies the job of the rest of the compiler which does not want to deal with each expressivity of each programming language.

- The *middle-end* also called optimizer or *Pass Manager* provides analyses (over *data-flow*, *dependence*, *alias*, *pointer* etc.) mainly used for optimizations (e.g *dead code elimination*, *constant propagation*, *loop invariant code motion* etc.). Those transformations are supposed to produce a better (faster or smaller) IR while preserving the semantic of the program.

- The *back-end* also provides optimizations while generating the targeted assembly code. Those optimizations are said "machine dependent". This translation is also very technical because it goes from a virtual machine language to a real (mechanical) one. For instance, because moving variables in and out of memory can slow considerably the whole program, the *memory allocation* strategy needs to be optimized regarding to the *liveness* of each variable used at a time. This issue can be reduced to the problem of *K-coloring* which is *NP-complete* thus why interplays need to be found.

This three-phases design has several advantages:

- it avoids to have $n \times m$ compilers for translating n languages into m targets.
- because it supports more languages and targets, it brings more contributors (when the project is open-sourced).
- it tends to create specialists: a middle-end developer does not need back-end or front-end skills and can focus on his part.

This strategy adds steps which slow down the compilation and come with potential errors. Remember that for each step the semantic should be preserved. It also means that the quality of analysis and transformations will depend on the quality of the intermediate language and its translators.

1.1.2 Intermediate Representations

The middle end or *optimizer* (`opt` in LLVM) aims to work over the same intermediate representation in actual compilers. The main purpose of this choice is the genericity and portability it offers: by this way it is relatively easy to implement your analysis over all (sourced or targeted) languages supported by the compiler. The perfect IR is able to represent the source code and the target language without loss of information. By reading it, we should be able to guess how the corresponding source was and what the machine language will look like.

Firstly, intermediate languages were made to simplify transformations. For instance, the *three-address* form has been used for many years because of its simplicity. Decomposing all expressions into a sequence of basic simple assignments like `r0 := r1 OP r2` (where `OP` is a binary operator) helps to detect, replace, delete or add sub-expressions. It also helps for translating into an assembly language.

Another useful form is the Static Single Assignment (SSA): a program is said to be in SSA form if each of its variables is defined exactly once and each use of a variable is dominated by that variable's definition. This form can cohabit with the tree-address one and simplifies *data-flow* analyses and transformations because only a single definition can reach a particular use of a value, we will observe that more technically further. This form implies a unbounded number of virtual registers and also the apparition of φ function which is used to select an incoming value depending on which basic block the control flow came from (see how it could be a drawback in section 3.6).

A lot of Languages can be considered as "intermediate", among them we can find the *Java bytecode*, the GNU *Register Transfer Language* (RTL), the *GENERIC and GIMPLE*, the *LLVM IR* etc. Some of them are considered as *Tree Intermediate Languages* since tree structure facilitates the implementation of new analyses and optimizations closer to the source. For example, at the beginning, GCC had one front-end translating source languages to RTL (which is really low-level) and one back-end containing optimizer, code generator. The optimizer was considered part of the back-end because it provided only "low-level optimizations"[Novillo,]. Later, a *Tree Optimizer* was designed to deal with Control Flow Graphs and SSA forms given from the GIMPLE before translating into RTL as can be seen in Figure 1.3 (which is translated from the C program in Figure 1.2).

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

int main(){
   srand(time(NULL));
   int n=rand()%100;
   printf("n = %d", n);
   return 42;
}
```

Figure 1.2: Random program in C

But also variants of source languages were designed to provide their features as portable assembly language (e.g *C Intermediate Language* is used by *Frama-C*[Mantere et al., 2009] and *Compcert*[Tristan, 2009]).

More generally, we can suppose that for one specific static analysis there exists a "best" intermediate form or representation to optimally perform it. The "best" IR for compilers is a source language independent representation with the best interplay of expressivity regarding to all the analyses we want to perform.

The common point of the proposed features is the simplicity. But simplicity does not fit with richness. We can interpret that into Figure 1.4 by the translation in red. What we can do is to juggle with different representation in a way to gain certain "kind of" expressivity.

```
main ()
{
  long int D.2829;
  unsigned int D.2830;
  int D.2831;
  int D.2832;

  {
    int n;

    D.2829 = time (0B);
    D.2830 = (unsigned int) D.2829;
    srand (D.2830);
    D.2831 = rand ();
    n = D.2831 % 100;
    printf ("n = %d", n);
    D.2832 = 42;
    return D.2832;
  }
  D.2832 = 0;
  return D.2832;
}
```

(a) GIMPLE in textual form

```
...
(call_insn 27 26 28 2 (set (reg:SI 0 ax)
        (call (mem:QI (symbol_ref:DI ("printf") [flags 0x41]
<function_decl 0x7faae3afbb60 printf>) [0 __builtin_printf S1 A8])
            (const_int 0 [0]))) test2.c:8 -1
     (nil)
    (expr_list (use (reg:QI 0 ax))
        (expr_list:DI (use (reg:DI 5 di))
            (expr_list:SI (use (reg:SI 4 si))
                (nil)))))
(insn 28 27 31 2 (set (reg:SI 90 [ _10 ])
        (const_int 42 [0x2a])) test2.c:9 -1
     (nil))
(insn 31 28 35 2 (set (reg:SI 91 [ <retval> ])
        (reg:SI 90 [ _10 ])) -1
     (nil))
(insn 35 31 36 2 (set (reg/i:SI 0 ax)
        (reg:SI 91 [ <retval> ])) test2.c:10 -1
     (nil))
(insn 36 35 0 2 (use (reg/i:SI 0 ax)) test2.c:10 -1
     (nil))
...
```

(b) A part of RTL in lisp-like form

originally 130 lines

Figure 1.3: GCC Translations

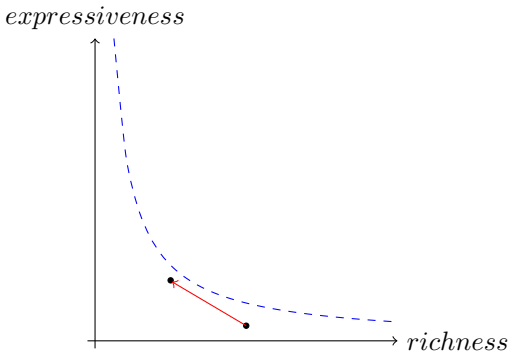


Figure 1.4: Translation to intermediate languages.

Most of the intermediate representations are pretty similar to an assembly language. It is a lower-level programming language than the input one but it is a higher-level than real assembly language. For instance type can be preserved without influencing analysis. The LLVM IR comes with high-level type information and polymorphic instructions, for instance the `add` instruction can operate over several different types of operands. This reduce the set of opcodes then simplify the language.

1.2 Static Analysis

"Program analysis offers static compile-time techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically at run-time when executing a program on a computer."[Nielson et al., 1999] Only approximations because, most real-world programming languages are Turing-complete so any non-trivial property is undecidable. A naive example could be the program in Figure 1.5.

$x := read();$
$y := 1;$
`if` $x > 0$ `do` :
  $y := 0;$
  $f();$
$z := y;$

Figure 1.5: A naive example

With f a pure function. Intuitively, the possible values of y that can reaches the z assignment will be 1 or 0. But maybe the right answer is just 1 because it is possible that f does not terminate and the execution flow never reaches the $z := y;$ statement. Static Analysis can not determine if a function terminates or not, this is why the analysis will answer a larger set of possible values for z. We also would like to have the smallest set as possible: the best safe approximation.

We can answer to different questions and ensure different properties regarding to this safe approximation, at least more easily than with tests. Testing is too much focused on specific cases. It takes forever to cover all possibilities and, over all, it does not prove anything. As mentioned before: "Testing can show the presence of errors, but not their absence."

Static Analysis does not solve undecidable problem but wrap or confine it the better it can. An over-approximate analysis is said to be sound when it covers all targeted behaviours. In a way to have the best approximation regarding to the targeted property, the analysis needs a specific abstraction.

1.2.1 Approximation

Static program analysis is performed without actually executing the analyzed program, it is opposed to dynamic analysis. At this stage we don't have inputs, then we can't know what will be the accurate behavior of the

program as languages permit infinitely many executions. We can only study all the possibilities or paths that might be executed, then cover the entire program. Static analysis is mainly used to detect errors, dead code, variables unused etc... therefore to prevent bugs or find possible optimizations.

An idea is to map real world into an *abstract* world easier to analyse or express. Regarding to a specific problem it may be possible to map the concrete semantics to an abstract and finite one which is statically analysable. This new abstraction needs to keep enough information to answer the specific problem. To give an example, here is an enigma (given to me by a good friend Anupam Das):

```
There is a village of wizards and a village of dwarves. Once a year,
the wizards go over to the village of dwarves and line all the dwarves
up in increasing height order, such that each dwarf can only see the
dwarves smaller than himself. The wizards have an infinite supply of
white and black hats. They place either a white or black hat on the
head of each dwarf. Then, starting with the tallest dwarf (in the back
of the line), they ask each what color hat he is wearing. If the dwarf
answers incorrectly, the wizards kill him (the other dwarves can hear
his answer, but can't tell if he was killed or not). What strategy can
the dwarves use to minimize the number of dwarves that are killed?
What is the most number of dwarves that will be killed using this
optimal strategy?
```

The answer to the last question is 1, only the first dwarf may be sacrificed (*if the reader wants to find how by itself it is pleased to jump to subsection 1.2.2*).

The key of the answer is to use parity. Just knowing the parity of black hats is sufficient for the rest of the dwarfs: they use the number of black hats in front and behind to deduce which hat they have on their head. They just need to agree in the meaning of the data (here a simple bit) given by the first poor dwarf (for instance saying "black" means there is an odd number of black hats and "white" an even one). This abstraction allows to give enough information about all hats to solve the problem. Finally the real trick is to know what is the minimum information I need and how can I map it into the smallest data structure (here parity into a bit).

This kind of *Abstract interpretation* is basically what is used in static analysis. "Abstract interpretation allows the compiler to answer questions which do not need full knowledge of program executions or which tolerate an imprecise answer (e.g. partial correctness proofs of programs ignoring the termination problems, type checking, program optimizations which are not carried in the absence of certainty about their feasibility,...)" [Cousot and Cousot, 1977].

1.2.2 Data-Flow Analysis

Programs can be seen as a set of nodes (Basic Blocks) connected by directed edges. The whole compose the *Control Flow Graph* of the program (section 1.4.3. There is one entry node and there may be several exits. Each node takes values as input and can create/delete/modify them. Given a special set of values to the entry node, it creates states. A state is a set of current values (or environment) associated with the current node information (control).

This can quickly creates an infinite number of states because of the infinite possible inputs and the potential loops the connected nodes provide. *Abstract interpretations* come over those states thus allow convergent static analysis gathering information about the program's *data-flow*. We can generalise by saying that all *Data-Flow Analyses* aim to solve data-flow equations and the common way to do it is by using iterative algorithms over the nodes. Those algorithms are designed to work on an abstract domain in a way to converge to a *fixpoint*. For instance, in section 2.2.2 our analysis can only add dependencies between variables and this operation is monotonic, furthermore the number of variable is finite then it converges to a fixpoint.

In the naive example Figure 1.6, we found that z could have two possible values. Because the *control flow* of the program allow us to do a *forward analysis* and calculates the set of definitions of y that potentially influence the current assignment on z.

$x := read();$
$y := 1;$
`if` $x > 0$ `do` :
  $y := 0;$
  $f();$
$z := y;$

Figure 1.6: Forward analysis

1.2.3 Sensitivities

Previously we said that regarding to the data-flow equation the analysis needs more or less information. Here are different sensitivity a data-flow analysis can have:

- Flow-sensitiveness takes into account the order of the statements: dominance information is always up to date in compilers. An example of flow-insensitive analysis could be (static) type analysis where a variable needs to be well typed at all program points.

- Path-sensitiveness considers all facts that happened on a taken path execution, when it is possible, this give more precise or less approximated results. But here exponentially many paths could make it difficult to perform such analysis, this is why path-sensitive analyses are combined with analyses of feasibility of paths.

- Context-sensitiveness uses call graph to induce calling relationships to have a better precision on the behaviour of functions. This approximates the call stack regarding to a specific depth because each call generate a different context and recursivity may generate infinite contexts.

1.2.4 Trade-offs in Static Analysis

Three main parameters come out when comparing static analyses: soundness, precision (or here more specifically the recall) and performance.

- Soundness for static analysis is, as said previously, when the analysis is able to capture only the targeted behaviours of the program. For instance, a sound analysis which says the program is bug-free means that it is truly bug-free but when the analysis reports a bug it means that there is potentially a bug, it may be a false negative.

- The recall refers to its ability to capture the maximum amount of cases guaranteeing the targeted behaviour regarding to the total targeted behaviour cases. It is a way to have the fewest false negative as possible.

- The performance is the amount of resources, again time and memory usage, spent by the analysis to reach the fixpoint. Surprisingly, in the field of static analysis, asymptotic bounds are not often used. It may be because analyser designers don't expect the nesting depth to increase as the program length increases if it is a human who writes it. For example a program may have a million lines of code, but it will never have a thousand nested if-then-else statements. It is a kind of gamble that in fact the nesting depth is a small constant and the worst-case is never encountered for real programs.

The challenge in the design and implementation of static analysis is to achieve soundness with a trade-off between recall and performance.

1.3 A strategic choice: LLVM

In our time, two mainly used compilers exist: GCC and LLVM. For our first prototype, our choice was LLVM because: first of all, LLVM is well documented; the community is huge and very active; it uses the same *Intermediate Representation* throughout the compilation; it is modular; it can use GCC's front-ends; it is more and more used. For instance, more, and more efforts have been done to build Debian with Clang[1]. By comparison, GCC remains more used but performances and accessibility are equivalents. However the LLVM community's documentation and help are more appropriate. The modularity also helps to contribute without knowing the entire working flow. The analyses are, of course, feasible in GCC, Compcert[2], etc.

The LLVM Project [Lattner and Adve, 2004] is a collection of modular and reusable compiler and tool chain technologies. LLVM is an acronym for Low-Level Virtual Machine, but the scope of the project is not limited to the creation of virtual machines. As the scope of LLVM grew, it became an umbrella project that included a variety of other compiler and low-level tool technologies as well.

1.3.1 Architecture

LLVM was firstly designed for optimizations at *compile-time*, *link-time*, *run-time* and *idle-time* (between runs using profiling). The strategy is to reuse the same "LLVM code" at each time and refine it. The main idea is based on the belief that many "high-level" optimizations are not really language-dependent, and are often special cases of more general optimizations that may be performed on the IR (e.g. both virtual function resolution for object-oriented languages and tail-recursion elimination for functional languages can be done at this level). All others "language-specific optimizations" must be performed in the front-end.

Now LLVM is a toolchain with many components (e.g. assemblers, compilers, debuggers, etc.) designed to be compatible with existing tools. The large goal of this tool chain seems to be the search of the best modularity to oppose to "single monolithic specialized-purpose tools".

It has been designed to work with GCC, it uses GCC's front-ends and produce an LLVM-IR to run the LLVM Optimizer and Code Generator on it. Also the LLVM Link Time Optimizer can be used over object files generated. It results a faster compile and execution time[3].

[1] http://clang.debian.net/
[2] compcert.inria.fr/compcert-C.html
[3] http://llvm.org/pubs/2008-10-04-ACAT-LLVM-Intro.html

1.3.2 Program Representation

The LLVM *Intermediate Representation* is a Typed Assembly Language (TAL) and a Static Single Assignment (SSA) based representation which provides type safety, low-level operations, flexibility and capability to represent any high-level languages cleanly. As we said, this representation is used throughout all phases of the compilation in LLVM.

The LLVM Intermediate Representation is source-language-independent, mainly because it uses a low-level instruction set slightly richer than assembly languages, it is a RISC-like virtual instruction set. The instruction set consists of 31 opcodes, just enough to not loose type expressivity but still a *low-level* representation. Most of these operations are in a *three-address* form (recall: they take one or two operands and produce one result).

But, unlike RISC instructions, LLVM-IR is strictly typed, thus type mismatch can easily be detected. Types can be primitive or constructive (composed by several primitive types or constructive types). Each instruction has restrictions on the arguments types but they can be polymorphic: for instance `add` can operate on different types, this widely reduces the number of opcodes.

Because it is typed and the function conventions are abstracted through basic instructions like `call`, `declare`, `define` and `ret` with explicit arguments, the LLVM-IR is user-friendly. For example Figure 1.7 shows the LLVM-IR generated from the C program in Figure 1.2.

We can easily see functions, arguments and types. For instance the global variable which contains the printed string has a composed type of `[7 x i8]` aka a table of 7 characters.

But what we see here is one of the three isomorphic form this IR can have. The format above is the textual one, an other is a data-structure in-memory and the last one is a dense on-disk binary "bitcode". Tools are provided to translate one format to the other.

A lot of well known optimizations are dealing with this IR: Dead Code Elimination, Loop Invariant Code Motion, Constant Propagation etc. For example: The Instruction Combination Pass is one of the simplest passes. It knows some optimizable patterns like `add X,0` is substituted in `X` or `xor X,X` in `0` etc. can detect and replace them. Some optimizations are really specific to certain computation (e.g. detection of $\sum_{i=1}^{n} i$ is also made by finding a specific pattern).

LLVM has already shown that an efficient low-level representation enriched with type information can support high-level analyses and transformations [Tavares et al., 2014].

```
...
@.str = private unnamed_addr constant [7 x i8] c"n = %d\00", align 1

; Function Attrs: nounwind uwtable
define i32 @main() #0 {
  %1 = tail call i64 @time(i64* null) #2
  %2 = trunc i64 %1 to i32
  tail call void @srand(i32 %2) #2
  %3 = tail call i32 @rand() #2
  %4 = srem i32 %3, 100
  %5 = tail call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([7 x i8], ↩
      [7 x i8]* @.str, i64 0, i64 0), i32 %4)
  ret i32 42
}

; Function Attrs: nounwind
declare void @srand(i32) #1

; Function Attrs: nounwind
declare i64 @time(i64*) #1

; Function Attrs: nounwind
declare i32 @rand() #1

; Function Attrs: nounwind
declare i32 @printf(i8* nocapture readonly, ...) #1
...
```

Figure 1.7: LLVM Intermediate Representation

1.3.3 Optimizer and Data Structure Analysis

The *optimizer* is one of the several tools or modules provided by LLVM. Designed for more modularity, the optimizations are built into distinct libraries and the LLVM *Intermediate Representation* is preserved permanently, making it simpler for other-ends to use them.

This optimizer is mainly composed by a *Pass Manager* that keeps analyses information up to date, manages memory used, enforces enabled passes with a given order and make pass developer's life simple thanks to its modularity. These passes visit and change the *Intermediate Representation* in the middle-end.

Analyses information comes to enrich the in-memory data-structure which lives during the compilation. For instance, Definition-Use chains, which consist to link definition to all the uses for each variable, are preserved and updated each time modifications are performed on the IR. We can guess that, regarding to what can be modified by which pass, it is worth to delay or foresee some analyses to avoid doing it after each single transformation. Passes are expected to stand on their own or declare their dependencies among other passes if they

depend of some other analyses. When a series of passes is given, the PassManager uses the explicit dependency information to satisfy these dependencies and optimize the execution of passes.

Options can be sent to the optimizer, the most known option is the optimization level. The level `-O0` is without any optimization (aka the IR produced by the front-end). The `-O2` enable most of fast and not aggressive optimizations. `-Os` and `-Oz` are like `-O2` but aim to reduce the size of the code (by disabling unrolling for instance). `-O3` uses slow and more aggressive optimizations and does not care about the size of the generated code. Those options enable passes but also order them. We can see how by reading the `PassManagerBuilder` class which set up "standard" optimization sequence for C and C++.

```
Pass Arguments: -tti -targetlibinfo -tbaa -scoped-noalias -assumption-cache-tracker
     -profile-summary-info -forceattrs -inferattrs -ipsccp -globalopt -domtree -mem2reg -deadargelim
     -domtree -basicaa -aa -instcombine -simplifycfg -pgo-icall-prom -basiccg -globals-aa -prune-eh
     -inline -functionattrs -domtree -sroa -early-cse -lazy-value-info -jump-threading
     -correlated-propagation -simplifycfg -domtree -basicaa -aa -instcombine -tailcallelim -simplifycfg
     -reassociate -domtree -loops -loop-simplify -lcssa-verification -lcssa -basicaa -aa
     -scalar-evolution -licm -loop-unswitch -simplifycfg -domtree -basicaa -aa -instcombine -loops
     -loop-simplify -lcssa-verification -lcssa -scalar-evolution -indvars -loop-idiom -loop-deletion
     -loop-unroll -mldst-motion -aa -memdep -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -gvn
     -basicaa -aa -memdep -memcpyopt -sccp -domtree -demanded-bits -bdce -basicaa -aa -instcombine
     -lazy-value-info -jump-threading -correlated-propagation -domtree -basicaa -aa -memdep -dse -loops
     -loop-simplify -lcssa-verification -lcssa -aa -scalar-evolution -licm -postdomtree -adce
     -simplifycfg -domtree -basicaa -aa -instcombine -barrier -elim-avail-extern -basiccg
     -rpo-functionattrs -globals-aa -float2int -domtree -loops -loop-simplify -lcssa-verification -lcssa
     -basicaa -aa -scalar-evolution -loop-accesses -lazy-branch-prob -lazy-block-freq
     -opt-remark-emitter -loop-distribute -loop-simplify -lcssa-verification -lcssa -branch-prob
     -block-freq -scalar-evolution -basicaa -aa -loop-accesses -demanded-bits -lazy-branch-prob
     -lazy-block-freq -opt-remark-emitter -loop-vectorize -loop-simplify -scalar-evolution -aa
     -loop-accesses -loop-load-elim -basicaa -aa -instcombine -scalar-evolution -demanded-bits
     -slp-vectorizer -simplifycfg -domtree -basicaa -aa -instcombine -loops -loop-simplify
     -lcssa-verification -lcssa -scalar-evolution -loop-unroll -instcombine -loop-simplify
     -lcssa-verification -lcssa -scalar-evolution -licm -alignment-from-assumptions
     -strip-dead-prototypes -globaldce -constmerge -domtree -loops -branch-prob -block-freq
     -loop-simplify -lcssa-verification -lcssa -basicaa -aa -scalar-evolution -branch-prob -block-freq
     -loop-sink -instsimplify
```

Figure 1.8: Ordered list of pass given for `-Oz`

Figure 1.8 shows in color that some passes are invoked several times. Most of them are analyses: as we said previously, metadata need to be updated in a way to be reused after a certain modification. But the most surprising may be the several invocations of transformation passes. Indeed, we may think that an optimization done on a program does not need to be performed again, that's not true because of one thing: the form is constantly changing and a transformation can reveal a new potential optimization. For example, unrolling loops (which consists in duplicating the body before performing the loop) could make appear redundant instructions which can be combined afterward. A lot of passes are used to prepare optimizations or clean the IR like `simplifycfg` or `loop-simplify`.

1.4 Non Size Increasing (NSI) Analysis: a good start

1.4.1 Introduction

In this section, we present an experiment in bringing analysis from Implicit Computational Complexity into real life compilers.

The analysis we described here, based on NSI programs, it simple enough to be expressed on a small assembly-like language. Since it only focus on memory allocation and deallocation, we can concentrate on these operations (that is, the `malloc` and `free` in a `C` program) and on the control flow, and forget all the complicated constructions that may be used by the programming language. Since this is a purely syntactical analysis (as all ICC analyses), it is perfectly suited to happen at compile time.

Moreover, compilers already contain many analyses and optimisation tools that we can reuse. Most of these tools are spread in modular *passes* that can be applied in various order. Typically, there is no need to rebuild the control flow of the program. It is something that is already used by many compiler optimisations and thus that already exists as a standalone pass. We need to call this pass and use its result. This limits the amount of code we have to write in order to perform our analysis.

As we said in the motivations, real compilers aim to produce efficient code as fast as possible. On the other side, ICC mainly provides analysis ensuring certain properties. But analyses can be used to fuel further optimisations!

Currently, common systems allocate a finite space of memory, if it's not enough during the execution, the program needs to ask and wait for more space and waste time. If it's too many, it's a waste of space. Here, knowing the precise amount of memory that a function or program will need can help optimising system calls: rather than using the standard library to find free memory and allocate it, it becomes possible to let the program reuse its own memory efficiently.

Thus, we chose to focus on a simple analysis that is easy to express in the compiler's intermediate representation rather than on a powerful analysis/optimisation which require more work to be used.

1.4.2 Non Size Increasing programs

Bellantoni and Cook identified the repetition of iteration (or recurrence) as a source of potential exponential growth. That is, given a program like in Figure 1.9.

```
for i=1 to m do
  for j=1 to n do
    N:=N+1
  done
  n:=N
done
```

Figure 1.9: Iterations and exponential growth

The inner loop is relatively innocuous but the outer loop takes the result of the inner loop and use it to control again the same iteration. This results in an exponential growth. In order to prevent this, Bellantoni and Cook designed some syntactic restriction on the allowed programs. This can be seen as giving "energy" to some variables of a program. Each loop (or recurrence) must be controlled by a variable that has energy (that is "normal") and the loop uses this energy, thus the result has no energy (is "safe"). This *de facto* prevents a program like the one above because `N`, as the result of a loop, has no energy and thus cannot be used to control another loop. However, not all iterations of iterations are bad. Typically, selection sort works by iterating the selection of the maximum element, itself an iteration. Hofmann express this by saying that iterating is not harmful if the result is smaller than the input, or only marginally larger. This open way for studying functions whose output has the same size has the input, that is functions which do not increase the size of data, hence the name of Non Size Increasing functions.

If we concentrate on the NSI part, without trying to handle polynomial bounds, this analysis is simple enough and can be expressed in a small assembly like language. Especially, the analysis is done by only checking memory allocation and de-allocation along the control flow graph. Thus, it is particularly well suited to be performed in a compiler.

From Safe Recursion to Non Size Increasing

The term rewriting system below computes a list of exponential length (in the length of the input). `[]` is the empty list and `::` is the (infixed) cons operation.

```
double(a, []) -> a::[]
double(a, x::xs) -> a::x::(double(a, xs))

exp([]) -> []
exp(x::xs) -> double(x, exp(xs))
```

Figure 1.10: Exponential in TRS

The point is that in the last line, the result of a recursive call to **exp** is used to control another recursive call (to **double**). This allows to "pile up" calls to **double** and, in the end, computes an exponential. To prevent this, Bellantoni and Cook forbade the writing of functions like **exp** via a syntactical restriction on programs which can be assimilated to energy used to actually perform the recurrence.

Repeated iterations is a powerful expressive construction to build many reasonable programs. Indeed, writing a program by respecting the normal/safe tiering of arguments is often difficult. Typically, Hofmann looked at an insertion sort like the program in Figure 1.11.

```
insert(y, []) -> y::[]
insert(y, x::xs) ->
   if x<y
      then x::(insert(y, xs))
      else y::x::xs

sort([]) -> []
sort(x::xs) -> insert(x, sort(xs))
```

Figure 1.11: Insertion sort in TRS

Obviously, such a program should be accepted and it makes little sense to reject insertion sort. However, Safe Recursion does reject the insertion sort as the last line tries to control the recurrence in **insert** with the result of a recurrence in **sort**.

Moreover, a close look at the programs shows that **exp** and **sort** are actually exactly the same function, which is why Safe Recursion rejects both. Thus, Hofmann came to the conclusion that the culprit is *not* **exp** but **double**. Indeed, there is a main difference between **double** and **insert** which justifies rejecting one and accepting the other. **double** produces an output that is twice as large as the input. On the other hand, **insert** produces an output which is essentially as large as the input. In other words, **insert** is Non Size Increasing and can thus be iterated without harm. A "pile" of **insert** will not create a growth exponential in the number of

insert performed. Whereas **double** does increase the size of its input (by a non-constant factor) and thus should not be iterated.

NSI and rewriting system

In order to detect NSI programs, Hofmann introduced a new datatype, the diamond ($\diamond$), with the particularity that this datatype has no constructor. That is, there is no closed term of type $\diamond$ and only variables can have this type. Moreover, variables of type $\diamond$ must be used linearly (one by one, without duplication) in the right-hand side of rules. Thus, in order to use $\diamond$ in the right-hand size of a rule, one has to provide the same number on the left-hand side. In a sense, diamonds in the reducee are "price" to be paid to apply the rule and can only be paid by diamonds already existing in the redex. And the linearity condition avoids duplicating an existing diamond to pay several "prices" with the same one. The next step is to make the type system aware of $\diamond$. When working with lists, it is the *cons* that make the size of lists, Hofmann requires a diamond for each *cons*. Thus, instead of having the classical type $\alpha, \alpha\ list \to \alpha\ list$, *cons* is now of type $\diamond, \alpha, \alpha\ list \to \alpha\ list$.

To prove a bound on space usage, we only need to know the maximum amount of diamonds required at any time. That is, we can focus on the *quantitative* part of the analysis and forget about the *qualitative* part of how and where these diamonds are reused. With this System, it is still possible to write the insertion sort function Figure 1.12 (where d and d' are $\diamond$).

```
insert (d, y, []) -> cons(d, y, [])
insert (d, y, cons(d', x, xs)) ->
        if x<y
        then cons(d', x, (insert(d, y, xs)))
        else cons(d, y, cons(d', x, xs))

sort ([]) -> []
sort (cons(d, x, xs)) -> insert(d, x, sort(xs))
```

Figure 1.12: Insertion with diamonds

The diamonds that are liberated by destructing *cons* on the left-hand side can be reused to build new *cons* on the right-hand side. It is worth noticing that **insert** itself requires an extra $\diamond$ which can be provided by the initial call made in **sort**. On the other hand, it is now impossible to write **double** like in Figure 1.13.

Indeed, the last line uses its two diamonds for the external *cons* and cannot any more pass one to the recursive

```
double(d, a, []) -> cons(d, a, [])
double(d, a, cons(d', x, xs)) ->
       cons(d, a, cons(d', x, (double(???, a, xs))))
```

Figure 1.13: Double with diamond - impossible

call. And if one tries to remove the $\diamond$ argument from **double**, then it is impossible to perform the two *cons* in the right-hand side of the rule with only one diamond provided by the left-hand side (because of the linearity condition).

Thus, this apparently simple change in the type system allows to detect NSI programs. Moreover, it should be noted that Safe Recursion prevented to write both **exp** and **sort** while NSI directly prevents to write **double** and thus identifies the correct culprit.

With this new type system, it is still possible to write the insertion sort in the usual way, but exponentiation is not possible anymore.

1.4.3 NSI detection in a compiler

NSI and imperative programs

When it comes to languages with explicit memory management, the diamonds have a very natural interpretation. Indeed, Hofmann shown that NSI programs [Hofmann, 1999b] can be compiled into `C` programs without `malloc`. The diamonds are immediately translated into *pointers to free memory*. If the program is NSI, then it reuses diamonds but does not need new ones, that is, it reuses memory but does no need new pointers (obtained by a `malloc`).

A typical implementation of linked lists in `C` as in Figure 1.14a is to represent a cell as a value and a pointer to the next cell, and a list as a pointer to the first cell of the list. With this implementation, *cons* needs to create a new cell to store the new value, that is needs to perform a `malloc` to obtain free memory. Moreover, once a cell is no longer in use, it is up to the programmer to `free` it, even if some other cell is immediately recreated: there is no reuse of memory.

Now, if one thinks of the analogy "diamond = pointer" Figure 1.14, it is possible to rewrite the *cons* without any `malloc`, provided that an extra argument is used: a pointer to free memory, that is, the diamond in the functional version above. Note that obviously nothing ensure that the pointer indeed points toward some

```
struct cell {
  int value;
  struct cell * next;
};

typedef struct cell* list;

liste nil() {
  return (list) NULL;
}

int is_empty(list l) {
  return (l==NULL);
}

list cons(int a, list l) {
  struct cell *c;
  c=malloc(sizeof(struct cell));

  c->value = a;
  c->next = (struct cell *) l;

  return (list) c;
}
```

(a) Linked list in C

```
typedef struct cell* diamond;

list cons(diamond d, int a, list l) {
  d->value = a;
  d->next = (struct cell *) l;

  return (list) d;
}
```

(b) Diamond = Pointer

Figure 1.14: NSI into C programs

reusable memory and any data in it is destroyed by the operation. It is up to the programmer to ensure that the diamonds are indeed available to be used. Note that a `diamond` is *essentially* a pointer to (enough memory to store) a cell while a list is only represented as such a pointer by commodity.

With this version, we can write an insertion sort that does not use any `malloc` Figure 1.15.

Note that this recursive version is made to look like the functional insertion sort above. It is likely to be inefficiently compiled while an imperative version with loops would be more efficient due to compiler optimizations, and can still be written without `malloc`.

It is possible to have an overview of the diamonds (*i.e.* of the `malloc` and `free` in an imperative version of the program) behaviour during the recurrence. The recursion gets a diamond when pattern matching is performed to read and compare; if it is the right position for the value it uses two diamonds (calls `cons` two times): one to add the new element and another to replace the previous element; otherwise, it simply replaces the old element (with its own diamond, `dd`). This way, we understand that the **insert** function will globally constructs one element, and thus require and extra diamond, `d`, which can be provided by **sort**.

This NSI analysis is not only a complexity analysis. Apart from the complexity information it gives some security properties (the program won't overflow memory and won't cause memory leaks) and some possibilities

```
list insert (diamond d, int y, list l) {
  diamond dd; int x; list xs;
  if (is_empty(l))
    return cons(d,y,l);
  else {
      x=l->value;
      xs=l->next;
      dd=l; // destroyed, we can reuse it
      if (x<y)
        return cons(dd,x,insert(d,y,xs));
      else
        return cons(d,y,cons(dd,x,xs));
    }
}
```

```
list sort( list l) {
  diamond d; int x; list xs;
  if (is_empty(l))
    return nil();
  else {
      x=l->value;
      xs=l->next;
      d=l; // the first cell is
           destroyed
      return insert (d,x, sort(xs));
}
```

Figure 1.15: Insertion without `malloc`

for optimization. It becomes indeed possible to completely remove the `malloc` from the code and let the program efficiently reuse its memory. This will avoid several system calls and calls to the standard library that can slow down the program execution.

Forbidding heap allocation avoid heap overflow, fix the memory complexity and avoid system calls. By opposition, stack-based memory allocation is simpler, faster and safer. Because data is added and removed in a Last-In-First-Out way it is easy to track static allocation by counting every local *alloca* and waiting final block of local function (or module) to consider the amount allocated blocks freed. Furthermore the stack design is already in a reuse principle similarly to the NSI principle.

But we can't force programmers to only use static allocation and to consider memory complexity in all cases. It is not realistic for humans to learn and have an overview on each structure and each dependency. Therefore we are looking for an automatic analysis. In this case the programmer will not need to know the existence of this analysis as many others in compilers. In most of cases, the programmer has no idea of the becoming of his algorithm during the compilation.

For the NSI analysis we are searching for the maximum amount of diamonds consumed at a time. By analogy with a real language like C, we count the number of `malloc` and `free` called at each step of the algorithm. Here, we will consider a stack machine which allocates and deallocates an indivisible part of memory: we simulate a stack behavior on heap.

In Figure 1.16, we will consider a stack machine which allocates and deallocates an indivisible part of

memory: we simulate a stack behavior on heap. If we define a `pop` function that destructs a cell (here a `struct` called "node") and a `push` one that constructs one cell, we obtain a *C* code as below.

```
struct node* push(struct node* head,int data){
    struct node* tmp = (struct
        node*)malloc(sizeof(struct node));
    if (tmp == NULL)
        exit (0);
    tmp->data = data;
    tmp->next = head;
    head = tmp;
    return head;
}

struct node* pop(struct node *head,int *element){
    struct node* tmp = head;
    *element = head->data;
    head = head->next;
    free(tmp);
    return head;
}
```

```
struct node * insert_rec (struct node *head, int
    data){
    if (empty(head))
        return push(head,data);
    else {
        int element;
        head = pop(head,&element);
        if (element < data)
            return push
                (insert_rec(head,data),element);
        else
            return push(push(head,element),data);
    }
}
```

Figure 1.16: Recursive insertion sort

Thanks to the stack machine simulation, it is easier to have an overview of the `malloc` and `free` behaviors in the recurrence. The loop consumes a cell in order to read and compare, if it is the right position in the list it constructs a node two times: one time to add the new cell and, in all cases, another time to replace the previous cell. By this way, we understand that the insert function will globally construct one cell.

Control Flow Graph

It is easy to do this analysis using a Control Flow Graph (CFG). A CFG is a graph representation of all paths that might be traversed through a program during its execution. Each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps. Directed edges are used to represent jumps. A *CFG* starts with one *entry-block* and has one or several *exit-blocks* (or leaves). That builds the structured programming concept. In the following we give a simple example: a reverse list function.

In the following subsection, we simplify the CFG generated in Figure 1.18a with a pseudo representation as in Figure 1.18b.

(∗ with accumulator ∗)

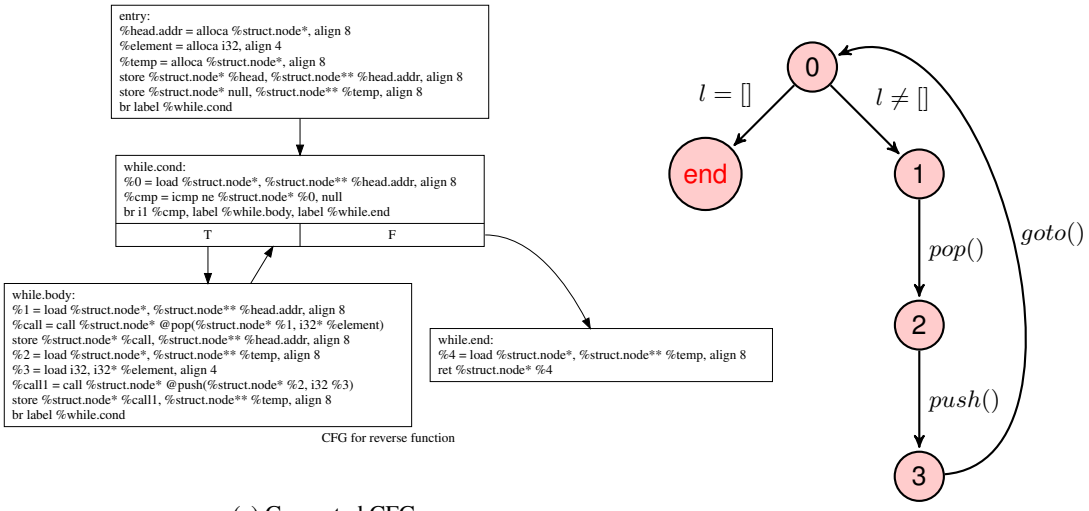
rev(l) −> **rev1**(l, nil)
rev1(nil, acc) −> acc
rev1(cons(t), acc) −> **rev1**(t,cons(acc))

(a) Reverse function in TRS

0: **if** l = [] **then** goto 4;
1: h:=**pop**(l); (∗ frees a diamond∗)
2: **push**(h,acc); (∗consumes a diamond∗)
3: goto 0;
4: **end**;

(b) Reverse function in Stack Machine

Figure 1.17: Simple example: Reverse list function



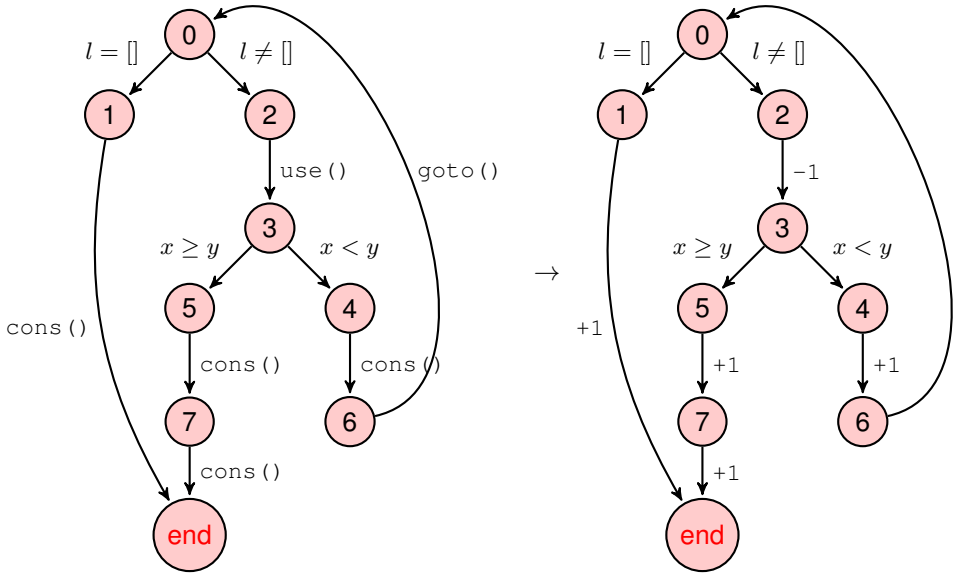
(a) Generated CFG

(b) Simplified CFG

Figure 1.18: Reverse list CFG of program in

A Resource Control Graph analysis

Now if we come back to our main example, Figure 1.19a is the simplified CFG for the insertion sort function seen above. For our analysis, we need to augment this CFG by adding a weight (the diamond usage) to each instruction. This way it becomes the Resource Control Graph [Moyen, 2009] (RCG) Figure 1.19b. Note that if we drop the qualitative part of the analysis, we don't need the $\diamond$ type anymore as counting variables of this type is equivalent to counting instructions producing or consuming it. This, however, only provides a proved bound on space usage but cannot actually remove `malloc` and `free` from the program. Using this *RCG* given in Figure 1.19b we can find the most expensive path according to this weight. A maximum weighted path is quickly computable with a classical algorithm such as Dijkstra's or Bellman-Ford's. It is equivalent to find the shortest paths in a weighted graph. We also need to detect positive loop in a polynomial time. Here we are in the case where we have a single entry source. The Dijkstra's algorithm can provides the shortest path in $O(|V|^2)$



(a) Control Flow Graph of the insertion sort

(b) Resources Control Graph of the insertion sort

Figure 1.19: Insertion sort RCG

time (with $|V|$ the number of vertices) but unlike the Bellman-Ford algorithm it can't be used on graph with negative edge weights and it does not detect negative loop. Bellman-Ford's algorithm runs in $O(|V|.|E|)$ time (with $|E|$ the number of edges).

Bellman-Ford's algorithm in a pass

Initialisation To initialize all vertices, we attribute an infinite negative number represented with a null pointer expect for the entry node which is initialized to 0 (Figure 1.20a).

Relaxation (Figure 1.20b) If we find a higher weight by a certain edge we actualise the vertex's distance.

Positive loop detection Positives loops are detected by searching vertex which has a lower weight than a precedent vertex plus the corresponding edge cost (Figure 1.21).

We understand that, because the analysis is only static, it is not accurate. We only consider the worst case to ensure the NSI property, this is why we want to have false negatives instead of false positives. Avoiding both is undecidable...

```
// init
for(std::vector<Vertex>::iterator
    it=vertices.begin();
        it!=vertices.end(); ++it){
    if(*it==entry)
        distance[*it]=0;
    else
        distance[*it]=nullptr; // suposed
            infinit negative number
}
```

(a) Initialisation.

```
// relax
for(unsigned long i=1; i<vertices.size(); i++){
    for(Edges::iterator ed=edges.begin(); ed!=edges.end();
        ed++){
        Vertex u = ed->first.first;
        Vertex v = ed->first.second;
        if(distance[u] + ed->second > distance[v])
            distance[v] = distance[u] + ed->second;
    }
}
```

(b) Relaxation.

Figure 1.20: First pass sample.

```
// check the positives cycles
for(Edges::iterator ed=edges.begin(); ed!=edges.end(); ed++){
    Vertex u = ed->first.first;
    Vertex v = ed->first.second;
    if(distance[u] + ed->second > distance[v]){
        return false;
    }
}
return true;
```

Figure 1.21: Positive loop detection.

Implementation details

In the LLVM-IR, the `malloc` instruction allocates one or more elements of a specific type on the *heap*, returning a typed pointer to the new memory. The `free` instruction releases memory allocated through `malloc`. When the native code is generated, those instructions are converted to the appropriate native function calls, allowing also customizations. There are no implicit accesses to memory, this simplifies all memory access analyses. LLVM also builds the *CFG* of every functions. As we said previously, this data structure will help us to do static analysis. LLVM provides some tools to visit and match instructions targeted in the entire graph given. This representation can give foundations in order to create a new one. For instance, the *LoopInfo* pass calculates the nesting structure of loops in a function.

In order to build our RCG we use a LLVM tool: Basic Blocks visitor (Figure 1.22) which goes through each basic block on the CFG. We can add a function to run for each basic block. Here we compute their weight and map this to be used by another pass. We can also do it recursively by recalling the evaluate function for each *CFG* branch.

```
// InstVisitor a LLVM tool
// Here it's a definition of an herited class
// Counts easily occurence of specific call
struct CountCallVisitor : public InstVisitor <CountCallVisitor>{
    int weight;
    CountCallVisitor () : weight(0) {}
    void visitCallInst ( CallInst &CI){
        if (const Function *F = CI. getCalledFunction () ){
            if (F->getName() == "malloc")
                weight++;
            else if (F->getName() == "free")
                weight--;
        }
    }
};
```

Figure 1.22: A visitor.

In both case the complexity is $O(|V|)$ where $|V|$ is the number of vertices or Basic Blocks.

Maximal weight computation Now we should be able to calculate the maximal weight or worst case space that might be used by the program. As we said previously, by reconsidering our graph we can use the Bellman-Ford's algorithm to find the longest path in our weighted graph.

Type definitions can show how we designed our RCG using BasicBlock Figure 1.23a. If we build our CFG using the following code, list of BBs in functions will be not in a good order according to the control flow. Basically, BBs are stocked in a list in the Function Class and not as a graph. We need to, recursively, travel through each successor of blocks, starting with the entry one. To fill up this new graph we will prefer to use a Depth-First Search to obtain our nodes in the correct order.

1.4.4 Conclusions and further work

We built a static analyzer in almost 250 lines of code mostly because it reuses the LLVM's environment and tools. It can be split in two parts: the first builds a Resources Control Graph and the second computes functions weights and detects positives loops. This analysis has been tested on classical lists manipulation such as `reverse`, `concat`, `insertion sort` and `quick sort`. This tool can answer to the question "Is this program NSI?" in some cases. It assumes that every loops' body will be executed an undecidable number of time then it does not provide accurate bounds.

```
typedef llvm::BasicBlock* Vertex;
//we need to be able to reach pred and
    next BB with the edge structure
typedef std::pair<Vertex, Vertex>Edge;
//edges with their weight (corresponding
    to the weight of the first BB)
typedef std::map<Edge, int> Edges;
//need to begin with the first entry
std::vector<Vertex> vertices;
std::map<Vertex, int> distance;
```

(a) The type definitions.

```
virtual void gimmeVertices(BasicBlock* first ){
    //mark checked BB
    BBmap2[first]=true;
    //get terminator instructions informations
    const TerminatorInst *TInst = ( first )->getTerminator();
    //local analysis (research "malloc" and "free" calls)
    vertices.push_back(first);
    //for each branch
    for (unsigned I = 0, NSucc =
        TInst->getNumSuccessors();
            I < NSucc; ++I) {
        BasicBlock *newSucc = TInst->getSuccessor(I);
        //prevent from recursive loops
        if (!BBmap2[newSucc]){
            //recursive call
            gimmeVertices(newSucc);
        }
    }
}
```

(b) Recursive Depth First Search visitor.

Figure 1.23: Implementation details

Furthermore, if this analysis is done on the entire program, it can be seen as a tool to detect memory leak. This work is our first step of further implementations of ICC theories into widely used compilers.

Figure 1.24 shows the result given by our analysis for the sort function using the insertion already analyzed by the pass (we can see "`call @insert:1`" which means that the maximum weight of the insert function is equal to 1). At the end we can read that the maximum weight of the `sort_by_insert` function is 0.

A lot of work remains to be done. First of all, dependence problems appear for non-analyzed functions called in the current CFG. External libraries should be analyzed first and results need to be kept somewhere to avoid recompilation, maybe by using an annotated system like the Clang Language Extensions[4] or something similar for the *Intermediate Representation*. It could be a great idea to provide an external library like *libc* entirely certified with some Implicit Complexity properties. Those properties would be attached with the compiled library. Then, because it is only added, this could work on any pre-existent code. By this way we could globalize the "proof-carrying code" [Necula, 1997] movement.

Optimizations can be considered by introducing a type diamond in a way to have more information about reusability of each memory and by customizing the standard dynamic allocations and deallocations. Elimination

[4] http://clang.llvm.org/docs/LanguageExtensions.html

```
struct node * sort_by_insert (struct node *head){
        int element;
        struct node * temp = NULL;
        while(head!=NULL){
                head = pop(head,&element);
                temp = insert (temp,element);
                display (temp);
        }
        return temp;
}
```

(a) Sort by insert in C

```
Function sort_by_insert
entry:
-2 successors :
|while.end:
|-0 successors :
|while.body.preheader:
|-1 successors :
| |while.body:
| | free here... -1
| | call @insert : 1
| |-2 successors :
| | |if.else.i:
| | | call @puts : 0
| | |-1 successors :
| | | |while.cond.backedge:
| | | |-2 successors :
| | | | |while.end.loopexit:
| | | | |-1 successors :
| | | | | |Loop !
| | | | |Loop !
| | |if.then.i:
| | | call @printf : 0
| | |-1 successors :
| | | |do.body.i:
| | | | call @printf : 0
| | | |-2 successors :
| | | | |do.end.i:
| | | | | call @putchar : 0
| | | | |-1 successors :
| | | | | |Loop !
| | | | |Loop !
Function max weight : 0
```

(b) Output given by the analysis

Figure 1.24: Results

of `malloc` calls is not a new idea [Hofmann and Jost, 2003] but, as far as we know, it has never been done in a real compiler. Here we can replace `malloc` and `free` calls by our own instructions to simulate them without any system call. We did not study more this option yet.

We can also, by studying more accurately relations between input and bounds [Albert et al., 2008a], approximate a *Space Complexity* [Avanzini et al., 2013] and, maybe, the *termination* [Lee et al., 2001] because this last work is also based on weighted Control Flow Graphs or Resources Control Graphs [Moyen, 2009].

Chapter 2

Data Flow Approach for Complexity Analysis

In this chapter I present an idea introduced to Jean-Yves Moyen and me by Lars Kristiansen. I then worked in collaboration with them and Thomas Seiller in Copenhagen to reformulate and enhance it (at the time of writing, a common paper is in process). This framework is then reused for potentials optimizations in .

First an historical part of the idea is given, second I try to introduce a formal definition of the framework with some abstract examples and finally I propose different instances of practical uses of this idea.

2.1 History of this idea

This section is a retranscription of what have been said by Lars Kristiansen at DICE2017, it introduces the idea of this data-flow approach for complexity analysis and also explains why we decided to work on a specific instance of this Data-Flow Analysis framework for finding loop quasi-invariants chunks in chapter 3.

First, when Lars studied Bellantoni and Cook recursion in 2000, he observed on a simple loop-language like shown in Figure 2.1 that if the data flow graph of a loop does not contain any *circle* then the computed values will be bounded by polynomials in the input, thus the program runs in polynomial time. Those *circles* can be detected by an analysis. [Kristiansen and Niggl, 2004] described a method to do so using a stacks.

(Variables)	X	::=	$X_1 \mid X_2 \mid X_3 \mid \ldots \mid X_n$
(Expression)	*exp*	::=	$X \mid X_1 + C \mid X_1 - C \mid X_1 + X_2 \mid X_1 - X_2$
(Command)	*com*	::=	$X = exp \mid com;com \mid$ `LOOP` X `DO` *com* `END`

Figure 2.1: Simple LOOP-language.

Lars knew that he could extend his language with conditions and refined the data-flow analysis by introducing different kinds of flow (m for max, w for weak-polynomial and p for polynomial) that are more or less "*harmful* things".

While m-flows are harmless, w-flow and p-flow might be harmful. If it exists a *circle* of such flow, the program may computes a value not bounded by a polynomial in the input. The difference between w-flow and p-flow is that w-flow is iteration-independent, the value computed does not depend on the number of times the loop's body is iterated, it is invariant. To avoid harmful things, we need to put restrictions on w-flow and p-flow.

[Kristiansen and Jones, 2009] developed a syntactic proof calculus for assigning mwp-flow graphs to programs represented as matrices over a semi-ring $(\{0, m, w, p\}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$. Lars wondered how powerful is this mwp-method... We may argue that it is by experimenting the method on large quanta of programs or with relevant well known examples or proving the characterization of some complexity classes. He talked about another way to test this method: by finding good approximation of approximation etc... of an undecidable problem and the complexity of those approximations. Here the non-trivial question was "how to find good approximations?". Approximations often come with restrictions, the goal is to avoid restrictions that exclude programs that programmers actually would like to write.

Lars worked on a simple loop-language with deterministic conditions and standard assignments but no constant were allowed, this implies that no variables can be reset. Amir Ben-Amram proved in [Ben-Amram, 2010] that the problem is still decidable if we allow the constant 0 but becomes *PSPACE*-complete. If we allow the constant 1 the problem is still open [Ben-Amram and Kristiansen, 2012]. In the previous paper, they consider two kinds of loops (one definite and one indefinite), three kinds of assignments (standard, max and weak) and four forms of expressions (`X+1`, `X+Y`, `X`, `0`) therefore give different result of characterization regarding to the loop configuration.

A real-life application came in his mind when Neil Jones asked:

> Concretely, here is a puzzle I have been playing with recently : how to do CODE MOTION automatically, in a way that can yield superlinear program speedup. Traditionally, code motion involves moving a single command to the top of a loop. But clearly this cannot result in an asymptotic program speedup. On the other hand, if one could move a group of commands, one could gain much more,, eg move an inner loop outside to gain quadratic speedup. The question: could your (and Amir's) analyses be used to this end: identifying groups of commands that can be moved outside a loop without changing semantics?

The answer is yes, as we will show and explain in section 3.2.

2.2 A Data Flow Analysis Framework

In this section we give a formal definition of the framework starting from the methods presented by Lars in [Kristiansen, 2012] and [Kristiansen and Jones, 2009]. It is voluntarily reformulated and generalized to give the reader enough intuitions for the following potential instances of analysis, particularly the one presented in chapter 3.

To introduce our generic Data Flow Analysis framework, we will work with a simple imperative `WHILE`-language with semantics similar to `C`. The grammar is given by:

(Variables)	X	::=	$X_1 \mid X_2 \mid X_3 \mid \ldots \mid X_n$
(Expression)	*exp*	::=	$X \mid$ `op(`*exp*`,...,`*exp*`)`
(Command)	*com*	::=	$X =$ *exp* $\mid$ *com*`;`*com* $\mid$ `skip` $\mid$ `while` *exp* `do` *com* `od` $\mid$
			`if` *exp* `then` *com* `else` *com* `fi` $\mid$ `use(`X_1`,...,`X_n`)`

Figure 2.2: Simple WHILE-language.

A `WHILE` program is thus a sequence of statements, each statement being either an *assignment*, a *conditional*, a *while* loop, a *function call* or a *skip*. The *use* command represents any command which does not modify its variables but use them and should not be moved around carelessly (typically, a `printf`). In practice, we currently treat all function calls as *use*, even if the function is pure. *Statements* are abstracted into *commands*. A *command* can be a statement or a sequence of commands. We also call a sequence of commands a *chunk*.

2.2.1 Data Flow Graph

A generic definition of a Data Flow Graph for a given command `C` will be a weighted relation on the set V of variables involved in `C`. Formally, this can be represented as a matrix over a semi-ring, with the implicit choice of a denumeration of the set V. We can imagine a generic semi-ring $(\mathcal{S}, +, \times)$.

Definition 1 *A* Data Flow Graph *(*DFG*) for a command* `C` *is a* $n \times n$ *matrix over the semi-ring* $(\mathcal{S}, +, \times)$ *where* n *is the number of variables involved in* `C`*.*

We write $\mathrm{M}(\mathtt{C})$ *the* DFG *of* `C`*.*

For technical reasons, we identify the DFG of a command `C` with any embedding of $\mathrm{M}(\mathtt{C})$ in a larger matrix.

I.e. we will abusively call the DFG of C any matrix of the form

$$\begin{pmatrix} \mathrm{M}(\mathtt{C}) & 0 \\ 0 & \mathrm{Id} \end{pmatrix},$$

implicitly viewing the additional rows/columns as variables that do not appear in C.

In all examples, we will be using weighted relations, or weighted bi-partite graphs, to illustrate these matrices. We leave it to the reader to convince herself that these matrices and graphs are in one-to-one correspondence. Moreover, all examples will be based on the semi-ring $(\{0, 1, \infty\}, \max, \times)$, since it is the specific case that will be under study in later chapter: the operators describe the tropical algebra in Figure 2.3. It will be used to represent dependencies. This will be recalled in subsection 3.3.1.

max	0	1	∞
0	0	1	∞
1	1	1	∞
∞	∞	∞	∞

$\times$	0	1	∞
0	0	0	0
1	0	1	∞
∞	0	∞	∞

Figure 2.3: Addition and Multiplication in the semi-ring $(\{0, 1, \infty\}, \max, \times)$.

In examples, we represent relations by matrices with two types of arrows between two copies of the set of variables. The left-hand side represents variables before execution and the right-hand side represents the variables after execution (i.e time flows from left to right). Figure 2.4 illustrates these types of relations: plain arrows for ∞, dashed arrows for 1. Relation of type 0 then corresponds to the absence of arrows ending on the right occurrence of z.

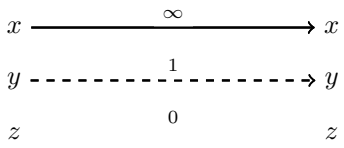


Figure 2.4: Example of different types of relations

Of course, this is a simple example of semi-ring imagined $(\{0, 1, \infty\}, \max, \times)$, but we can guess that it could have bigger set of elements - or kind of relations. For convenience we define, given a command C, the following two sets of variables.

Definition 2 *Let* $\mathtt{C}$ *be a command. We define* $\mathrm{In}(\mathtt{C})$ *(resp.* $\mathrm{Out}(\mathtt{C})$*) as the set of variables* influencing *(resp.* influenced*) by* $\mathtt{C}$*.*

While in later sections we will only be studying a specific case of those, the theory is quite general and pinpoints to formal links with various works on static analysis [Lee et al., 2001, Abel and Altenkirch, 2002, Kristiansen and Jones, 2009] and semantics [Laird et al., 2013, Seiller, 2016a, Seiller, 2016b].

2.2.2 Constructing DFGs

We now describe how the DFG of a command can be computed by induction on the structure of the command. Base cases (skip, use and assignment) should be defined depending on what the DFG will be used for. The DFG for assignments are obtained by straightforward generalisation of the cases. Next, define a variable $\mathtt{e}$ – standing for *effect* – not part of the language, and define:

- $\mathrm{M}(\mathtt{skip})$ as the "empty matrix" with 0 rows and columns[1];
- $\mathrm{M}(\mathtt{use}(\mathtt{X_1}, \dots, \mathtt{X_n}))$ as the matrix with coefficients from each X_i and $\mathtt{e}$ to $\mathtt{e}$ equal to ∞, and 0 coefficients otherwise. Here, the $\mathtt{use}$ is seen as a $\mathtt{printf}$ function, it creates dependencies on involved variables and it is non-invariant because it can have side-effects.

Composition and Multipaths.

We now turn to the definition of the DFG for a (sequential) *composition* of commands. This abstraction allows us to see a block of statements as one command with its own DFG.

Definition 3 *Let* $\mathtt{C}$ *be a sequence of commands* $[\mathtt{C_1}; \mathtt{C_2}; \dots; \mathtt{C_n}]$*. Then* $\mathrm{M}(\mathtt{C})$ *is defined as the matrix product* $\mathrm{M}(\mathtt{C_1})\mathrm{M}(\mathtt{C_2}) \dots \mathrm{M}(\mathtt{C_n})$*.*

The product of two $N \times N$ matrices A, B is defined here as usual as the matrix C with coefficients: $C_{i,j} = \sum_{k=1}^{n}(A_{i,k} \times B_{k,j})$ (a definition that makes sense from the semi-ring axioms).

It is standard that the product of matrices of weights of two graphs F, G represents a graph of length 2 paths. This operation of matrix multiplication corresponds to the computation of *multipaths* [Lee et al., 2001] in the graph representation of DFGs. We illustrate this intuitive construction on an example in Figure 2.5.

[1] Up to the identification of the DFG with its embeddings, it is therefore the identity matrix of any size.

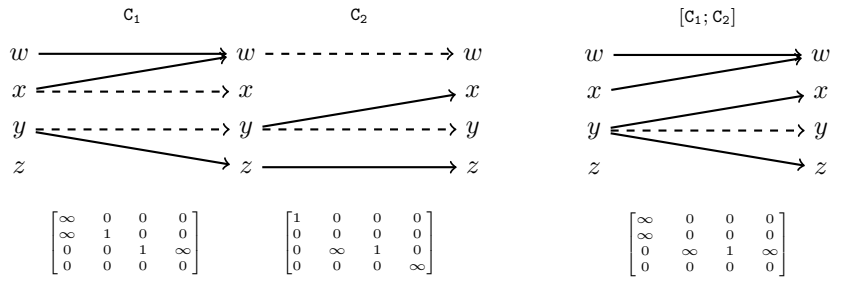


Figure 2.5: DFG of Composition.

Conditionals.

We now explain how to compute the DFG of a conditional, i.e. we define the DFG of $\mathtt{C} := \mathtt{if\ E\ then\ C_1\ else\ C_2;}$ from the DFG of the commands $\mathtt{C_1}$ and $\mathtt{C_2}$.

Firstly, we need to take into account that both commands $\mathtt{C_1}$ and $\mathtt{C_2}$ may be executed. In that case, the overall command $\mathtt{C}$ should be represented by the sum $\mathrm{M}(\mathtt{C_1}) + \mathrm{M}(\mathtt{C_2})$.

However, in most cases, it is not enough to consider $\mathrm{M}(\mathtt{C_1}) + \mathrm{M}(\mathtt{C_2})$, and the DFG of the command $\mathtt{C}$ should be obtained by adding a *conditional correction* that may depend on the expressions $\mathtt{E}$ and $\mathtt{C}$. This correction will here be written as $\mathcal{C}^{\mathtt{C}}(\mathtt{E})$.

Definition 4 *Let* $\mathtt{C} = \mathtt{if\ E\ then\ C_1\ else\ C_2;}$. *Then* $\mathrm{M}(\mathtt{C}) = \mathrm{M}(\mathtt{C_1}) + \mathrm{M}(\mathtt{C_2}) + \mathcal{C}^{\mathtt{C}}(\mathtt{E})$.

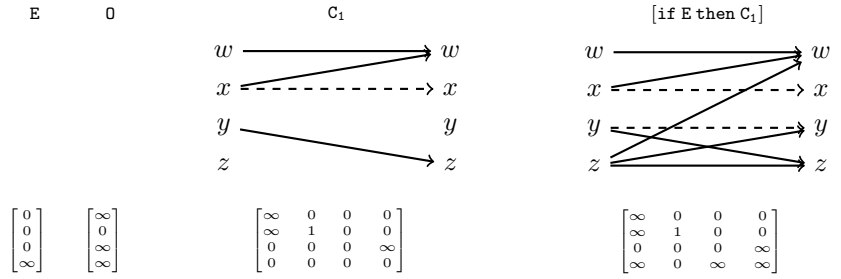


Figure 2.6: DFG of Conditional.

While Loops

Lastly, we define the DFG of a command $\mathtt{C} := \mathtt{while\ E\ do\ C_1;}$ from $\mathrm{M}(\mathtt{C_1})$. First, we define a matrix $\mathrm{M}(\mathtt{C_1^*})$ representing iterations of the command $\mathtt{C_1}$. Then, as for conditionals, we introduce a *loop correction* $\mathcal{L}^{\mathtt{C}}(\mathtt{E})$.

When considering iterations of $\mathtt{C}_1$, the first occurrence of $\mathtt{C}_1$ will influence the second one and so on. Computing the DFG of $\mathtt{C}_1^n$, the n-th iteration of $\mathtt{C}_1$, is just computing the power of the corresponding matrix, i.e. $\mathrm{M}(\mathtt{C}_1^n) = \mathrm{M}(\mathtt{C}_1)^n$. But since the number of iteration cannot be decided *a priori*, we need to sum over all possible values of n. The following expression then defines the DFG of the (informal) command $\mathtt{C}_1^*$ corresponding to "iterating $\mathtt{C}_1$ a finite (but arbitrary) number of times":

$$\mathrm{M}(\mathtt{C}_1^*) = \mathrm{limit}_{k\to\infty} \sum_{i=0}^{k} \mathrm{M}(\mathtt{C}_1)^i$$

To ease notations, we note $\mathrm{M}(\mathtt{C}_1)^{(k)}$ the partial summations $\sum_{i=0}^{k} \mathrm{M}(\mathtt{C}_1)^i$.

Definition 5 *Let* $\mathtt{C} = \mathtt{while\ E\ do\ C_1;}$. *Then* $\mathrm{M}(\mathtt{C}) = \mathrm{M}(\mathtt{C}_1^*) + \mathcal{L}^{\mathtt{C}}(\mathtt{E})$.

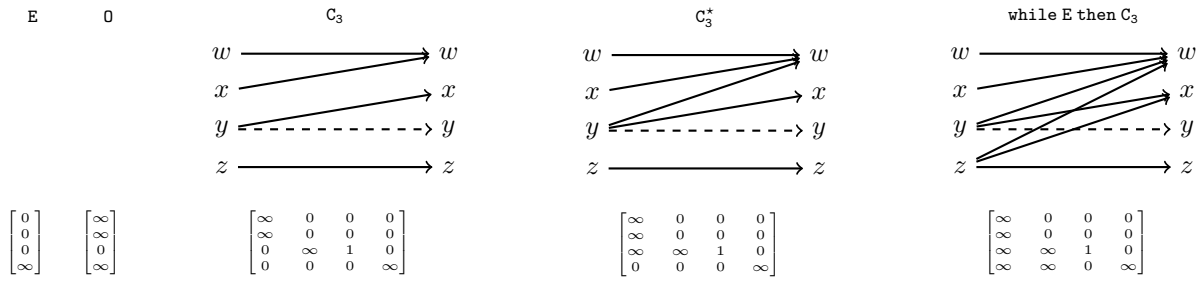


Figure 2.7: DFG of While Loop.

Note that $\mathrm{M}(\mathtt{C}^*)$ is not always defined (depending on the choice of semi-ring). In the case of the semiring $(\{0, 1, \infty\}, \max, \times)$, the set of all relations is finite and the sequence $(\mathrm{M}(\mathtt{C}_1)^{(k)})_{k\geqslant 0}$ is monotonic, hence this sequence is eventually constant. I.e., there exists a natural number N such that $\mathrm{M}(\mathtt{C}_1)^{(k)} = \mathrm{M}(\mathtt{C}_1)^{(N)}$ for all $k \geqslant N$. One can even obtain a reasonable bound on the value of N.

Lemma 1 *Let* $\mathtt{C}$ *be a command, and* $K = \min(i, o)$, *where* i *(resp.* o*) denotes the number of variables in* $\mathrm{In}(\mathtt{C})$ *(resp.* $\mathrm{Out}(\mathtt{C})$*). Then, the sequence* $(\mathrm{M}(\mathtt{C}^{(k)}))_{k\geqslant K}$ *is constant.*

2.3 Potential Instances of Analysis

To perform complexity bounds analysis. We've already talked about [Kristiansen and Jones, 2009]. They trace different kind of value growth flow to provide complexity bounds then termination analyses on a LOOP-language. It is straightforward to see how we can use our framework to do so by understanding the *mwp*-Algebra they use: it is a finite closed semi-ring $(\{0, m, w, p, \infty\}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$. As shown in section 3.5.1 and subsection 3.6.1 it is simple to implement such semi-ring and customize the behaviour of such DFA to fit with the method described by Lars Kristiansen and Neil D. Jones. [Ben-Amram et al., 2008] present a similar work for imperative programs with bounded loops. Compared to [Kristiansen and Jones, 2009] its inferences are sound and complete and its complexity is polynomial. They give sets of *dependency types* like $(\{0, 1, 1+, 2, 3\}$ respectively corresponding to *none*, *copy*, *additive*, *polynomial*, and *at least exponential*. This forms composable matrices describing data-flow from variables to variables.

To perform static allocation analysis. We can imagine that a RCG section is a specific case of DFG. But we can do better here if we consider only variables corresponding to the `malloc` and `free` calls with a specific weight (the amount of memory allocated). It would be more precise than RCG since we can keep a trace of the sum and the variables.

To do energy consumption analysis. As in [Grech et al., 2015] we can use our framework to do Energy consumption prediction at Intermediate Representation level. [Eder and Gallagher, 2017] shows that we can measure consumption by (hardware) instructions. We can use a kind of map of energy consumed regarding to the instructions (and maybe the operands). Here we can guess that it will be necessary to have bounds over loop iterations since energy can only be summed. This could be useful for safeness of course but also for consumption optimization. For instance [van Gastel and van Eekelen, 2017] propose to study energy consumption of a specific hardware regarding to an algorithm in a way to customize algorithms regarding to the hardware or the other way around. We can then imagine a compiler which chooses a specific implementation regarding to some parameters on the worst case energy consumption.

To compute "worst" dependency graphs. We will see in detail a possible use of this customizable framework in the next chapter: the main idea is to provide a "worst" dependency graph over chunks of commands in a way

to detect invariants and quasi-invariants. It could be an enhancement of the famous Loop Invariant Code Motion therefore an optimization which was developed independently of questions about complexity analysis.

More theoretically we suppose that DFGs can measure any desirable properties of complexity - within the meaning of Blum axioms - if we know how to bound the complexity of each individual instruction. Also we can imagine that external sources could provide the analyses of more or less complex pieces of code. External sources could be humans who have proven such bounds. In such cases, the tool will have to combine and automatically infer the complexity of the whole program. This means that the analysis does not need to be smart therefore expensive, such thing could be possible with interactions between the compiler and the user. Annotations à la Frama-C [Duprat et al., 2016] could ease this kind of interactions.

2.4 Conclusion

I see this framework as a tunable data flow analyzer taking as parameters a semi-ring and a pass or a method describing the way the basic matrices have to be enriched. This framework could be used for different kind of purpose. As briefly mentioned previously, it can enhance pre-existent analysis and can make data flow analysis more accessible for researchers and individuals. In the next chapter we present one potential use we developed during my thesis.

Chapter 3

Loop Quasi-Invariant Chunk Motion

3.1 Introduction

We've seen that compilers turn some high-level program into a (semantically) equivalent low-level assembly program. This translation implies many smaller transformations, notably because features such as objects, exceptions, or even loops need to be expressed in assembly language. The compiler also performs many optimisations aiming at making the code more efficient. These are often needed to streamline the code generated by the transformations but can also be used to optimise the source code.

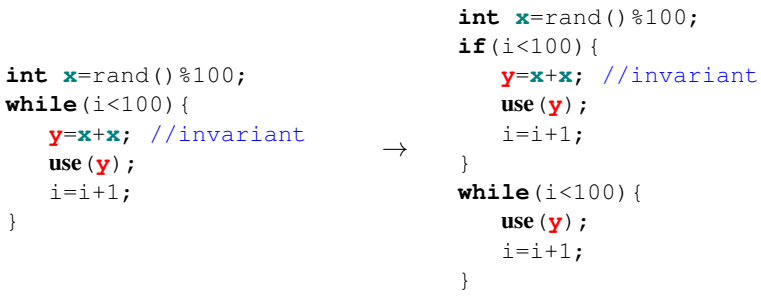


Figure 3.1: Hoisted loop invariant

A command inside a loop is *loop invariant code* if its execution has no effect after the first iteration of the loop. Typically, an assignment `y:=x+x` in a loop is invariant (provided `x` is not modified inside the loop). Loop invariants can safely be moved out of loops (*hoisted*) in order to make the program run slightly faster like in Figure 3.1. While loop invariant code is maybe not so frequent in source code, many transformations along the compilation process can generate some. For example, when compiling the editors `vim` or `emacs`, an average of 2 instructions per loop can be hoisted. These are mostly generated by other optimisations.

A command inside a loop is *quasi-invariant* if its execution has no effect after a finite number of iterations of the loop. Typically, if a loop contains the sequence `z:=y*y; y:=x+x` with `x` invariant, then `y:=x+x` is invariant. However, `z:=y*y` is **not** invariant. The first time the loop is executed, `z` will be assigned the old value of `y*y`, and only from the second time onward will `z` be assigned the value `(x+x)*(x+x)` which is invariant Figure 3.2. Hence, this command is *quasi-invariant*. It can still be hoisted out of the loop, but to do so requires to *peel* the loop first: execute its body once (by duplicating it before the loop). The number of times a loop must be executed before a quasi-invariant can be hoisted is called here the *degree* of the invariant.

```
while(i<100){
    z=y*y; //quasi-invariant
    use(z);
    y=x+x; //invariant
    use(y);
    i=i+1;
}
```

→

```
if(i<100){
  z=y*y;
  use(z);
  y=x+x;
  use(y);
  i=i+1;
}
if(i<100){
  z=y*y;
  use(z);
  use(y);
  i=i+1;
}
while(i<100){
  use(z);
  use(y);
  i=i+1;
}
```

Figure 3.2: Hoisted loop quasi-invariant

An obvious way to detect quasi-invariants is to first detect invariants (that is, quasi-invariants of degree 1) and hoist them; and iterate the process to find quasi-invariant of degree 2, and so on. This is, however, not very efficient since it may require a large number of iterations.

We provide here an analysis able to directly detect the invariance degree of any statements in the loop. Moreover, our analysis is able to assign an invariance degree not only to individual statements but also to groups of statements (called *chuncks*). That way it is possible, for example, to detect that a whole inner loop is invariant and hoist it, thus decreasing the asymptotic complexity of the program. This analysis and transformation has first been implemented as a *Proof of Concept* in a toy `C`-parser. Next, the analysis has been implemented as a prototype *pass* of the mainstream compiler LLVM and the transformation is under way. The prototype is

currently unable to handle several common situations (and leave them untouched) because of choices made for the sake of simplicity. It is, nonetheless, powerful enough to make significant progress compared to the existing loop invariant code motion techniques (it can handle many more cases).

Loop optimization techniques based on quasi-invariance are well-known in the compilers community[Aho et al., 1986]. The transformation idea is to unroll loops a finite number of time and hoist invariants until there is no more quasi-invariant. As far as we know, this technique is called "peeling" and it was introduced by Song *et al.* [Song et al., 2000]. Loop peeling and unrolling can also happen for entirely different reasons, mostly to improve vectorization and other optimizations [Fog, 2010]. In these cases, the decision to unroll is based on loop size and (predicted) number of iterations but not on the presence of quasi-invariants. It may, of course, happen that quasi-invariant removal is performed as a side effect of this unrolling, but only as a side effect and not as the main goal.

The present section offers a new point of view on invariant and quasi-invariant detection. Adapting ideas from an optimization on a `WHILE` language by Lars Kristiansen [Kristiansen, 2012], we provide a way to compute invariance degrees based on techniques developed in the field of Implicit Computational Complexity.

Implicit Computational Complexity (ICC) studies computational complexity using restrictions of languages and computational principles, providing results that do not depend on specific machine models. Based on static analysis, it helps to predict and control resources consumed by programs, and can offer reusable and tunable ideas and techniques for compilers. As said in the introduction, ICC mainly focuses on syntactic [Cobham, 1962, Bellantoni and Cook, 1992], type [Girard, 1987, Baillot and Terui, 2009] and Data Flow [Lee et al., 2001, Hofmann, 1999a, Kristiansen and Jones, 2009, Moyen, 2009] restrictions to provide bounds on programs' complexity. The present work was mainly inspired by the way ICC community uses different concepts to perform Data Flow Analysis, e.g. "Size-change Graphs" [Lee et al., 2001] or "Resource Control Graphs"[Moyen, 2009] which track data values' behavior and uses a matrix notation inspired by [Abel and Altenkirch, 2002], or "mwp-polynomials" [Kristiansen and Jones, 2009] to provide bounds on data size. For our analysis, we focus on dependencies between variables to detect invariance. Dependency graphs [Kuck et al., 1981] can have different types of arcs representing different kind of dependencies. Here we will use a kind of Dependence Graph Abstraction [Cocke, 1970] that can be used to find local and global quasi-invariants. Based on these techniques, we developed an analysis pass and we will implement the corresponding transformation in LLVM.

We propose a tool which is notably able to give enough information to peel and hoist an inner loop, thus automatically decrease the complexity of a program from $O(m * n)$ to $O(m + n)$.

3.1.1 State of the art on Quasi-Invariant detection in loop

Modern compilers find loop invariant code by recursively searching for variables whose value only depends on either code that is outside the loop; or other loop invariant code. To our knowledge, no compiler searches for loop quasi-invariant code.

A quasi-invariance detection has been described in [Song et al., 2000]. The authors define a *variable dependency graph* (VDG) and detect a loop quasi-invariant variable `x` if, among all paths ending at `x`, no path contain a node included in a circular path. Then they deduce an *invariant length* which corresponds to the length of the longest path ending in `x`. To our knowledge, this analysis has not been implemented in a compiler. Moreover, they only analyse individual commands and do not handle chunks. In the present section, this *length* is called *invariance degree* and presented further in subsection 3.4.1.

3.1.2 Contributions

This application lies between the fields of Implicit Computational Complexity and Compilation and provides significant advancement to both. To the authors' knowledge, this is the first application of ICC techniques on a mainstream compiler. One interest is that our tool potentially applies to programs written in any programming language managed by LLVM. Moreover, this work should be considered as a first step of a larger project that will make ICC techniques more accessible to programmers (section 2.2). Thus, we show that 25 years after Bellantoni & Cook breakthrough [Bellantoni and Cook, 1992], ICC techniques are ready to move into "the real world". On a more technical side, our tool aims at improving on currently implemented loop invariant detection and optimization techniques. The main LLVM tool for this purpose, the Loop Invariant Code Motion pass (LICM) combined with other passes (like `instcombine`), does not detect quasi-invariant of degree more than 3 (and not all of those of degree 2). More importantly, LICM will not detect quasi-invariant chunks, such as whole loops. Our tool, on the other hand, detects quasi-invariants of arbitrary degree and is able to deal with chunks. For instance the optimization shown in Figure 3.3 is performed neither by LLVM nor by GCC even at their maximum optimization level or with profiling methods.

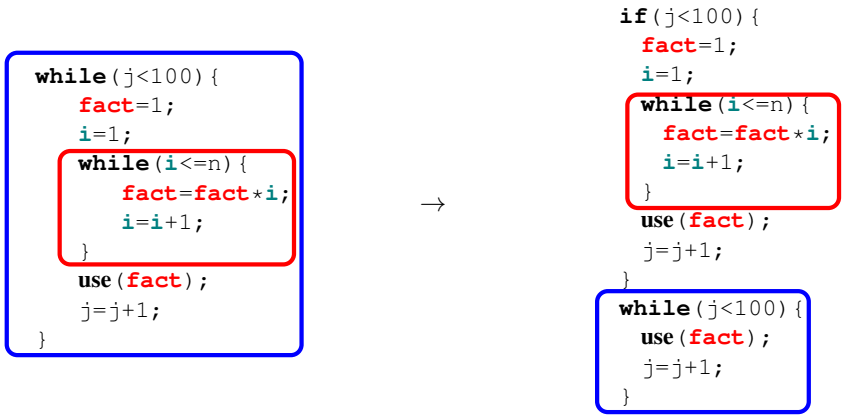


Figure 3.3: Hoisted invariant factorial

3.2 First Idea on WHILE-language

We already explained in section 2.1 why Lars Kristiansen started to be interested in resource-analysis theories and how he brought us to work on loop-invariants. Here it is his basic ideas of this potential peeling optimization he had in mind and written in [Kristiansen, 2012]. This idea is then reformulated and enhanced in the following section. First, it was focused on a WHILE-language as presented in Figure 2.2. Second, he introduced the notion of chunks with inputs and outputs here represented as `[X ← Y]` where `Y` (resp. `X`) is the set of variables in the input (resp. output) of this chunk. Third, he defined the notion of independence of chunks or commands and give a first theorem we will specialize further:

Definition 6 *Independence of commands (or chunks of commands).*

$$if\ \mathrm{Out}(\mathtt{C}_1) \cap \mathrm{In}(\mathtt{C}_2) = \emptyset\ then\ \mathtt{C}_1\ is\ independent\ to\ \mathtt{C}_2.$$

with $\mathrm{In}(\mathtt{C})$ *(resp.* $\mathrm{Out}(\mathtt{C})$*) the variables used (resp. modified or created) by the command* `C`

Theorem 3.2.1 *Let* $\mathtt{C} := \mathtt{while\ E\ do\ C}$ *be a command. Then*

$$[\![\mathtt{C}]\!] \equiv [\![\mathtt{if\ E\ do\ C; while\ E\ do\ C}]\!]$$

Lars proved that it is possible to peel part of the code by finding independents chunks like in the following

example:

$$\texttt{while i} < ? \texttt{ do } [[\texttt{X} \leftarrow \texttt{Y}];\ \texttt{i} := \texttt{i} + 1;\ \texttt{use(X)};\ [\texttt{Y} \leftarrow \texttt{Z}]]$$

we may transform it to:

$$\texttt{if i} < ? \texttt{ do } [[\texttt{X} \leftarrow \texttt{Y}];\ \texttt{i} := \texttt{i} + 1;\ \texttt{use(X)};\ [\texttt{Y} \leftarrow \texttt{Z}]];$$

$$\texttt{while i} < ? \texttt{ do } [[\texttt{X} \leftarrow \texttt{Y}];\ \texttt{i} := \texttt{i} + 1;\ \texttt{use(X)};]$$

And then into:

$$\texttt{if i} < ? \texttt{ do } [[\texttt{X} \leftarrow \texttt{Y}];\ \texttt{i} := \texttt{i} + 1;\ \texttt{use(X)};\ [\texttt{Y} \leftarrow \texttt{Z}]];$$

$$\texttt{if i} < ? \texttt{ do } [[\texttt{X} \leftarrow \texttt{Y}];\ \texttt{i} := \texttt{i} + 1;\ \texttt{use(X)};];$$

$$\texttt{while i} < ? \texttt{ do } [\ \texttt{i} := \texttt{i} + 1;\ \texttt{use(X)};]$$

Lars proposed a method to peel all the chunks at once by generating fresh variables. In the next section we try to enhance this method by adding new kind of dependencies and new algorithms to compute a dependency graph between chunks.

3.3 Improve the Theory with the previous DFA

In this section, we sketch the main lines of the theory around the *dependency flow graphs* (DepFG). Here dependency flow graphs are used to represent (weighted) relations between variables, that is relations that carry some additional information represented by elements of a *semi-ring*. In the next section, for instance, the semi-ring[1] $(\{0, 1, \infty\}, \max, \times)$ will be used to represent various kinds of dependencies between variables. Consequently, all examples will be given with this specific choice of semi-ring. We will work on the same `WHILE`-language as described in Figure 2.2.

This instance of data-flow graph for a given command `C` will be a weighted relation on the set V of variables involved in `C`. We now fix, until the end of this section, an arbitrary semi-ring $(\mathcal{S}, +, \times)$ corresponding to that DepFG.

[1] The convention here is that $0 \times \infty = 0$.

3.3.1 Kind of dependence

As mentioned previously, here we use the semi-ring $(\{0, 1, \infty\}, \max, \times)$ to represent dependencies: 0 will represent *reinitialization*, 1 will represent *propagation*, and ∞ will represent *dependence*. Figure 3.4 introduces both these notions and the graphical convention used throughout this chapter.

As previously, dependencies are represented by two types of arrows from variables on the left to variables on the right: plain arrows for *direct dependency*, dashed arrows for *propagation*. *Reinitialisation* of a variable z then corresponds to the absence of arrows ending on the right occurrence of z. Figure 3.4 illustrates these types of dependencies; let us stress here that the DEPFG would be the same if the assignment $y = y;$ were to be removed[2] from $\mathtt{C}$ since the value of y is still propagated. Finally $x = x + 1$ can be seen as $x_r = x_l + 1$[3] where x_r depends directly on x_l.

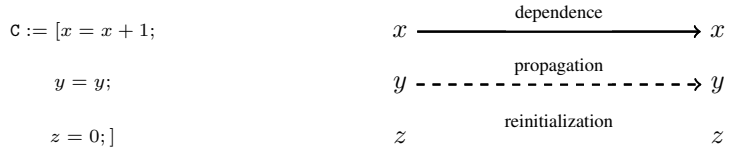


Figure 3.4: Types of dependence

Note that in this case of dependencies, $\mathrm{In}(\mathtt{C})$ (defined in definition 2) is exactly the set of variables that are source of a "dependence" arrow, while $\mathrm{Out}(\mathtt{C})$ is the set of variables that either are targets of dependence arrows or were reinitialised.

3.3.2 Constructing DEPFGs

The DEPFG of a command can be computed as described in subsection 2.2.2. The semi-ring is the same, $(\{0, 1, \infty\}, \max, \times)$. Here we will quickly represent the given examples previously with a corresponding command to match dependencies.

Composition and Multipaths. For the composition, there is no particular behaviour for the case of dependencies Figure 3.5 illustrate an example of DEPFG composition.

[2]Note that $y = y;$ does not create a direct dependence
[3]in SSA form

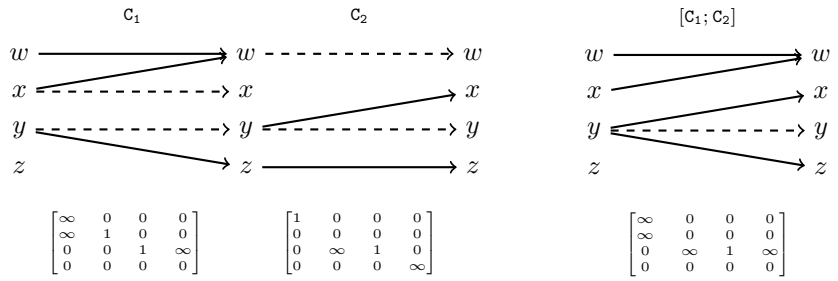


Figure 3.5: DEPFG of Composition.
Here $\mathtt{C}_1 := [w = w + x; z = y + 2;]$ and $\mathtt{C}_2 := [x = y; z = z * 2;]$

Conditionals. In the case of dependencies where $\mathtt{C} := \mathtt{if\ E\ then\ C}_1\ \mathtt{else\ C}_2$; we can notice that all modified variables in $\mathtt{C}_1$ and $\mathtt{C}_2$ should depend on the variables used in E. Denoting E the vector representing variables in[4] $\mathrm{Var}(\mathtt{E})$, O the vector representing variables in $\mathrm{Out}(\mathtt{C}_1) \cup \mathrm{Out}(\mathtt{C}_2)$, and $(\cdot)^t$ the matrix transpose, we define $\mathcal{C}^{\mathtt{C}}(\mathtt{E}) = E^t O$. Figure 3.6 illustrates this on an example.

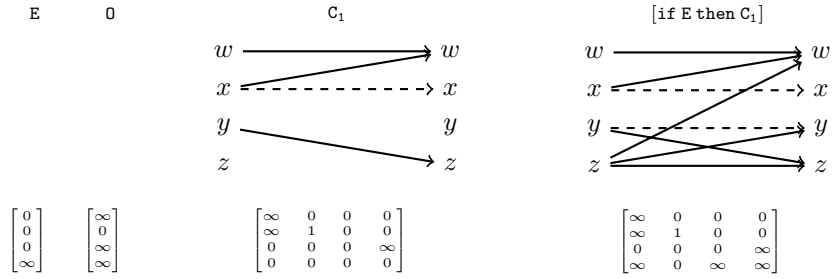


Figure 3.6: DEPFG of Conditional.
Here $\mathtt{E} := z \geq 0$ and $\mathtt{C}_1 := [w = w + x; z = y + 2; y = 0;]$;

While Loops In Figure 3.7 we define the DEPFG of a command $\mathtt{C} := \mathtt{while\ E\ do\ C}_1$; from $\mathrm{M}(\mathtt{C}_1)$. In the case of dependencies, the loop correction and the conditional correction coincide: $\mathcal{L}^{\mathtt{C}}(\mathtt{E}) = \mathcal{C}^{\mathtt{C}}(\mathtt{E})$.

3.4 Dependencies and Quasi-Invariants

We now study in more details the DEPFG representation of programs for the semiring $(\{0, 1, \infty\}, \max, \times)$. Each different weight represents different types of dependencies.

[4] I.e. the vector with a coefficient equal to ∞ for the variables in $\mathrm{Var}(\mathtt{E})$, and 0 for all others variables.

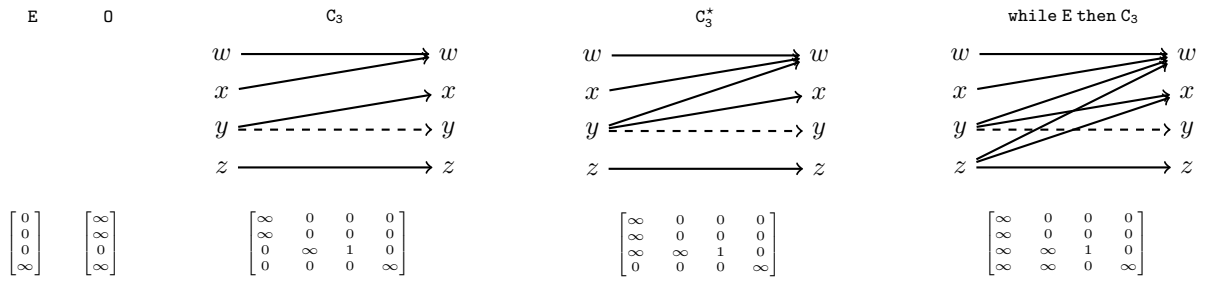


Figure 3.7: DEPFG of While Loop.
Here $\mathtt{E} := z \leq 100$ and $\mathtt{C_3} := [w = w + x; x = y; z = z + 1]$;

Each weight express how the values of the involved variables *after* the execution of the command depend on their values *before* the execution. There is a *direct* dependence between variables appearing in an expression and the variable on the left-hand side of the assignment. For instance x directly depends on y and z in the statement $x = y + z$;. When variables are unchanged by the command we call it *propagation*; this includes statements such as $x = x$;. Propagation only happens when a variable is not affected by the command, not when it is copied from another variable. If the variable is set to a constant, we call this a *reinitialization*.

3.4.1 Invariance Degree

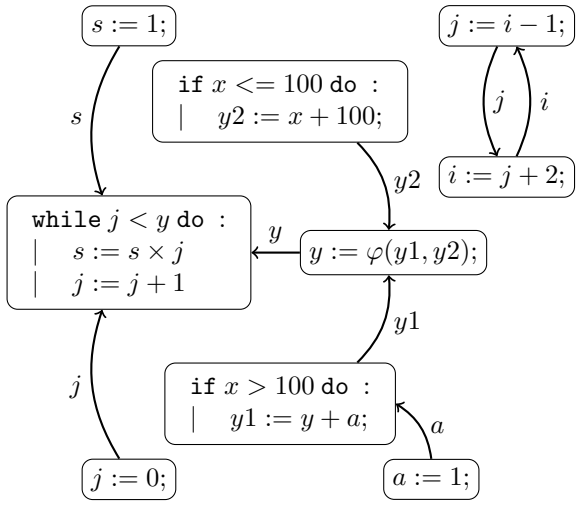
We now explain how, based on the computation of DEPFGs, we are able to define a notion of *dependence degree* for commands within a `while` loop. Based on this notion of degree, we show how the loop can be optimised by *peeling* it in order to extract all quasi-invariant commands, reducing the overall complexity while preserving the semantics.

Before going into details, the reader should be aware that the studied transformation applied on arbitrary `WHILE` programs gives rise to non-trivial renaming issues; in particular when a peeled conditional changes the value of a reused variable (we give an intuition of that in section 3.5.1). To simplify the exposition while being able to show an interesting example, we here introduce the *φ-function* that, at runtime, can choose the correct value of a variable depending on the path just taken. *φ-functions* are a standard tool of compilation, used when the code is set in *SSA form* (see section 3.6). Thus, we do not delve into details concerning how they precisely work and assume some sort of "black box" able to select the correct value.

We now consider a loop $\mathtt{C} := \mathtt{while\ E\ do}\ [\mathtt{C_1}; \mathtt{C_2}; \ldots, \mathtt{C_n}]$. We will build a *dependence graph* $\mathrm{Dep}(\mathtt{C})$ from the information given by the DEPFGs like in Figure 3.8.

```
i := 0;
x := 42;
y := 5;
a := 12;
while i < 100 do :
|   j := 0;
|   s := 1;
|   while j < y do :
|   |   s := s × j;
|   |   j := j + 1;
|   if x > 100 do :
|   |   y1 := x + a;
|   if x <= 100 do :
|   |   y2 := x + 100;
|   y := φ(y1, y2);
|   j := i − 1;
|   a := 1;
|   i := j + 2;
```

(a) An example



(b) Dependency graph of the outer `while` loop

Figure 3.8: Example of dependency graph

We will define the subset of *principal dependences* of the command $\mathtt{C_m}$ w.r.t. a given variable i. Intuitively, this principal dependence is the last command preceding $\mathtt{C_m}$ which modified the value of the variable i. However, since `while` and `if` commands may be skipped, we have to consider several main dependences in general. Based on this, we will then build the *dependence graph* which simply consists in writing the principal dependences of each command.

First, we introduce some notations. Given a variable i, we define the set $i^{\prec}$ as $\{\mathtt{C_k} \mid i \in \mathrm{Out}(\mathtt{C_k})\}$, the set of command modifying variable i. Given a command $\mathtt{C_m}$ and a variable $i \in \mathrm{In}(\mathtt{C_m})$, we denote as $\mathrm{PrD}_i(\mathtt{C_m})$ the subset of $i^{\prec}$ – the *use-def chain* over our ordered commands – defined as follows:

- it contains the *smallest* element of $i^{\prec}$ w.r.t. the order $<_m$ defined as

$$\mathtt{C_{m-1}} <_m \mathtt{C_{m-2}} <_m \cdots <_m \mathtt{C_1} <_m \mathtt{C_n} <_m \mathtt{C_{n-1}} <_m \cdots <_m \mathtt{C_m};$$

- if it contains a command $\mathtt{C}_h$ which is either a `while` or `if`, it contains the next element of $i^{\prec}$ w.r.t. the order $<_m$.

Definition 7 *Let* $\mathtt{C} := \mathtt{while\ E\ do}\ [\mathtt{C_1}; \mathtt{C_2}; \ldots, \mathtt{C_n}]$ *be a command. We define the directed graph* $\mathrm{Dep}(\mathtt{C})$ *as follows:*

- *the set of vertices* $V^{\mathrm{Dep}(\mathtt{C})}$ *is equal to* $\{\mathtt{C_1}, \ldots, \mathtt{C_n}\}$ *(the set of commands in the loop);*
- *the set of edges* $E^{\mathrm{Dep}(\mathtt{C})}$ *is equal to* $\uplus_{m=1}^{n} \uplus_{i \in \mathrm{In}(\mathtt{C_m})} \mathrm{PrD}_i(\mathtt{C_m})$ *(the set of all principal dependencies);*
- *the source* $s(i)$ *of the edge* $\mathtt{C_k} \in \mathrm{PrD}_i(\mathtt{C_m})$ *is* $\mathtt{C_k}$*;*
- *the target* $t(i)$ *of the edge* $\mathtt{C_k} \in \mathrm{PrD}_i(\mathtt{C_m})$ *is* $\mathtt{C_m}$.

The *invariance degree* $\deg_{\mathtt{C}}(\mathtt{C_m})$ of a command $\mathtt{C_m}$ w.r.t. $\mathtt{C}$ is then defined as follows. When clear, we will avoid writing the subscript $\mathtt{C}$ to ease notations. If $\mathtt{C_m}$ is a source in $\mathrm{Dep}(\mathtt{C})$, then $\deg(\mathtt{C_m}) = 1$. If $\mathtt{C_m}$ has a reflective edge in $\mathrm{Dep}(\mathtt{C})$, then $\deg(\mathtt{C_m}) = \infty$. Otherwise, we write $\mathrm{Fib}(\mathtt{C_m})$ – the *fiber* over $\mathtt{C_m}$ – the set of vertices in $\mathrm{Dep}(\mathtt{C})$ defined as $\{\mathtt{C_k} \mid \exists e \in E^{\mathrm{Dep}(\mathtt{C})}, s(e) = \mathtt{C_k}, t(e) = \mathtt{C_m}\}$, and define $\deg(\mathtt{C_m})$ by the following equation, where $\chi_{>m}(i) = 1$ if $i > m$ and $\chi_{>m}(i) = 0$ otherwise:

$$\deg(\mathtt{C_m}) = \max\left(\{\deg(\mathtt{C_i}) + \chi_{>m}(i) \mid \mathtt{C_i} \in \mathrm{Fib}(\mathtt{C_m})\}\right)$$

In particular, if $\mathtt{C_m}$ is part of a cycle in $\mathrm{Dep}(\mathtt{C})$, its degree is equal to ∞.

For all $i \in \mathbf{N} \cup \{\infty\}$, we define the inverse image $\deg^{-1}(\mathtt{i})$, i.e. $\deg^{-1}(\mathtt{i}) = \{\mathtt{C_k} \mid \deg(\mathtt{C})_k = i\}$, and we note $\mathrm{maxdeg}(\mathtt{C})$ the largest integer (i.e. not equal to ∞) such that $\deg^{-1}(\mathrm{maxdeg}(\mathtt{C})) \neq \emptyset$. The following lemma will be used in the proof of the main theorem.

Lemma 2 *Consider the set* $\deg^{-1}(\mathtt{i})$ *for an integer* $i > 0$ *and the relation induced from the dependency graph, i.e.* $\mathtt{C_i} \to \mathtt{C_j}$ *if and only if there is a sequence of edges from* $\mathtt{C_i}$ *to* $\mathtt{C_j}$ *in* $\mathrm{Dep}(\mathtt{C})$. *Then* $(\deg^{-1}(\mathtt{i}), \to)$ *is a partial order.*

Proof 1 *It is clear that* $\to$ *is transitive and reflexive. We only need to show that it is antisymmetric. I.e. that there are no two commands* $\mathtt{C_i}, \mathtt{C_j}$ *such that* $\mathtt{C_i} \to \mathtt{C_j}$ *and* $\mathtt{C_j} \to \mathtt{C_i}$. *We suppose that two such commands can be found and show it leads to a contradiction. Indeed, if such a situation arises, it means that the dependency graph contains a cycle* $\mathtt{P_1}, \ldots, \mathtt{P_k}$ *with* $\mathtt{P_1} = \mathtt{P_k} = \mathtt{C_i}$. *By definition of the degree, one has* $\deg(\mathtt{P_{i+1}}) \geqslant \deg(\mathtt{P_i})$. *More to the point, one has* $\deg(\mathtt{P_{i+1}}) > \deg(\mathtt{P_i})$ *as long as* $\mathtt{P_i} = \mathtt{C_k}$ *and* $\mathtt{P_{i+1}} = \mathtt{C_h}$ *with* $k > h$. *Now, it is clear that one of the inequalities* $\deg(\mathtt{P_{i+1}}) \geqslant \deg(\mathtt{P_i})$ *has to be strict, as no sequence* $\mathtt{C_{i_1}}, \mathtt{C_{i_2}}, \ldots, \mathtt{C_{i_k}}$ *with* $i_1 < i_2 < \cdots < i_k$ *can form a cycle. This implies that* $\deg(\mathtt{C_i}) > \deg(\mathtt{C_i})$*; a contradiction.*

Based on the invariance degree, we will be able to *peel* loops. For this purpose, we define the following notation. Given a sequence of commands $[\mathtt{C}_1;\mathtt{C}_2;\ldots;\mathtt{C}_n]$, we write $[\check{\mathtt{C}_1};\check{\mathtt{C}_2};\ldots;\check{\mathtt{C}_n}]^{(i)}$ the subsequence in which all commands of degree strictly less than i are removed. We then define $\mathtt{if}^i = \mathtt{if}\ \mathtt{E}\ \mathtt{then}\ [\check{\mathtt{C}_1};\check{\mathtt{C}_2};\ldots;\check{\mathtt{C}_n}]^{(i)}$, and $\mathtt{while}^i = \mathtt{while}\ \mathtt{E}\ \mathtt{then}\ [\check{\mathtt{C}_1};\check{\mathtt{C}_2};\ldots;\check{\mathtt{C}_n}]^{(i)}$. We can now state the main theorem of the section, denoting by $[\![\mathtt{C}]\!]$ the semantics of the command $\mathtt{C}$.

Theorem 3.4.1 *Let* $\mathtt{C} := \mathtt{while}\ \mathtt{E}\ \mathtt{do}\ [\mathtt{C}_1;\mathtt{C}_2;\ldots,\mathtt{C}_n]$ *be a command. Then*

$$[\![\mathtt{C}]\!] \equiv [\![\mathtt{if}^1;\mathtt{if}^2;\ldots;\mathtt{if}^{\mathrm{maxdeg}(\mathtt{C})};\mathtt{while}^\infty]\!]$$

The proof of this theorem is based on an induction, using the following lemma.

Lemma 3 *Let* $\mathtt{C} := \mathtt{while}\ \mathtt{E}\ \mathtt{do}\ [\mathtt{C}_1;\mathtt{C}_2;\ldots,\mathtt{C}_n]$*, and* $\mathtt{D} := \mathtt{if}^1;\mathtt{while}^2$*. Then* $[\![\mathtt{C}]\!] \equiv [\![\mathtt{D}]\!]$ *and for each command* $\mathtt{C}_m$ *appearing in* $\mathtt{while}^2$*,* $\deg_{\mathtt{C}}(\mathtt{C}_m) = \deg_{\mathtt{D}}(\mathtt{C}_m) + 1$.

Proof 2 *We start by proving the claim that for each command* $\mathtt{C}_m$ *appearing in* $\mathtt{while}^2$*,* $\deg_{\mathtt{C}}(\mathtt{C}_m) = \deg_{\mathtt{D}}(\mathtt{C}_m) + 1$*. This is a consequence of the fact that the dependency graph* $\mathrm{Dep}(\mathtt{D})$ *is obtained from* $\mathrm{Dep}(\mathtt{C})$ *by removing all vertices* $\mathtt{C}_m$ *for commands* $\mathtt{C}_m$ *of degree* 1*, together with their outgoing edges. Note that this defines a well-formed graph since by definition of the degree, a command of degree* 1 *may only depend on commands which are themselves of degree* 1 *(i.e. edges of target* $\mathtt{C}_m$ *are removed as well). Now, it is clear from the definition of the dependence degree that* $\deg_{\mathtt{C}}(\mathtt{C}_m) = \infty$ *implies* $\deg_{\mathtt{D}}(\mathtt{C}_m) = \infty$*, and that if* $\deg_{\mathtt{C}}(\mathtt{C}_m)) = d$ *the command* $\mathtt{C}_m$ *only depended commands of degree at most* d*. From Lemma 2 we can prove by induction that* $\deg_{\mathtt{D}}(\mathtt{C}_m) = d - 1$.

Then, one should realise that commands of degree 1 *are in fact invariants of the loop* $\mathtt{C}$*. It is then clear that* $[\![\mathtt{C}]\!] \equiv [\![\mathtt{D}]\!]$.

3.5 A proof of Concept from C to C

In the previous section, we have seen that the transformation is possible from and to a WHILE-language. This section will progressively show that we can do it in real languages by introducing our implementations. First it will present our *proof of concept*[5] which does both analysis and propose a transformation from C to C. After

[5] https://github.com/ThomasRuby/LQICM_On_C_Toy_Parser

that, we will explain how we implemented a prototype analysis in a mainstream compiler.

3.5.1 PoC in Python over a toy parser

To easily and quickly integrate our transformation, we decided to use "pycparser"[6], a `C` parser written in `Python`. The principal interest was to simply get and manipulate an Abstract Syntax Tree Figure 3.9. Using a "WhileVisitor" we list all nested `while`-loops, then, with a depth-first strategy (the inner loop first), this tool analyses and transforms the code if an invariant or quasi-invariant is detected. The analysis is divided in two parts: the DEPFG construction and the invariance degree computation.

Analysis

The first part aims to build the DEPFG and to list relations between statements. In this implementation we decided to define a DEPFG object as a matrix of pairs of bit and defining simply the two operations of the semi-ring:

```
# Empty, Propagation, Dependence, Total (Propagation and Dependence)
Keys = [(0,0),(1,0),(0,1),(1,1)]

    # Product
    dicProd = {
        (0,0): {(0,0): (0,0), (1,0): (0,0), (0,1): (0,0), (1,1): (0,0)},
        (1,0): {(0,0): (0,0), (1,0): (1,0), (0,1): (0,1), (1,1): (1,1)},
        (0,1): {(0,0): (0,0), (1,0): (0,1), (0,1): (0,1), (1,1): (0,1)},
        (1,1): {(0,0): (0,0), (1,0): (1,1), (0,1): (0,1), (1,1): (1,1)}
    }
    # Sum
    dicSum = {
        (0,0): {(0,0): (0,0), (1,0): (1,0), (0,1): (0,1), (1,1): (1,1)},
        (1,0): {(0,0): (1,0), (1,0): (1,0), (0,1): (1,1), (1,1): (1,1)},
        (0,1): {(0,0): (0,1), (1,0): (1,1), (0,1): (0,1), (1,1): (1,1)},
        (1,1): {(0,0): (1,1), (1,0): (1,1), (0,1): (1,1), (1,1): (1,1)}
    }
```

Here we can remark that we keep the information that a variable can be (in the same time as for quantum mechanism) strongly connected and propagated. This choice will be explain later.

[6] `https://github.com/eliben/pycparser`

```
while(j<n){
    fact=1;       //0
    i=1;          //1
    while (i<=y) { //2
        fact=fact*i;
        i=i+1;
    }
    if(x<100){    //3
        y=x+a;
    }
    if(x>=100){ //4
        y=x+100;
    }
    a = 12;       //5
    i=j+1;        //6
    j=i+1;        //7
}
```

```
While:
  BinaryOp <cond>: op=<
    ID <left>: name=j
    ID <right>: name=n
  Compound <stmt>:
    Assignment <block_items[0]>: op==
      ID <lvalue>: name=fact
      Constant <rvalue>: type=int, value=1
    Assignment <block_items[1]>: op==
      ID <lvalue>: name=i
      Constant <rvalue>: type=int, value=1
    While <block_items[2]>:
      BinaryOp <cond>: op=<=
        ID <left>: name=i
        ID <right>: name=y
      Compound <stmt>:
        Assignment <block_items[0]>: op==
          ID <lvalue>: name=fact
          BinaryOp <rvalue>: op=*
            ID <left>: name=fact
            ID <right>: name=i
        Assignment <block_items[1]>: op==
          ID <lvalue>: name=i
          BinaryOp <rvalue>: op=+
            ID <left>: name=i
            Constant <right>: type=int, value=1
    If <block_items[3]>:
      BinaryOp <cond>: op=<
        ID <left>: name=x
        Constant <right>: type=int, value=100
      Compound <iftrue>:
        Assignment <block_items[0]>: op==
          ID <lvalue>: name=y
          BinaryOp <rvalue>: op=+
            ID <left>: name=x
            ID <right>: name=a
    If <block_items[4]>:
      BinaryOp <cond>: op=>=
        ID <left>: name=x
        Constant <right>: type=int, value=100
      Compound <iftrue>:
        Assignment <block_items[0]>: op==
          ID <lvalue>: name=y
          BinaryOp <rvalue>: op=+
            ID <left>: name=x
            Constant <right>: type=int, value=100
    Assignment <block_items[5]>: op==
      ID <lvalue>: name=a
      Constant <rvalue>: type=int, value=12
    Assignment <block_items[6]>: op==
      ID <lvalue>: name=i
      BinaryOp <rvalue>: op=+
        ID <left>: name=j
        Constant <right>: type=int, value=1
    Assignment <block_items[7]>: op==
      ID <lvalue>: name=j
      BinaryOp <rvalue>: op=+
        ID <left>: name=i
        Constant <right>: type=int, value=1
```

Figure 3.9: Hard factorial example with AST

A DEPFG is computed for each command using a top-down strategy following the dominance tree. The DEPFG is composed when the corresponding command is a sequence of commands. As described previously, we compute the correction and the maximum relations possible for a `while` or `if` statement. In Figure 3.10 it is the DEPFG for the inner `while`.

```
While:
  BinaryOp <cond>: op=<=
    ID <left>: name=i
    ID <right>: name=y
  Compound <stmt>:
    Assignment <block_items[0]>: op==
      ID <lvalue>: name=fact
      BinaryOp <rvalue>: op=*
        ID <left>: name=fact
        ID <right>: name=i
    Assignment <block_items[1]>: op==
      ID <lvalue>: name=i
      BinaryOp <rvalue>: op=+
        ID <left>: name=i
        Constant <right>: type=int, value=1
i      |      (1, 1)   (0, 1)   (0, 0)
fact   |      (0, 0)   (1, 1)   (0, 0)
y      |      (0, 1)   (0, 1)   (1, 0)
```

Figure 3.10: Inner loop DEPFG

With this DEPFG, we compute an invariance degree for each statement in the loop regarding to the relations listed Figure 3.11. Here the dependencies are listed regarding to the order of the `block_items` of the main node, the outer `while` statement. For instance the tuple "`(1,2,'i')`" means C_1 (aka `i=1;`) influences *strongly* C_2 (aka the `while` statement) with the variable `i`. The list of list of dependencies below shows which commands depend on which other commands inside the main loop. Here the third list "`[1, 0, 4, 3]`" says that C_2 depends of the commands C_1,C_0,C_4 and C_3. Remark that this is $\mathrm{PrD}_i(C_2)$, the ordered subset described in subsection 3.4.1.

```
Dependence Graph:
[(1, 2, 'i'), (0, 2, 'fact'), (4, 2, 'y'), (3, 2, 'y'), (5, 3, 'a'), (7, 6, 'j'), (6, 7, 'i')]

List of List of Dependencies:
[[], [], [1, 0, 4, 3], [5], [], [], [7], [6]]
```

Figure 3.11: Relations between chunks

```
    (initialisations not shown)

 7      while (i<n) {
 8          j=0;                    // 1
 9          s=1;                    // 1
10          while (j<y) {           // 3
11              s=s*j;
12              j=j+1;
13          }
14          if (x>100) {            // 2
15              y1=x+a;
16          }
17          if (x<=100) {           // 1
18              y2=x+100;
19          }
20          y=φ(y1,y2);             // 2
21          a=1;                    // 1
22          j=i-1;                  // ∞
23          i=j+2;                  // ∞
24      }
25      return i;
26  }
```

peeling

```
 7      if (i<n) {
 8          j=0;
 9          j1=j;
10          s=1;
11          s1=s;
12          while (j<y) {...}
13          if (x>100) {...}
14          if (x<=100) {...}
15          y21=y2;
16          y=φ(y1,y2);
17          a=1;
18          j=i-1;
19          i=j+2;
20      }
21      if (i<n) {
22          j=j1;
23          s=s1;
24          while (j<y) {...}
25          if (x>100) {...}
26          y12=y1;
27          y2=y21;
28          y=φ(y1,y2);
29          j=i-1;
30          i=j+2;
31      }
32      if (i<n) {
33          j=j1;
34          s=s1;
35          while (j<y) {...}
36          y1=y12;
37          y2=y21;
38          y=φ(y1,y2);
39          j=i-1;
40          i=j+2;
41      }
42      while (i<n) {
43          j=i-1;
44          i=j+2;
45      }
46      return i;
47  }
```

Figure 3.12: Hoisting inner loop

Peeling loops from `C` to `C`

On a non-SSA form (see section 3.6), variables are often reused to store temporary values. The problem is that if we hoist a part of loop which changes the value of one of those variables it is possible to change the semantic. Furthermore, it is harder if those variables are modified in a conditional chunk, in this case a *φ-function* is needed. This issue is illustrated in Figure 3.12 if we replace `y1`, `y2` by `y` and *φ-function* is removed.

Implementation details

For this PoC we decided to fix the renaming issue by adding variables to save values (including booleans values) after each hoisted assignments and restore them at the beginning of each new hoisted block like in Figure 3.13. But this trick is not relevant anymore at IR level. Indeed, these are already standard in compilers, our ultimate goal, and there is no need to rewrite a full "put into SSA form" algorithm. The PoC is nonetheless able to peel `C` programs in SSA form with no invariant conditional statements (see the examples in the repository).

This implementation is almost 700 lines of `Python`. It is able to compute relations of each commands or sequence of commands. This tool focuses on a restricted `C` syntax and considers all functions as non-pure. Functions with side effects can be seen as an anchor in the sequence of statements, commands can not be moved around. But we can restrain the conditions for peeling. We can theoretically hoisting pure functions as in [Song et al., 2000]. All other side effects can be broken by this transformation, and thus should not be moved.

```
while(j<n){
   fact=1;      //0
   i=1;         //1
   while (i<=y) { //2
      fact=fact*i;
      i=i+1;
   }
   if(x<100){   //3
      y=x+a;
   }
   if(x>=100){ //4
      y=x+100;
   }
   a = 12;      //5
   i=j+1;       //6
   j=i+1;       //7
}
```

→

```
if (j < n){
   fact = 1;
   å0 = fact;
   i = 1;
   å2 = i;
   while (i <= y){
      fact = fact * i;
      i = i + 1;
   }
   if (x < 100){
      y = x + a;
   }
   if (x >= 100){
      y = x + 100;
   }
   å4 = y;
   å5 = x >= 100;
   a = 12;
   i = j + 1;
   j = i + 1;
}
if (j < n){
   fact = å0;
   i = å2;
   while (i <= y){
      fact = fact * i;
      i = i + 1;
   }
   if (x < 100){
      y = x + a;
   }
   if (å5){
      y = å4;
   }
   i = j + 1;
   j = i + 1;
}
if (j < n){
   fact = å0;
   i = å2;
   while (i <= y){
      fact = fact * i;
      i = i + 1;
   }
   i = j + 1;
   j = i + 1;
}
while (j < n){
   i = j + 1;
   j = i + 1;
}
```

Figure 3.13: Renaming issue in C

3.6 A prototype in LLVM

Compilers, and especially LLVM on which we are working, use an *Intermediate Representation* (IR) presented in subsection 1.1.2 to handle programs. This is a typed assembly-like language that is used during all the stages of the compilation. Programs (in various different languages) are first translated into the IR, then several optimizations are performed (implemented in so-called *passes*), and finally the resulting IR is translated again in actual assembly language depending on the machine it will run on. Using a common IR allows to do the same optimizations on several different source languages and for several different target architectures.

One important feature of the LLVM IR is the *Single Static Assignment* form (SSA) also presented in subsection 1.1.2. A program is in SSA form if each variable is assigned at most once. In other words, setting a program in SSA form requires a massive α-conversion of all the variables to ensure uniqueness of names. The advantages are obvious since this removes any name-aliasing problem and ease analysis and transformation.

The main drawback of SSA comes when several different paths in the Control Flow reach the same point (typically, after a conditional). Then, the values used after this point may come from any branch and this cannot be statically decided. For example, if the original program is `if (y) then x:=0 else x:=1;C`, it is relatively easy to turn it into a pseudo-SSA form by α-converting the `x`: `if (y) then` x_0`:=0 else` x_1`:=1;C` but we do not know in `C` which of x_0 or x_1 should be used.

SSA solves this problem by using φ*-functions*. That is, the correct SSA form will be `if (y) then` x_0`:=0 else` x_1`:=1;` $X:=\varphi(x_0, x_1)$`; C`.

While the SSA itself eases the analysis, we do have to take into account the φ functions and handle them correctly.

Existing. LLVM does have a Loop Invariant Code Motion (LICM) pass which hoists invariants out of loops. Used with unrolling and instruction combination optimizations it can sometimes "peel" quasi-invariants. However, as far as we know, it does not compute invariance degrees and does not detect quasi-invariant chunks. Hence, if peeling occurs, it is as a side effect of another transformation (mostly, pipeline optimisation) and not to hoist quasi-invariants.

3.6.1 Preliminaries

Our pass is currently working over the branch "release_40" of LLVM[7]. First, we have written the framework in a library (in the file `LQICMUtils.h`) in which we define what is a `Relation`, an object to represent DEPFG. This `Relation` is mainly composed by `variables` (which is a set of `LLVM::Values`) and a map of dependencies which link a type of dependence for a pair of variables:

```
typedef std::pair<Value*,Value*> Arrow;
typedef DenseMap<Arrow,DepType> DepMap;
typedef SmallPtrSet<Value*,8> VSet;
```

Here we need to think about each instruction to consider what could be the corresponding DEPFG of each? We could consider that all operands create a strong dependence to the value but it is not really true. For instance a φ-instruction create more propagations as in Figure 3.14: We discuss our choice further.

$$\begin{bmatrix} 1 & \infty & 0 \\ 0 & 0 & 0 \\ 0 & \infty & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

Figure 3.14: DEPFG of φ instruction.
$\%x \;=\; phi\ i32\,[\,\%w,\ \%while.end\,],\ [\,\%y,\ \%while.cond\,]$

[7] https://github.com/llvm-mirror/llvm/tree/release_40

A dependency type is defined regarding to the semi-ring given:

```
enum DepType {
  EMPT, // (0,0)
  PROP, // (1,0)
  STRO, // (0,1)
  TOTA, // (1,1)
};

DepType Plus[4][4] = { {EMPT,PROP,STRO,TOTA},
                       {PROP,PROP,TOTA,TOTA},
                       {STRO,TOTA,STRO,TOTA},
                       {TOTA,TOTA,TOTA,TOTA}
                     };
DepType Time[4][4] = { {EMPT,EMPT,EMPT,EMPT},
                       {EMPT,PROP,STRO,TOTA},
                       {EMPT,STRO,STRO,TOTA},
                       {EMPT,TOTA,TOTA,TOTA}
                     };
```

Again here we have kept the information of a potential dependence which is a propagation plus a strong dependence. We will see how this can improve our analysis. Obviously the framework comes with special function to manipulate DFGs here DEPFGs.

Also we can see how a `Chunk` is represented in our pass. A `Chunk` is basically a map which links a set of commands to corresponding `Relations`. It has his own computed `Relation` too. It is delimited by two `LLVM::BasicBlocks`, most importantly an "end" block which is the *unique* successor of the chunk and contains some information about its form: is it a loop, an if-statement, a branch or a simple block. A chunk of if-statement is initialized when a fork is encountered (a block with two successors), a chunk is initialized for each branch in a way to be "summed" when their first common post dominator is encountered. For the moment it is the only case considered, loops with several exit-blocks are not considered (yet).

Secondly, we want to visit all loops using a deep-first strategy (the inner loop first). Then, as for the LICM, our pass is derived from the basic `LoopPass` (loops are detected by the LLVM's `LoopInfo` pass).

Which means that each time a loop is encountered, our analysis is performed. At this point, the purpose is to gather the relations of all instructions in the loop to compose them and provide the final relation for the entire current chunk loop. Then a `Relation` is generated for each command using a top-down strategy following the dominance tree. The *SSA* form helps us to gather dependence information on instructions. By visiting operands of each assignment, it is easy to build our map of `Relation`. With all the current loop's relations gathered, we compute the compositions, condition corrections and the maximums relations possible as described in section 2.2.2. Obviously this method can be enhanced by an analysis on bounds around conditional and number of iterations for a loop. Finally, with those composed relations we compute a *dependency graph*. This graph is defined as `DepMapChunks` and is a list of list as in Figure 3.11. Those *principal dependences* are computed as described in Figure 3.4.1. And here, because φ(and select)-instructions are considered as two potentials propagations we must look at the two values and figure out what are the new "closest" dependencies. It is then possible to have from one variable to another a strong dependency and a propagation (i.e. the insert instruction add two potential propagation with a strong condition correction). In this case we still need to look at the previous affectation to add potential previous dependencies. The reader can have a look to the recursive function called "`computeDepForX`" which computes for a variable (in input of a `Relation` here) the closest strong dependence. From this dependency graph it is possible to compute an invariance degree for each statement in the loop like in Figure 3.15. algorithm 1 gives the algorithm which computes those degrees.

```
...
while.body:
  %mul = mul nsw i32 %y.0, %y.0 ← 2
  call void @use(i32 %mul)      ← ∞
  %add = add nsw i32 %rem, %rem ← 1
  call void @use(i32 %add)      ← ∞
  %inc = add nuw nsw i32 %i.0, 1 ← ∞
  br label %while.cond          ← ∞
...
```

```
...
   while(i<100){
      z=y*y;
      use(z);
      y=x+x;
      use(y);
      i++;
   }
...
```

Figure 3.15: Invariance degrees example.

This algorithm is dynamic. It stores progressively each degree needed to compute the current one and reuse them. Note that, for the initialization part, we are using LLVM methods (`canSinkOrHoist`, `isGuaranteedToExecute` etc...) to figure out if an instruction is movable or not. These methods provide the anchors instructions for the current loop.

Data: Dependency Graph and Dominance Graph
Result: List of invariance degree for each command
Initialize degrees of `use` to ∞ and others to 0;
for *each command* $\mathtt{C_m}$ **do**
  if *the current degree* $\deg(\mathtt{C_m}) \neq 0$ **then**
    `return` $\deg(\mathtt{C_m})$;
  else
    Initialize the current degree $\deg(\mathtt{C_m})$ to ∞;
    if *there is no dependence for the current chunk* **then**
      $\deg(\mathtt{C_m}) = 1$;
    else
      for *each dependent command ordered (subsection 3.4.1) compute the degree* $\deg(\mathtt{C_d})$ **do**
        if $\deg(\mathtt{C_d}) = \infty$ ***or*** $\mathtt{C_d} = \mathtt{C_m}$ **then**
          `return` ∞;
        end
        if $\deg(\mathtt{C_m}) \leq \deg(\mathtt{C_d})$ ***and*** $d > m$ **then**
          $\deg(\mathtt{C_m}) = \deg(\mathtt{C_d}) + 1$;
        else
          $\deg(\mathtt{C_m}) = \deg(\mathtt{C_d})$;
        end
      end
    end
    `return` $\deg(\mathtt{C_m})$
  end
end

Algorithm 1: Invariance degree computation.

3.6.2 Peeling loop idea

The transformation will consist in creating as many basic blocks before the loop as needed to remove all quasi-invariants as in 3.4.1. For each block created, we include every commands with a higher or equal invariance degree. For instance, the first `preheader` block will contain every commands with an invariance degree higher or equal to 1; the second one, higher or equal to 2 etc... to maxdeg. The final loop will contain every commands with an invariance degree equal to ∞. In the basic example of the same factorial computed several times Figure 3.16, the entire inner loop (in red) is hoisted out of the parent loop (in blue) as shown in Figure 3.17. More technically we chose to reused and modify `LoopUnrollPeel.cpp`.

Of course, hoisting quasi-invariants of high degrees is not necessarily a good idea since it requires peeling the loop many times and thus greatly increase the size of the code. The final decision to hoist or not will depend on the max quasi-invariance degree and the size of the loop. It will be possible to enable a default value for both or take those values throughout compilation options given by the user.

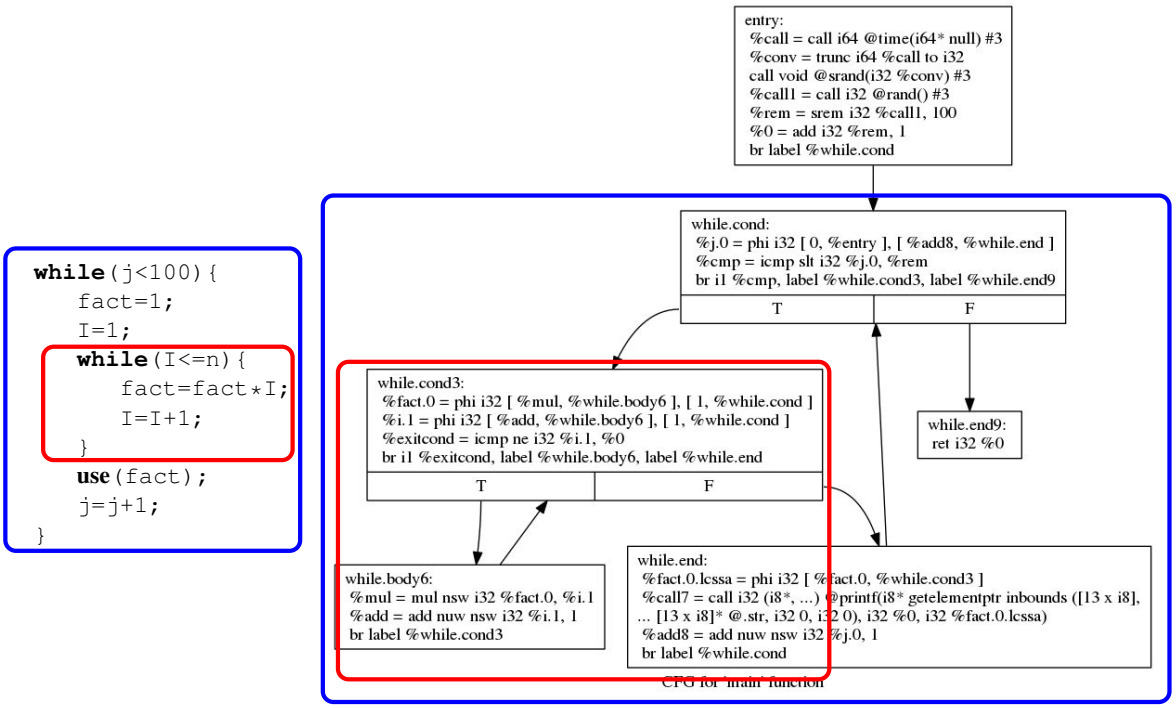


Figure 3.16: Simple factorial CFG.

3.6.3 Results

In this example (Figure 3.12), we compute the same factorial several times. We can detect it statically, so the compiler has to optimize it at least in `-O3`. Our tests showed that is done neither in LLVM nor in GCC (we also tried `-fpeel_loops` with profiling). The generated assembly shows the factorial computation in the inner loop. Moreover, the computation time of this kind of algorithm compiled with `clang` in `-O3` still computes n times the inner loop so the computation time is quadratic, while hoisting it results in linear time. For the example shown in Figure 3.12 (LLVM-IR in Figure 3.18), our pass will compute the correct degrees.

To each instruction printed corresponds an invariance degree. The assignment instructions are listed by loops, the inner loop (starting with `while.cond5`) and the outer loop (starting with `while.cond`). The inner loop has its own invariance degree equal to 4 (line `10`). Remark that we do consider the `phi` initialization instructions of an inner loop. Here `%fact.0` and `%i.1` are reinitialized in the inner loop condition block. So `phi` instructions are analyzed in two different cases: to compute the relation of the current loop or to give the initialization of a variable sent to an inner loop. Our analysis only takes the relevant operand regarding to the current case and do not consider others.

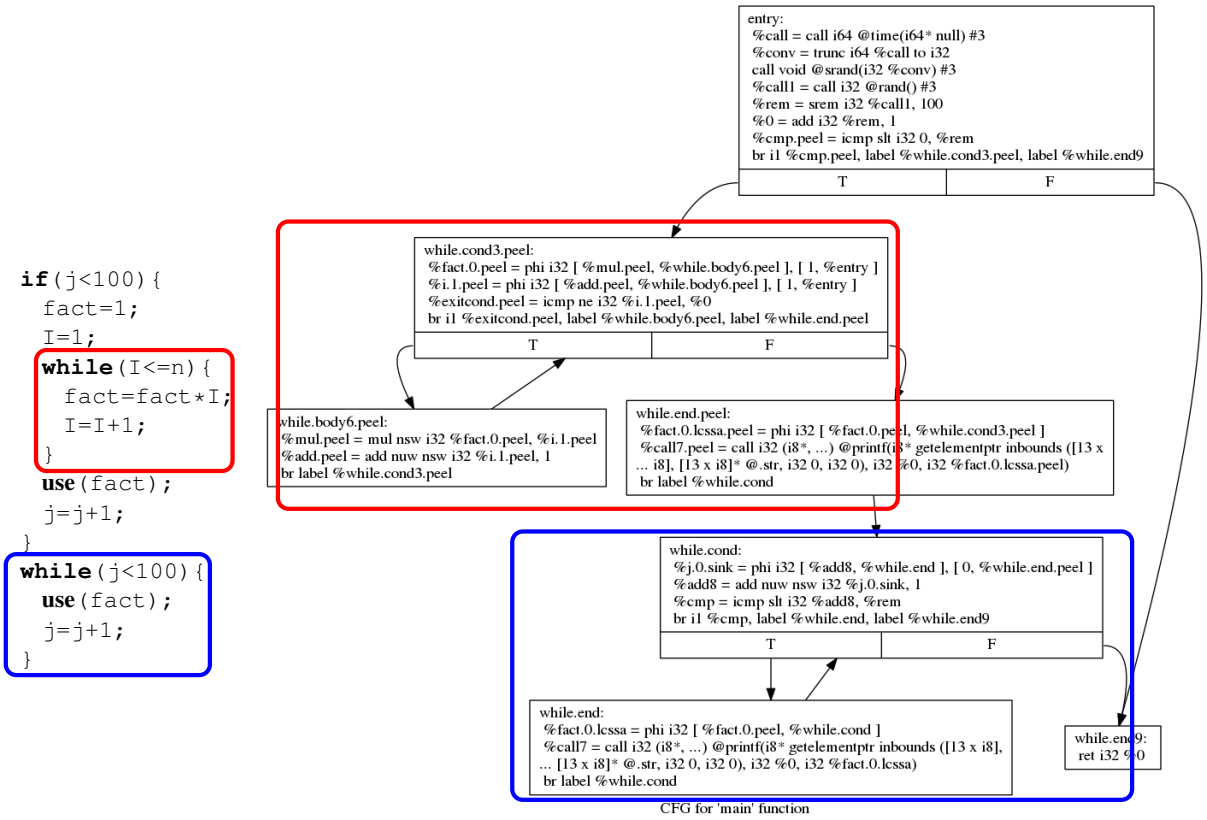


Figure 3.17: Peeled Simple factorial CFG.

Statistics have been generated by our pass on the editor `vim` to evaluate the magnitude of new possible optimizations (Figure 3.19). Note that the result changes a lot regarding to when our pass is called. Here, to compare, it is placed before all `LICM` iterations but without peeling. We can observe that, despite the number of aborted analysis due to the lack of flexibility of our young pass, over less than half of the loops analyzes `LICM`, we find 5984 quasi-invariants instructions, 73% of the total invariants currently hoisted by `LICM`. The compilation time increased by around 23%.

The code of this pass is available online[8]. To provide some real benchmarks on large programs we need to finish the transformation. We are currently working on it.

For the moment the transformation is not enough stable to be performed on long compilation like this. It only works on our specific examples. We are currently working on the stability of our pass.

[8] `https://github.com/ThomasRuby/lqicm_pass`

```
...
while.cond:
  %j.0 = phi i32 [ 0, %entry ], [ %add20, %while.end ]
  %y.0 = phi i32 [ 5, %entry ], [ %y.2, %while.end ]
  %i.0 = phi i32 [ undef, %entry ], [ %j.0, %while.end ]
  %a.0 = phi i32 [ 5, %entry ], [ 0, %while.end ]
  %cmp = icmp slt i32 %j.0, %rem
  br i1 %cmp, label %while.cond5, label %while.end21

while.cond5:
  %fact.0 = phi i32 [ %mul, %while.body8 ], [ 1, %while.cond ]
  %i.1 = phi i32 [ %add, %while.body8 ], [ 1, %while.cond ]
  %cmp6 = icmp sle i32 %i.1, %y.0
  br i1 %cmp6, label %while.body8, label %while.end

while.body8:
  %mul = mul nsw i32 %fact.0, %i.1
  %add = add nsw i32 %i.1, 1
  br label %while.cond5

while.end:
  %cmp10 = icmp sgt i32 %rem3, 100
  %add12 = add nsw i32 %rem3, %a.0
  %add12.y.0 = select i1 %cmp10, i32 %add12, i32 %y.0
  %cmp14 = icmp sle i32 %rem3, 100
  %add17 = add nsw i32 %rem3, 100
  %y.2 = select i1 %cmp14, i32 %add17, i32 %add12.y.0
  %add20 = add nsw i32 %j.0, 1
  br label %while.cond

while.end21:
  ret i32 %i.0
...
```

Figure 3.18: LLVM Intermediate Representation

```
(LQICM Analysis called before each LICM(3X) occurence)
Compiler: clang release_40 -Oz
Time (with - without)
8m8,020s - 7m48,244s
--- vim v8.00442 ---
13407   Number of loop
5009    Loops well analyzed by LQICM
5984    LQICM Quasi-Invariants detected
8126    LICM Invariants Hoisted
656     LQICM Quasi-Invariants blocks detected
7632    LQICM Aborted: several exit blocks
351     LQICM Aborted: Header not exiting
256     LQICM Aborted: Inner loop not analyzed
159     LQICM Aborted: Successor not found
```

Figure 3.19: Statistics on vim.

3.6.4 Implementation details

The only chunks considered in the current implementation are the one consisting of `while` (any loops in LCSSA form) or `if-then-else` (any forks which have a common post dominator existing in the loop) statements.

This implementation (including a preliminary version of peeling) is almost 3000 lines of `C++`. It is able to compute relations of each commands or sequence of commands. However, it has, for the moment, some restrictions on the form of the loop analyzed. First, loops with several exit blocks are ignored and left intact (typically a loop including a `break`); furthermore, this tool considers all functions as non-pure as for the Proof of Concept. Even with these restrictions, the pass is able to optimise code that was previously left untouched, thus illustrating the power of the method. We are currently trying to improve the flexibility of our pass.

3.6.5 Conclusion

Developers expect that compilers provide certain more or less "obvious" optimizations. When peeling is possible, that often means: either the code was generated; or the developers prefer this form (for readability reasons) and expect that it will be optimized by the compiler; or the developers haven't seen the possible optimization (mainly because of the obfuscation level of a given code).

Our generic pass is able to provide a reusable abstract dependency graph and the quasi-invariance degrees for further loop optimization or analysis.

Conclusions and Further works

Conclusions

The study of Implicit Computational Complexity is 20 years old and keeps playing with toy languages. This thesis is another step into "real world" programs for this community mainly because it tries to accompany the researcher into a widely used compiler with a new customizable Data Flow Analysis framework.

We have seen that those implementations of ICC methods need to deal with more technical cases thus generates more problematics often followed by more approximations but they still feasible. Now it is time to develop them the best way we can to show the real value of those methods. And in another hand, a lot of interesting certifications do not need accurate analysis, filling the gap should not be so difficult. It could bring safeness and avoid to waste time in testing.

We've also seen that optimizations are not so far from analyses. It is possible to deal with both verifications and optimizations as researchers around *Compcert* have proven. Providing certificates at compile time is not a dream, it feasible and methods are already here for it. This could offer reliability to all users.

This thesis, more that bringing interest on complexity analysis, it is providing tools as proof of concepts and prototypes. Particularly LQICM pass has, from my point of view, a good potential to be the first brick of a future "Loop Invariant Code Motion" like pass. I mean that it could be widely used in compilers.

Further works

LQICM pass is currently a prototype. The transformation is still in preliminary form and even the analysis is making some approximations (*e.g.* considering all functions as non-pure) that hamper its efficiency. We will obviously work further on the pass to finish the transformation and increase the number of cases we can handle.

On a more theoretical side, the current analysis is strongly inspired by other ICC analysis such as *Size Change Termination* [Lee et al., 2001] (from which the Data Flow Graphs and Multipaths are taken) or the mwp-analysis (from which the loop correction idea is taken) [Kristiansen and Jones, 2009]. As shown in section 2.3, it is easy to adapt the method to similar analysis, and most of the existing code can be reused. Thus, we plan on implementing a mwp-inspired complexity analysis in LLVM, which should be able to guarantee the polynomiality of large parts of the code. We hope that continuing working in the frontier of both field will expose new ideas and methods that we haven't considered yet.

Bibliography

[Abel and Altenkirch, 2002] Abel, A. and Altenkirch, T. (2002). A Predicative Analysis of Structural Recursion. *Journal of Functional Programming*, 12(1).

[Aho et al., 1986] Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers principles, techniques, and tools*. Addison-Wesley, Reading, MA.

[Albert et al., 2008a] Albert, E., Arenas, P., Genaim, S., Puebla, G., Ramírez, D., and Zanardini, D. (2008a). The COSTA Cost and Termination Analyzer for Java Bytecode and its Web Interface (Tool Demo). In Philippou, A., editor, *22nd European Conference on Object-Oriented Programming*.

[Albert et al., 2008b] Albert, E., Arenas, P., Genaim, S., Puebla, G., Ramírez, D., and Zanardini, D. (2008b). Upper Bounds of Resource Usage for Java Bytecode using COSTA and its Web Interface. In *Workshop on Resource Analysis*.

[Alias et al., 2010] Alias, C., Darte, A., Feautrier, P., and Gonnord, L. (2010). *Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs*, pages 117–133. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Alias et al., 2013] Alias, C., Darte, A., Feautrier, P., and Gonnord, L. (2013). Rank: a tool to check program termination and computational complexity. In *Constraints in Software Testing Verification and Analysis*, Luxembourg.

[Avanzini, 2013] Avanzini, M. (2013). *Verifying Polytime Computability Automatically*. PhD thesis, University of Innsbruck.

[Avanzini et al., 2013] Avanzini, M., Schaper, M., and Moser, G. (2013). Small Polynomial Path Orders in TcT. In *Proc. of 12th Workshop on Termination*, pages 3–7.

[Baillot et al., 2006] Baillot, P., dal Lago, U., and Moyen, J.-Y. (2006). On quasi-interpretations, blind abstractions and implicit complexity. In *8th International Workshop on Logic and Computational Complexity (LCC'06)*.

[Baillot and Terui, 2009] Baillot, P. and Terui, K. (2009). Light types for polynomial time computation in lambda calculus. *Information and Computation*, 201(1).

[Bellantoni and Cook, 1992] Bellantoni, S. and Cook, S. (1992). A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2.

[Ben-Amram, 2010] Ben-Amram, A. M. (2010). On decidable growth-rate properties of imperative programs. In *Proceedings International Workshop on Developments in Implicit Computational complExity, DICE 2010, Paphos, Cyprus, 27-28th March 2010.*, pages 1–14.

[Ben-Amram et al., 2008] Ben-Amram, A. M., Jones, N. D., and Kristiansen, L. (2008). Linear, polynomial or exponential? complexity inference in polynomial time. In *Proceedings of the 4th Conference on Computability in Europe: Logic and Theory of Algorithms*, CiE '08, pages 67–76, Berlin, Heidelberg. Springer-Verlag.

[Ben-Amram and Kristiansen, 2012] Ben-Amram, A. M. and Kristiansen, L. (2012). On the edge of decidability in complexity analysis of loop programs. *Int. J. Found. Comput. Sci.*, 23(7):1451–1464.

[Bonfante et al., 2001] Bonfante, G., Cichon, A., Marion, J.-Y., and Touzet, H. (2001). Algorithms with polynomial interpretation termination proof. *J. Funct. Program.*, 11(1):33–53.

[Bonfante et al., 2004] Bonfante, G., Marion, J.-Y., and Moyen, J.-Y. (2004). On complexity analysis by quasi-interpretations. In *2nd APPSEM II Workshop*.

[Bonfante et al., 2005a] Bonfante, G., Marion, J.-Y., and Moyen, J.-Y. (2005a). Quasi-Interpretations and Small Space Bounds. In Giesl, J., editor, *Rewrite Techniques and Applications*, volume 3467 of *Lecture Notes in Computer Science*. Springer.

[Bonfante et al., 2011] Bonfante, G., Marion, J.-Y., and Moyen, J.-Y. (2011). Quasi-interpretations a way to control resources. *Theoretical Computer Science*, 412(25):2776 – 2796.

[Bonfante et al., 2005b] Bonfante, G., Marion, J.-Y., Moyen, J.-Y., and Péchoux, R. (2005b). Synthesis of Quasi-interpretation. In *Proceedings of LCC'05*.

[Cobham, 1962] Cobham, A. (1962). The intrinsic computational difficulty of functions. In Bar-Hillel, Y., editor, *CLMPS*.

[Cocke, 1970] Cocke, J. (1970). Global common subexpression elimination. *SIGPLAN Not.*, 5(7).

[Cook et al., 2006] Cook, B., Podelski, A., and Rybalchenko, A. (2006). *Terminator: Beyond Safety*, pages 415–418. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Cousot and Cousot, 1977] Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California. ACM Press, New York, NY.

[Duprat et al., 2016] Duprat, S., MOYA LAMIEL, V., Kirchner, F., Correnson, L., and Delmas, D. (2016). Spreading Static Analysis with Frama-C in Industrial Contexts. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France.

[Eder and Gallagher, 2017] Eder, K. and Gallagher, J. P. (2017). Energy-aware software engineering. In Fagas, G., Gammaitoni, L., Gallagher, J. P., and Paul, D. J., editors, *ICT - Energy Concepts for Energy Efficiency and Sustainability*, chapter 05. InTech, Rijeka.

[Falke et al., 2011] Falke, S., Kapur, D., and Sinz, C. (2011). Termination Analysis of C Programs Using Compiler Intermediate Languages. In Schmidt-Schauß, M., editor, *22nd International Conference on Rewriting Techniques and Applications (RTA'11)*, volume 10 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 41–50, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Fog, 2010] Fog, A. (2010). *Optimizing subroutines in assembly language: An optimization guide for x86 platforms*. `http://agner.org/optimize/optimizing_assembly.pdf` (accessed version: 2010-09-25).

[Giesl et al., 2017] Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., and Thiemann, R. (2017). Analyzing program termination and complexity automatically with aprove. *Journal of Automated Reasoning*, 58(1):3–31.

[Girard, 1987] Girard, J.-Y. (1987). Linear logic. *Th. Comp. Sci.*, 50.

[Gonnord et al., 2015] Gonnord, L., Monniaux, D., and Radanne, G. (2015). Synthesis of ranking functions using extremal counterexamples. In *Programming Language Design and Implementation (PLDI)*, pages 608–618. ACM.

[Grech et al., 2015] Grech, N., Georgiou, K., Pallister, J., Kerrison, S., Morse, J., and Eder, K. (2015). Static analysis of energy consumption for llvm ir programs. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '15, pages 12–21, New York, NY, USA. ACM.

[Henry et al., 2012] Henry, J., Monniaux, D., and Moy, M. (2012). Pagai: A path sensitive static analyser. *Electron. Notes Theor. Comput. Sci.*, 289:15–25.

[Hofbauer and Lautemann, 1989] Hofbauer, D. and Lautemann, C. (1989). *Termination proofs and the length of derivations*, pages 167–177. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Hofmann, 1999a] Hofmann, M. (1999a). Linear types and Non-Size Increasing polynomial time computation. In *LICS*, pages 464–473.

[Hofmann, 1999b] Hofmann, M. (1999b). Linear types and non-size-increasing polynomial time computation. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 464–473.

[Hofmann and Jost, 2003] Hofmann, M. and Jost, S. (2003). Static prediction of heap space usage for first-order functional programs. *SIGPLAN Not.*, 38(1):185–197.

[Hofmann and Schopp, 2009] Hofmann, M. and Schopp, U. (2009). Pointer programs and undirected reachability. In *Proceedings of the 2009 24th Annual IEEE Symposium on Logic In Computer Science*, LICS '09, pages 133–142, Washington, DC, USA. IEEE Computer Society.

[Kristiansen, 2012] Kristiansen, L. (2012). Notes on code motion. manuscript.

[Kristiansen and Jones, 2009] Kristiansen, L. and Jones, N. D. (2009). The flow of data and the complexity of algorithms. *Trans. Comp. Logic*, 10(3).

[Kristiansen and Niggl, 2004] Kristiansen, L. and Niggl, K.-H. (2004). On the computational complexity of imperative programming languages. *Theor. Comput. Sci.*, 318(1-2):139–161.

[Kuck et al., 1981] Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B., and Wolfe, M. (1981). Dependence graphs and compiler optimizations. In *POPL*.

[Laird et al., 2013] Laird, J., McCusker, G., Manzonetto, G., and Pagani, M. (2013). Weighted relational models of typed lambda-calculi. In *IEEE/ACM LICS*.

[Lattner and Adve, 2004] Lattner, C. and Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California.

[Lee et al., 2001] Lee, C. S., Jones, N. D., and Ben-Amram, A. M. (2001). The Size-Change Principle for Program Termination. In *POPL*.

[Leivant and Marion, 1995] Leivant, D. and Marion, J.-Y. (1995). Ramified recurrence and computational complexity ii: Substitution and poly-space. In *Selected Papers from the 8th International Workshop on Computer Science Logic*, CSL '94, pages 486–500, London, UK, UK. Springer-Verlag.

[Mantere et al., 2009] Mantere, M., Uusitalo, I., and Roning, J. (2009). Comparison of static code analysis tools. In *Proceedings of the 2009 Third International Conference on Emerging Security Information, Systems and Technologies*, SECURWARE '09, pages 15–22, Washington, DC, USA. IEEE Computer Society.

[Moser, 2009] Moser, G. (2009). *Proof Theory at Work: Complexity Analysis of Term Rewrite Systems*. Habilitation thesis, University of Innsbruck.

[Moyen, 2001] Moyen, J.-Y. (2001). System Presentation: An analyser of rewriting systems complexity. In van den Brand, M. and Verma, R., editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier Science Publishers. RULE'01.

[Moyen, 2009] Moyen, J.-Y. (2009). Resource control graphs. *ACM Trans. Computational Logic*, 10.

[Necula, 1997] Necula, G. C. (1997). Proof-Carrying Code. In *Proceedings of POPL'97*.

[Nielson et al., 1999] Nielson, F., Nielson, H. R., and Hankin, C. (1999). *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Novillo,] Novillo, D. Tree ssa - a new optimization infrastructure for gcc. In *in 'Proceedings of the 2003 GCC Summit*, pages 181–194.

[Schopp, 2007] Schopp, U. (2007). Stratified bounded affine logic for logarithmic space. In *Proceedings of the 22Nd Annual IEEE Symposium on Logic in Computer Science*, LICS '07, pages 411–420, Washington, DC, USA. IEEE Computer Society.

[Seiller, 2016a] Seiller, T. (2016a). Interaction graphs: Additives. *Annals of Pure and Applied Logic*, 167.

[Seiller, 2016b] Seiller, T. (2016b). Interaction graphs: Full linear logic. In *IEEE/ACM LICS*.

[Song et al., 2000] Song, L., Futamura, Y., Glück, R., and Hu, Z. (2000). A loop optimization technique based on quasi-invariance.

[Spoto et al., 2010] Spoto, F., Mesnard, F., and Payet, E. (2010). A termination analyzer for java bytecode based on path-length. *ACM Trans. Program. Lang. Syst.*, 32(3):8:1–8:70.

[Ströder et al., 2014] Ströder, T., Giesl, J., Brockschmidt, M., Frohn, F., Fuhs, C., Hensel, J., and Schneider-Kamp, P. (2014). *Proving Termination and Memory Safety for Programs with Pointer Arithmetic*, pages 208–223. Springer International Publishing, Cham.

[Tavares et al., 2014] Tavares, A. L. C., Boissinot, B., Pereira, F. M. Q., and Rastello, F. (2014). Parameterized construction of program representations for sparse dataflow analyses. In Cohen, A., editor, *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8409 of *Lecture Notes in Computer Science*, pages 18–39. Springer.

[Tristan, 2009] Tristan, J.-B. (2009). *Formal verification of translation validators*. Theses, Université Paris-Diderot - Paris VII.

[van Gastel and van Eekelen, 2017] van Gastel, B. and van Eekelen, M. (2017). Towards practical, precise and parametric energy analysis of it controlled systems. In Bonfante, G. and Moser, G., editors, Proceedings 8th Workshop on *Developments in Implicit Computational complExity* and 5th Workshop on *FOundational and Practical Aspects of Resource Analysis,* Uppsala, Sweden, April 22-23, 2017, volume 248 of *Electronic Proceedings in Theoretical Computer Science*, pages 24–37. Open Publishing Association.

BIBLIOGRAPHY

List of Figures

List of Abbreviations

ANR	(French) National Agency for Research.
CFG	Control Flow Graph.
CFG	Context Free Grammar.
DepFG	Dependency Flow Graph.
DFG	Data Flow Graph.
DFA	Data Flow Analysis.
DLAL	Dual Light Affine Logic.
ELICA	Expanding Logical Idea for Complexity Analysis (project).
ICC	Implicit Computational Complexity.
IR	Intermediate Representation.
IL	Intermediate Languages.
LICM	Loop Invariant Code Motion.
LQI	Loop Quasi-Invariant.
LQICM	Loop Quasi-Invariant Chunk Motion.
NSI	Non Size Increasing.
PoC	Proof of Concept.
QI	Quasi-Interpretation.
QI	Quasi-Invariant.
RCG	Resource Control Graph.
RISC	Reduced Instruction Set Computer.
SCP	Size Change Principle.

SCT	Size Change Termination.
SSA	Static Single Assignment.
TAL	Typed Assembly Language.
TRS	Term Rewriting System.